The CAST Manual

John Yen*, Thomas R. Ioerger^, Michael S. Miller^, Shuang Sun*,
Kaivan Kamali*, Shizhuo Zhu*, Xiaocong Fan*, Richard. A. Volz^

Technical Report No. XXXXX

School of Information Sciences and Technology
The Pennsylvania State University

May 11, 2004

* 324 IST Building,
University Park, PA 16802
(814) 865-6178 (Tel)
(814) 865-5604 (Fax)

^ Department of Computer Science
Texas A&M University
College Station, TX 77843

Table of contents

1. Preface

CAST (Collaborative Agents for Simulating Teamwork) is a teamwork agent architecture that enables building collaborative agents as teams. This manual gives a high level introduction on CAST architecture, how to use CAST, and how to program with CAST. Readers may refer to [1] for more information on it from a theoretical perspective.

John Yen is the main architect for CAST. CAST version 1.0 was developed by Jianwen Yin. Michael S. Miller has developed the CAST version 2.0 that is currently used. Thomas R. Ioerger has coded JARE. At the Penn State University, CAST has been future extended on the basis of version 2.0 and different domain scenarios have been implemented and tested with CAST agents.

2. Getting started

System requirements

CAST v2.0 is developed with Java; the only requirement is JDK1.3.1 (or J2SE v 1.3.1 http://java.sun.com/j2se/1.3/). However, we recommend you install ant (http://ant.apache.org/), which will makes compiling and running easy.

Installation

When the package is unzipped, it will automatically create and copy the required files in the directory in which it was unzipped. You need to start the dTank simulator, an example domain, (under dTank-2.0.8a) before running the test. A dTank manual is available at http://acs.ist.psu.edu/dTank/.

Compiling

Use the ANT build.xml file in the cast3/src directory. In this file there are a number of targets for both compiling and executing the CAST software. If ANT is set up then typing "ant-projecthelp' returns the following entries.

Buildfile: build.xml

Main targets:

- Build Builds jar file
- Cast-all Runs CAST with example XML file
- Clean Removes all classes and jar files
- Docs Creates Java docs

Default target: build

Running

There are the targets that can be executed using ANT. ANT can both compile and execute cast3. Typing 'ant' with no arguments will compile the code. CAST has an XML parser that it uses to parse a configuration file. This reduces the amount of setup needed as a user can simply use the XML file to specify what agents will be started and their attributes along with a few CAST system attributes.

To Run CAST agents:

If you have installed ant:

Type "ant jtankstart" under the sub-folder "Cast"

If you do not have ant installed:

Type run_jtank or

Type java -classpath ..\lib\lib-cast.jar;..\classes -Djava.security.policy=cast.policy cast3.monitor.CastMonitor config\jTank\jTank.xml cast3.domain.JTankWorld.JTankAgentStarter under the sub-folder "run"

Testing

After you run the CAST agents in the agent monitor, you may click the button "unpause all" to run them.

Configuration

Under run folder, there are three sub-folders: config, images and teamplans. Config folder consists of the configuration files of the team, which are saved in XML files.
Images folder consists of the images for displaying the world simulator. Teamplans folder consists of the plans of the team, which are saved as MALLET files. Cast agents are configured with an XML file e.g. “config\jTank\jTank.xml”, which contains the configuration of the team of the agents. For every agent, we should explicitly indicate at least the NAME, DOMAIN, MALLET, MONITOR, KBTYPE, WORLD_HOST, and WORLD_HOST_PORT.

MALLET Plan

In the example tank scenario, the procedural knowledge contains a two-phase plan: search and attack. Initially, a team wanders around and searches for enemy tanks. Once the team finds an enemy, the team-members communicate to inform each other of the enemy location and to cooperate for attack. Next, the team attacks the target together until the target is destroyed. Such a process iterates until all the enemies have been destroyed. Declarative domain knowledge includes moving directions, stone locations, and so on.

3. About CAST

3.1. Overview

CAST is a teamwork architecture that enables building collaborative agents as teams. The agents can monitor the progress of team activities, synchronize their behaviors, and, more importantly, anticipate the needs of their teammates. CAST uses a high-level language called MALLET (a Multi-Agent Logic-based Language for Encoding Teamwork) to capture teamwork knowledge flexible enough for the team to adapt to a dynamic environment and efficient enough for the team to coordinate and assist with limited communications. In addition, each agent uses novel algorithms to (1) dynamically assign tasks to member of the team, and (2) to anticipate information needs of teammates based on the shared knowledge.

3.2. Main features

The central aspect to our architecture is the maintenance of a Shared Mental Model (SMM) among the agents. The SMM of a CAST agent has three components. First, team structure and team process knowledge is described in MALLET. The structure knowledge describes roles in the team, agents in the team, and the role each agent can play. The process knowledge describes what the team is planning to do and a plan of how the team is to accomplish its goals. All agents have a copy of these plans, and hence know what is to be done. Second, a MALLET parser compiles the teamwork knowledge into a PrT Net representation, which is an internal representation of the agent's SMM about the status of the team's process. PrT nets as a high-level formalism are more expressive than traditional PrT nets (i.e. place/transitions nets) [2].

The third component of the agent's SMM is a knowledge base that reasons about the agent's belief regarding the world and the structure of the team. This knowledge base also contains domain knowledge of the agents. The knowledge base of an agent is initialized by facts and domain knowledge known to the agent. However, it is continuously updated by sensor inputs and communication messages received by the agent. Therefore, even if a team of agents starts with the same knowledge base, they will evolve into different (but overlapping) ones as they sense different information from the environment and receive different messages from teammates.

3.3. CAST architecture

The CAST framework is designed to model well-structured agent teams and to be able to adapt to dynamic environments. There are five major integrated components in the CAST architecture to support these objectives: *teamwork knowledge specification, coordinated plan execution, world model, communication, and domain adapter*. The *teamwork knowledge specification* allows agent designers to use MALLET to design team structures and team process of the team. At the same time, domain knowledge is captured in the agent's world model. The *coordinated plan execution* module makes action decisions and communication decisions. The CAST agents operate in a distributed fashion and they use JAVA Remote Method Invocation (RMI) for communication. Each

agent interacts with domain through a *domain adapter*, which integrates the four other components and makes CAST agents adaptive to different domains. An overview of the CAST agent architecture is shown in Figure 1.

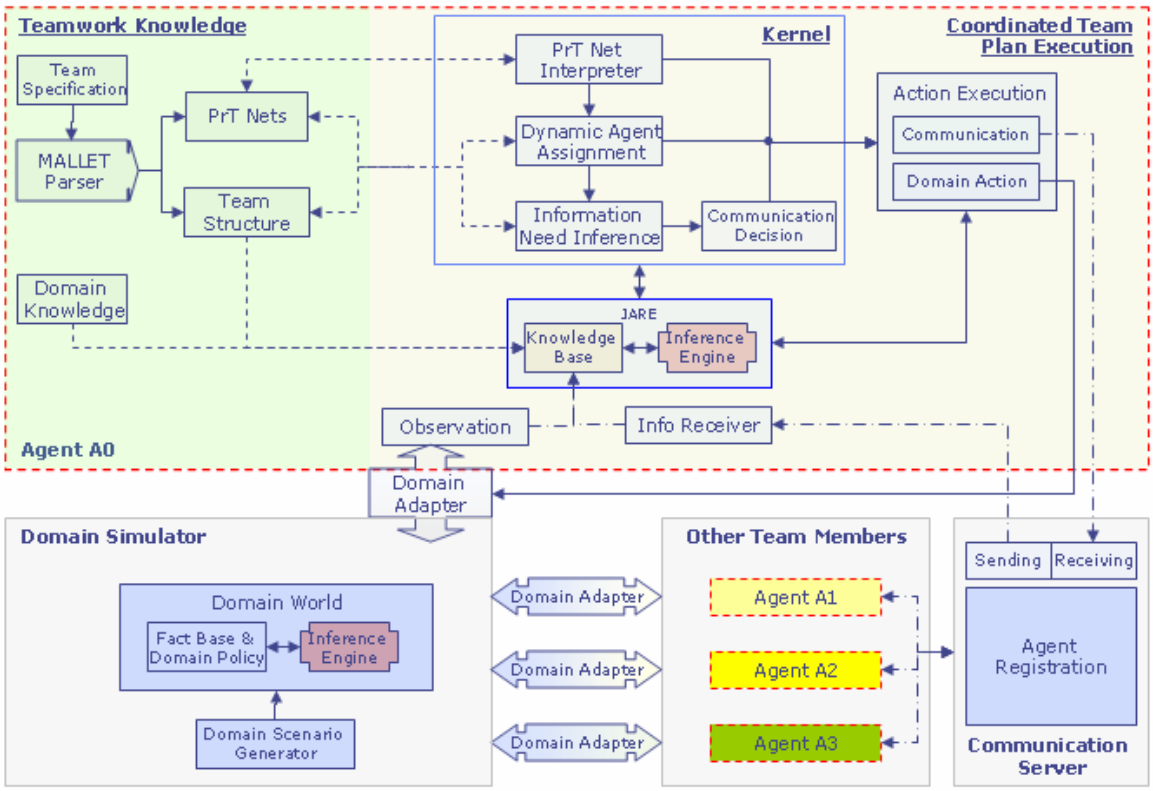

Figure 1: The CAST agent architecture.

Teamwork knowledge representation

Agent designers can use the team specification language MALLET to define team structures and team plans. MALLET supports most of the crucial elements of defining a team's organizational structure such as members of the team, roles and responsibilities, capabilities of agents, sub-teams, etc. Team structure information, once loaded into the agent knowledge base, becomes team members' shared mental knowledge about the team, which is important to help team-agents to reason about their teammates for task allocation and proactive communications.

The main function of MALLET is to represent teamwork knowledge (plans) for a team. The plans are organized in hierarchical structures such that each plan consists of one or several steps, which are either sub-plans or atomic operators. Those steps are controlled by different types of processes, which are specified as sequential, parallel, selective, iterative, or conditional. The whole team plan is converted into a PrT net, a process network which controls the execution of the team plan. Pre-conditions and post-conditions are defined for each of the sub-plans and operators. Pre-conditions specify the resource and information required for carrying out certain tasks or operations. For

example, it is a pre-condition that an attack plan requires the target location before an agent can carry out the task. The effect of the task is regarded as a post-condition, which, in this example, might be that the target is destroyed. Later, we will explain how the pre-condition information is used for the agent to infer about information need of other teammates.

In addition to team design, domain knowledge is also coded and loaded to the agent knowledge base. Domain knowledge includes both facts about domain and inference rules as Horn-clauses, with which agents can use to make decisions or to check the pre-conditions of tasks in the PrT net.

The team structure and team plans in MALLET, and coded domain knowledge will be parsed into a collection of PrT nets and an agent knowledge base. Both the process and knowledge about team and domain are important mental states that are shared among the whole team. Based on these shared mental states, agents can anticipate the needs of other team members and demonstrate strong intelligent team assistance behaviors.

JARE

JARE (Java Automated Reasoning Engine) is a back-chaining theorem-prover for making inferences using knowledge that is written in the form of a separate Horn-clause knowledge base, which stores agents' beliefs. Initially, an agent has domain knowledge and belief about the team. After the agent is launched in the domain environment, the knowledge base is updated dynamically by newly acquired information through agents' observations and communications. Post-conditions can also update the agents' knowledge base with the effect of actions that were just carried out. Furthermore, an agent can update its own beliefs after making certain decisions. JARE is mainly used to determine the truth-value of conditions or constraints that need to be evaluated in interpreting MALLET expressions at run-time. JARE is also used to bind variables through queries for plan instantiation, conditional execution, and making communication decisions.

CAST kernel

The kernel of CAST architecture includes a set of integrated algorithms (goal selection, dynamic task allocation, communication decision). It also controls the agent sensing, decision making, and action operation processes.

Dynamic agent assignment

During the execution process, agents select tasks to execute according to the PrT net and the beliefs about the team and domain. If the selected task is a sub-plan, the agent will set it as the current task that the team needs to accomplish. At the same time, the agent will try to coordinate with other agents who are also assigned to this task. The plan is represented in a PrT net that is generated from MALLET.

The algorithm for task allocation in CAST is called dynamic agent assignment (DAA). DAA first determines the current goal and finds an associated plan using the goal selection process. Each plan, defined in MALLET, specifies the constraint conditions on how the tasks in this plan are allocated. Such conditions may include role constraints, workload conditions, or domain related conditions. Before executing a task, each agent

will check these constraints with its current beliefs about the domain or teammates and determine the suitable agents whose conditions satisfy the constraints. Agents' workload status will also be considered during this process. The result of the DAA will determine which agents are going to do the task, and will play an important role in updating beliefs regarding information needs that agents use to provide information proactively for their teammates.

Communication decision

The algorithm used to infer information needs is called the Dynamic Inter-Agent Rule Generator (DIARG). DIARG is the most crucial part of CAST agent in terms of achieving the proactive assistance behavior in team environment. The foundation of DIARG algorithm has been established by extending the SharedPlan theory with the formal semantics of proactive communicative actions [3, 4]. We have shown that an agent's consideration of proactive assist behaviors can be derived from axioms in the formal framework. A team's SMM about the information needs of its members is captured in the framework by a model operator InfoNeed (*A*, *I*, *t*, *Cn*), which represents that agent *A* needs information *I* at time *t* under the context *Cn*. The arguments of these model operators correspond to attributes of the information need table, which serves as a computational SMM about information needs for a team of agents.

DIARG has an off-line component and an on-line component. The off-line component generates the information need table using teamwork knowledge specified in MALLET. The table contains information requirements (preconditions) for each task, and candidate agents who are potentially responsible for the task. When an agent is dynamically assigned to a task, the content of the table will be updated according to the result of DAA. Using this table, agents can anticipate what information other agents might need. DIARG is built on top of the theory for proactive information exchanges [4], and is used to anticipate action performing information needs.
The on-line component of DIARG monitors information newly sensed by an agent to determine whether it matches information needs of teammates. If there is a match, the agent decides whether to proactively inform teammates. Communications could introduce risks to the agent (e.g., that an enemy could intercept the communication and learn information potentially harmful to the team). Therefore, agents should consider the risk as part of the communication cost in making their decisions on whether to proactively communicate with teammate or not.

Domain adapter

The domain adapter acts as an interface to the simulation environment. Once agents make all of the decisions, they can carry out action operators. The CAST agent kernel does not define domain operators. A PrT net is executed at the abstracted operator level. The execution of the operators will invoke a set of separate domain operator definition class which is integrated with a domain adapter. The function of the domain adapter is to provide domain application layer communication such as API functions or messages. The domain adapters are implemented by dynamic class loading and can consist of arbitrary Java code. The domain definition class and domain adapter make CAST architecture adaptable to not only different domains but also heterogeneous network environment.

Agent communication

Like most multi-agent architectures, messages from CAST agents are encoded in KQML (Knowledge Query Manipulation Language) [5] format that defines both the content of the messages and performatives of the communication. CAST currently supports two kinds of performatives: (1) information exchange performatives, and (2) coordination performatives. Proactive delivery of information needed by teammates is realized using a novel performative called ProInform. The semantics of ProInform extend that of Inform to include the needs of addressee in the mutual belief the speaker attempts to establish with the addressee. The semantics of performatives have direct impact on the conversation policy between agents in a team. For instance, a new type of reply can be introduced for ProInform such that the addressee can indicate that the information delivered is not needed or contrary to what the speaker believes. We name the performative of this reply "RejectNeed". A more detailed discussion about the semantics of proactive communications among teammates can be found in [4].

The second type of performatives is for agents to coordinate with each other regarding the execution of a team process. For example, at the conclusion of a team operation, there will be communication by the performer(s) to let other members of the team (in some cases, not all members of a team need participate in a team action) know that the operation has been completed and it is permissible to proceed. A future optimization could take observability into account and only inform teammates who were believed unable to observe the performance of the action.

The actual agent communication is placed in a module outside the CAST kernel because the lowest level mechanisms used are domain dependent. In our test simulation worlds, we have used JAVA RMI and socket level communications. In a domain in which CAST is integrated into an operational setting, e.g., to provide assistance to live human operators, other forms of communications (e.g., voice, images, and video) might come into play, as well. For purposes of this paper, we simply assume that the communication module provides reliable communication among team members when called upon to do so. Most importantly, on the receiving end, it sends received information to a receiving agent within the Kernel that can interpret the information received and update the JARE knowledge base.

Agent interaction with domain

The domain or simulation environment is independent of CAST kernel. A typical domain contains its fact base and some policies. To interact with the domain, a CAST agent requires that domain environment periodically provide new "(sensing) information" and accept agent domain action commands. Each agent has its own instance of the CAST Kernel and domain adapter and interacts with the domain independent of the interaction of other agents with the domain.

The CAST Kernel is designed to be domain independent so that CAST can be readily adapted to different domains. In order to use CAST with a specific domain, one needs to do three things: 1) develop a domain specific adapter, 2) encode necessary domain knowledge in JARE rules, and 3) develop the teamwork plans, which will usually have domain specific aspects. CAST provides a template which can be used as a basis for

developing the domain specific adapters. Domain operations that agents can perform are specified in an interface, and the adapter must provide an implementation. The operations used in MALLET programs must conform to the interface. CAST then simply determines the operation names and parameter profile by parsing the MALLET plan and invokes the methods through the interface when it determines that an operation should be performed. The interface also specifies methods by which domain information can be acquired to satisfy JARE queries.

Execution

An important issue for CAST kernel to address is to achieve desired coordination and synchronization in a distributed manner. Synchronization for a team operator is established using the following conversation policy:

1. Each agent involved in the team operator (i.e., an actor) sends a synchronization message to other actors of the operator when it is ready to perform the operator (i.e., after it evaluates the operator's precondition successfully).
2. After the number of synchronization messages received by an actor exceeds or equal to the minimum number of agents required to performing the operator (as specified by the num constraint of the team operator in MALLET), the actor performs the team action.

Similar synchronization scheme is also used for agents to synchronize in invoking a team plan using *JOINTDO*. This simple synchronization scheme does not consider communication delays. Alternative synchronization scheme will be needed if agents need to synchronize in an environment that communication delays are not negligible.

Distributed coordinations of a team plan are realized by "coordination messages" between members of the team. Each coordination message contains an identifier that refers to a specific task (i.e., operator or subplan) in a team plan. We describe the communication protocol for achieving coordination below.

- If an agent is not an actor for the current task (i.e., operator of plan) in the team plan, the agent waits for coordination messages associated with the task from teammates.
- If the agent is an actor for the current task, it starts performing the task, synchronize with teammates if needed.
- When the agent finishes the task, it sends a "coordination message" (containing the identifier of the task) to all other teammates involved in the team plan.
- When an agent waiting for a coordination message receives the message, it proceeds to the next step of the team plan.

The parallel process in MALLET currently specifies parallel activities in a team, but not parallel activities for an agent. Therefore, each agent can choose only one parallel branch to enter. If an agent can enter multiple parallel branches, the current implementation of CAST randomly chooses one of these branches. This selection scheme is simple, but does not address the issue of potential deadlocks. For instance, suppose three agents A, B, and C enter a parallel process with three branches. Suppose the first branch invokes a team plan that requires A and B, the second branch requires B and C, the third branch requires C. Assuming agent A chooses the first branch, agent B chooses the second branch, and agent C chooses the third branch. Only agent C will be able to proceed, while agent A and B each waits for another teammate indefinitely to execute their branch.

This problem can be addressed in the future in two ways: (1) by developing algorithms to detect potential deadlocks in a MALLET team plan specification and, (2) by improving the coordination among agents in choosing parallel branches.
The CAST kernel includes a set of algorithms that CAST agents can use to determine the actions (including domain actions and communication actions) to be performed at each time step. Such algorithms include PrT net interpreter, DAA (dynamic agent assignment), DIARG (protell based on information flow table), decision-theoretic communication strategies.
The PrT net interpreter is used for *interpreting* (*manipulating)* the PrT nets so that all the team agents cooperate their behavior to embody the specified team behavior. The internal PrT net representations of team processes are generated offline by MALLET Parser through compiling the team plans specified in MALLET. Initially, a team plan is only partially instantiated, though fully represented in the PrT net. As well as initiating appropriate communication actions at the appropriate time to evolve these partial plans (determine parameters such as the actual doers of certain actions), one of the tasks of NetIntepreter is to ensure all the team members to behave, react, and deliberate strictly according to the committed (intended) plans (courses of actions), and synchronize their behaviors whenever necessary.

At each computation step, CAST agents sequentially execute three (mental) actions: *sense-decide-act*. During the *sense* phase, each individual agent queries the world simulation server to update its knowledge of the current state of the world, check its message queue and process the messages from other agents in this round. During the *decide* phase, by using NetIntepreter, each agent examines the PrT net representation of the team plan to see if there are any pending actions for which it is responsible. In cases of ambiguity about the actual doers of some pending actions, the agents might have to communicate in order to determine who will take the action, and synchronize their behavior if necessary. During the *act* phase, the kernel dynamically loads the class for the committed operation type, creates an instance of it, and invokes the operation. In addition, the kernel also applies the effects of the operation to its knowledge base, and if needed, proactively informs the new generated information to its teammates.
More specifically, the kernel accomplishes the *sense* phase by invoking operations in the domain dependent library to obtain the needed information from the Domain Simulator. The Domain Simulator determines which agent needs what information through analyzing relevant JARE rules and the specification of observabilities pre-defined in MALLET programs. As the Domain Simulator updates itself (the actions executed by individual agents affect the world state), it notifies the appropriate Domain Adapter, which forwards the information to the corresponding kernels (agents), which then update their belief base (maintained by JARE Engine) to reflect the new acquired information (See Figure 1).
The *decide* portion is an area which exposes promising research topics in CAST agent architecture. Amongst the features that have been developed and tested thus far are:

- Based upon the analysis of the MALLET programs, every CAST agent uses the DIARG algorithm (offline part) discussed earlier for determining agent- information needs relation and agent-information production relation, which are captured by its information-flow table.
- Upon acquiring new information (sensed from environment or derived from effects),
    1. Based on the common knowledge of observabilities (expressed in JARE), its information-flow table and other beliefs, an agent determines whether the

information needers among its teammates can get the same information by themselves (and hence the agent doesn't need to inform them).

2. When considering communication cost (risk), an agent will decide whether it should proactively inform the new acquired information to the needers by evaluating the utility of protell vs. not-protell.

- When lacking of certain information,
    1. An agent can determine the potential information providers by checking its information-flow table.
    2. When considering communication cost (risk), an agent will decide whether it should proactively ask an information provider by evaluating the utility of ask vs. not-ask.
- When executing along the shared PrT nets,
    1. By analyzing the currently active PrT nets, determine the actions within the team that are to be performed next (refer to algorithm 1).
    2. In the case of team operations or joint (*AND*/*XOR*/*OR*) operations, synchronize with teammates as necessary.
    3. Upon dynamic agent assignment, determine which agents will be dynamically assigned to some actions lacking pre-specified performers (this may involve coordinating with other agents).
    4. Determine whether or not this agent (each agent does such an analysis) is to perform the chosen friable next action.

In short, during the decide phase, each agent determines what operation, if any, it should perform during its next cycle; performing, if necessary, dynamic agents assignment, and determines what communication should take place with which other teammates.

4. MALLET

MALLET (Multi-Agent Logic Language for Encoding Teamwork) is a logic-based language for specifying the structures and processes of agent teams. MALLET syntax is based loosely on LISP, in the sense of using s-expressions and prefix notation. Variables are indicated with '?' prefix. The full syntax for MALLET is shown in Appendix A.

4.1. Team structure knowledge

At the top level, MALLET allows expression of knowledge about team structure in terms of:

- Agent-role relationship, i.e. which agent plays which role.
- Membership of a team

For example, the following specification defines the team *attackteam*, which has two members, F0 and U0, playing different roles (i.e. fighter and bomber, respectively).

```
(plays-role F0 (fighter))
(plays-role U0 (bomber))
(team attackteam (F0 U0))
```

The actions agents are able to perform are defined in terms of operators. Operators, specified by preconditions and effects, are classified into individual and team operators. Individual operators are executed by only one agent at a time, whereas team operators are performed by a *set* of agents (e.g. those playing a given role). For example, the following specification defines individual operator *move*:

```
(ioper move (?dir)
(pre-cond (at self ?x ?y) (can_move ?dir ?new-x ?new-y))
(effects (at self ?new-x ?new-y) (not (at self ?x ?y))))
```

This operator describes an action for the agent to move a step along a direction. The precondition of the operator includes two predicates, connected implicitly by logic conjunction operator (i.e., AND). The first predicate retrieves the current location of the agent (referred in MALLET by the keyword "self"). The second predicate tests whether the agent can move in the direction given, and if so, binds the values of such a move to the variables ?new-x and ?new-y. The evaluation of the test involves Horn clauses in JARE that considers obstacles and the boundary of the world in which the agent navigates. After the move, the operator's effect updates the location of the agent.

As a second example, the following defines a team operator *co_fire*, which involves at least two agents firing at a given coordinate (specified by the arguments ?x and ?y) simultaneously:

```
(toper co_fire (?x ?y)
        (num ge 2)
```

• • •

)

The *num* constraint specifies the number of agents that should be involved in the team operator. Obviously, agents involved in a team operator need to be synchronized. This is achieved by the CAST kernel

The capabilities of individual agents can be specified either explicitly via a capability specification or indirectly through a role specification. For example, to state that F0 and U0 have the same capabilities, and are able to perform actions move and detectEnemy, one would have:

```
(capability (F0 U0) ((move ?dir) (detectEnemy)))
```

Alternatively, one could state:

```
(role fighter ((move ?dir) (detectEnemy) (shoot ?x ?y)))
(role bomber ((move ?dir) (detectEnemy) (co_fire ?x ?y)))
```

where fighters have the additional capability to shoot an enemy fighter and bombers have an additional capability to bomb an enemy base through the *co_fire* team operator.

4.2. Process knowledge

Operators are the basis for the hierarchical construction of team plans (i.e. operators are atomic actions in a plan hierarchy). Plans essentially characterize team processes. A plan in MALLET is composed of precondition, effects, and termination conditions, constraints for task allocations, and the procedural description of the process. The precondition, represented by a logical conjunct, expresses a necessary condition under which a plan (or an operator) can be performed. The effects, also represented by a logical conjunct, state the conditions that hold after the plan is successfully accomplished. The termination conditions describe two kinds of conditions for terminating a plan: the success conditions and the failure conditions. The constraints for task allocation are specified by the *AGENTBIND* construct.

The process of a plan describes the procedure of how a team will accomplish their task. To be expressive, MALLET provides a rich set of constructs to define such procedures. A process consists of invocations of operators or plans, or arbitrary combinations using various constructs such as sequential, parallel, conditional, or iterative, blocks, etc. For example, the following is a high-level MALLET plan that characterizes the teamwork process for multiple teams to find and attack enemy base. The plan is the second stage of an overall plan. The first stage is to search and identify the location of the enemy base, and the second stage is to attack the enemy base. Therefore, this plan includes the location of the enemy base as a precondition.

```
(plan Attack-Enemy-Base ()
  (pre-cond (is-enemy-homebase ?h1) (at ?h1 ?x ?y))
```

```
  (process
    (par
      (do scoutteam (seeking))
      (do fighterteam (attacking-enemy))
      (do bomberteam (attacking-base ?x ?y))
    ); endpar
  );endproc
);endplan

(plan attacking-enemy ()
(pre-cond (enemy ?e) (at ?e ?x ?y))

        • • •
)
```

The keywords *SEQ* and *PAR* are used in MALLET to describe a sequential process and a parallel process respectively. For instance, the Attacked-Enemy-Base plan described above involves a parallel process with three branches. These three branches describe subplans for three subteams. The first branch specifies that the scout subteam should search for the enemy. The second branch states that the fighterteam should attack enemies detected by the scouts. The third branch specifies that a team of bomber agents should attack the enemy base. The precondition of the attacking-enemy plan includes the location of enemy, i.e., (at ?e ?x ?y). Therefore, a scout agent will proactively deliver the location of detected enemy to the fighterteam. Other types of MALLET processes include conditional, i.e., (*IF* statement), iteration (i.e., *WHILE* statement), selection (i.e., *CHOICE* statement). Components of these processes are other MALLET process. Hence, the MALLET language allows any processes to be embedded in another process to form more complex processes. An important feature of MALLET is its flexibility in specifying the actors of a process using *DO* and *JOINTDO*. We describe each of these MALLET constructs below. The formal BNF-style notation of the MALLET process is as follows:

```
<MalletProcess> ::=
(<OperatorInvocation>) |
(<PlanInvocation>) |
(SEQ ( <MalletProcess> )+) |
(PAR ( <MalletProcess> )+) |
(IF (<COND> ( <Cond>)+) <MalletProcess> [(<MalletProcess>)]) |
(WHILE ( <COND> ( <Cond>)+) <MalletProcess>) |
(FOREACH (<COND> (<Cond>)+) <MalletProcess>) |
(FORALL ( <COND> ( <Cond>)+) <MalletProcess>) |
```

(CHOICE (<MalletProcess>)+) |
(JOINTDO (<AND> | <OR> | <XOR>) ((<ByWhom> <MalletProcess>))+) |
(DO <ByWhom> <MalletProcess>) |
(AGENTBIND (<AgentVars>) (CONSTRAINTS (<Cond>)+))

The *FORALL* construct is an implied *PAR* over the condition bindings, whereas the *FOREACH* is an implied *SEQ* over the condition bindings. These process types are fairly expressive when the number of choices is unknown before runtime. *FOREACH* is most useful in giving a single agent a list of tasks with different arguments and *FORALL* can be used when multiple agents are needed, but their numbers are unknown. The *CHOICE* construct is a control structure that takes a list of processes and executes them in order until one completes successfully.

The *AGENTBIND* construct introduces flexibility to the teamwork process in that the agent selection is made dynamically at runtime to assure the satisfaction of certain teamwork constraints, such as finding an agent that is capable of some operation. The values for agent variables are to be assigned so as to satisfy the constraints. For example,

(AGENTBIND (?fi)
(CONSTRAINTS (playsRole ?fi fighter) (closestToEnemy ?fi)))

states that ?fi will be assigned to an agent that plays the role of fighter and is closest to the enemy. The selected agents are then responsible for performing later steps (operators, sub-plans, or processes) associated with the agent variables. 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. An *AGENTBIND* statement of an agent variable can occur anywhere in a plan, as long as it precedes the first *DO* statement in which the agent variable is an actor.

The *DO* statement in a MALLET plan provides two functions: (1) it invokes a process; (2) it specifies the actor(s) for the process. Syntactically, the actor specification can be a list of agent variables or the name of a team/subteam. Semantically, the agents included in the actor specification should be sufficient for accomplishing the process. From teamwork perspective, the actor specification in MALLET enables a knowledge engineer to describe the subteam for carrying out a team plan involving multiple agents.

The *JOINTDO* 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*, one or more subprocesses must be executed. A *JOINTDO* statement is not executed until all involved team members have reached this point in their plans. Furthermore, the statement following a *JOINTDO* statement in the team process does not begin until all involved team members have completed their part of the *JOINTDO*.

5. JARE

JARE (Java Automated Reasoning Engine) is a back-chaining theorem-prover for making inferences using knowledge that is written in the form of a separate Horn-clause knowledge base, which stores agents' beliefs. Initially, an agent has domain knowledge and belief about the team. After the agent is launched in the domain environment, the knowledge base is updated dynamically by newly acquired information through agents' observations and communications. Post-conditions can also update the agents' knowledge base with the effect of actions that were just carried out. Furthermore, an agent can update its own beliefs after making certain decisions. JARE is mainly used to determine the truth-value of conditions or constraints that need to be evaluated in interpreting MALLET expressions at run-time. JARE is also used to bind variables through queries for plan instantiation, conditional execution, and making communication decisions.

5.1. Syntax of JARE language

Jare uses a LISP-oriented syntax, with nested lists and infix notation (where the first element in each list is often the name of a predicate, and operator, etc.). Capitalization and white-spaces generally do not matter. Comments are indicated by semi-colons (';'), which cause disregard of all the remaining characters on the line.

The basic unit of expression in Jare is a predicate. As is first-order logic, predicates have a predicate name followed by a list of arguments, called terms. However, the predicate name is written inside the parentheses as the first member of the list. Terms can either be constants (symbols or numbers), variables, or functions. Variables are indicated by symbols prefixed with a '?', such as ?x or ?temp. Functions are like predicates, in that they are lists with function names and arguments, though they occur as arguments inside other functions or predicates. Here are some examples:

- (teaches bill ai-course)
- (has-phd bill)
- (sister judy ?sis)
- (loves jude (mother-of judy)) - loves is a predicate, mother-of is a function

Predicates can also be negated by enclosing them in another list, whose first element is 'not'. Predicates and negations of predicates together are called "literals." Predicates without negations are sometimes called "positive literals," while negated predicates are sometimes referred to as "negative literals."

```
(not (warm december)) - a negative literal
```

Sentences in Jare are based on *Horn clauses*, which are made out of one or more predicates. When a Horn clause contains a single literal, it acts as a *fact* (note, it has an extra set of enclosing parentheses). Facts by themselves cannot be negated. Here are examples of two facts:

```
((warm july))
((cold december))
```

When 2 or more literals are included, the Horn clauses can be read as a *rule*. Specifically, the first literal is known as the "head" (or consequent) of the rule, and remaining literals are called the "body" (or antecedents). A rule can be understood by reading in a "backwards" way by saying "if" after the head. For example, consider the following rule, which says that something is a good deal IF it has high quality and a low price:

```
((good-deal ?x) (quality ?x high) (price ?x low))
```

The antecedents are implicitly conjoined ("and-ed" together). There is no way to encode disjunction (except that multiple rules are alternatives to each other). The head of each rule must be a positive literal, but any of the other members of the clause (antecedents) may be negated. Note that there are no quantifiers; any variables are assumed to be universally quantified (read: forall). It is of central importance that the head of each clause be a positive literal (i.e. does not contain 'not'). In particular, facts must always be positive literals.

Here is a brief list of commands you can run in Jare:

- (query *conjunction*) - finds all solutions and prints out query with each set of bindings
- (query-all *conjunction*) - same as query
- (query-one *conjunction*) - finds first solution and prints out query with bindings substituted in
- (assert *predicate*) - adds fact to knowledge base
- (retract *predicate*) - retracts ALL clauses whose head unifies with given predicate (may have constants and/or variables)

```
>(query (dog fido))
((dog fido))

>(query (cat fido))
fail

>(query (animal fido))
((animal fido))

>(query (animal ?a))
((animal fido))
((animal fifi))
((animal tweety))
((animal opus))

>(assert (dog rex))
asserting: [dog, rex]

>(assert (dog poochie))
asserting: [dog, poochie]
```

```
>(retract (bird ?x))
retracting: [bird, ?x]

>(query (animal ?a))
((animal fido))
((animal fifi))
((animal rex))
((animal poochie))
```

5.2. List of features

In this section, several important features implemented in Jare are discussed. The first is negation. While technically, Horn clauses are not supposed to contain negative literals, Jare allows antecedents in a rule to be negated. Syntactically, a negative literal is written by enclosing a predicate in another list with 'not' as the prefix (an example is shown above). The semantics of negation is handled exactly like Prolog: through *negation-as-failure*. That is, an antecedent (not (P)) is found to be true exactly when (P) cannot be proved. Jare recursively starts a new search for a proof of (P), and then continues the original proof if and only if no solution for (P) is found. Note that the proof of (P) is carried out in the current binding environment (with current substitutions), but the fact that the original proof proceeds only if the proof of (P) fails means that the binding environment itself will not be modified. As an example, here is a rule with a negative antecedent. It says that something is a week-day if it is a day but not a weekend. Days could be 7 facts enumerating Sat, Sun, Mon, ... Each will be tried, but those that satisfy weekend (i.e. Sat and Sun) will be filtered out as solutions.

```
((week-day ?x) (day ?x) (not (weekend ?x)))
```

The next major feature of Jare is *procedural attachments*. Procedural attachments are predicates with special pre-defined meanings that are essentially implemented in Jare code. Currently, there are two classes of procedural attachments: math operations, and list operations.

Math operations include two types: relations (e.g. equality) and functions (e.g. +). Relations are generally 2-argument predicates. Both arguments must be bound (i.e. you can't evaluate a relation on a free variable; Jare will signal an error if you try), and both arguments must be numeric. Here is a list of relational predicates:

- (= ?x ?y)
- (< ?x ?y)
- (> ?x ?y)
- (<= ?x ?y)
- (>= ?x ?y)

Straight inequality may be implemented as: (not (= ?x ?y)). Note: Jare attempts to convert anything that begins with a +, -, ., or digit into a float, and math operations are carried out on these (so 6.0 and 6 are equal). (However, ‘+’, ‘-‘, and ‘.’, may still be used as symbols themselves.) The predicate 'eq' can be used for comparing equality of symbols.

Math functions generally take 3 arguments. They can be used to verify relations if all 3 args are bound, or *any one* of the three may be left unbound and Jare will fill in the answer (expect mod and pow, for which only the third argument can be unbound). The implemented math functions are:

- (+ ?x ?y ?z)
- (- ?x ?y ?z)
- (* ?x ?y ?z)
- (/ ?x ?y ?z)
- (mod ?x ?y ?z)
- (pow ?x ?y ?z)

Here are some examples of using these math predicates:

```
>(query (+ 1 2 3))
((+ 1 2 3))

>(query (+ 2 3 ?x))
((+ 2 3 5.0))

>(query (+ 3 ?x 4))
((+ 3 1.0 4))

>(query (mod 9 4 ?x))
((mod 9 4 1.0))
```

Recently, we added some other useful math predicates, like sqrt, exp, log, mod, sin, cos, and tan.

Another class of procedural attachments are provided for support of list operations. In Jare, lists can be represented as terms within predicates by using a special function 'list' that can have any length. For example, the list of a, b, and c, is (list a b c), and the empty list is just (list). Currently, procedural attachments are provided only for basic operations, such as 'cons', 'first', and 'rest'. These allow you to construct and deconstruct lists. Then a larger set of additional list predicates is defined over this through axioms in list.kb, which is automatically loaded into Jare on startup.

Cons works as follows. You must give it an object and a list, and it returns a new list with the object stuck in the front. Hence cons takes a total of 3 arguments. If the 3rd argument is bound, cons verifies its correctness (i.e. succeeds if it is correct, else fails). More interestingly, if the 3rd arg is unbound, Jare binds the answer (new list) to it. First and Rest extract the corresponding elements from a list, returning an object or a list, respectively. Again, if the second argument is bound, its correctness is checked, else the variable is bound to the answer. Here are the basic list predicates with examples:

- (cons ?x ?y ?z) - (cons x (list a b) (list x a b))
- (first ?x ?y) - (first (list x a b) x)
- (rest ?x ?y) - (rest (list x a b) (list a b)

The additional list predicates defined through axioms include:

- (empty ?x)
- (length ?x ?n)
- (nth ?x ?n ?y) - ?n is the position, starts counting from 0
- (second ?x ?y)
- (third ?x ?y)
- (member ?x ?y) - checks to see if ?x is in the list ?y
- (reverse ?x ?y)
- (append ?x ?y ?z) - e.g. (append (list 1 2) (list a b) (list 1 2 a b))

Generally, only the last argument can be unbound for these predicates.

6. How to use CAST

CAST agents can be displayed and controlled through CAST monitor, a graphical user interface. The CAST monitor has two parts team monitor and individual agents monitor. The team monitor is used for controlling the team actions (viewing or adding team members, starting or suspending team perorations) and displaying the behavior of the team by listing the actions for each agent (see Figure 2). Each agent has an individual agent monitor, a tab captioned as the agent name. In addition to the functions that included in the team monitor, the individual agent monitor provides other comprehensive monitoring functions including KB query panel that allows a user to access agent's knowledge base, a PrT net display that allows a user to monitor the current process states. In the following sections, we use a firefighting domain scenario to illustrate how the CAST monitor works for a firefighter team. The CAST monitor includes the following tabs: *Monitor CAST Agents*, *Agent F1*, *Agent F2*, and *Agent Am*. F1, F2, and Am are corresponding to firefighter agent 1, 2, and an ambulance agent respectively.

6.1. CAST team monitor

Monitor CAST Agents is the panel for monitoring all individual agents' behavior and status. The functions of buttons are listed in table 1.

Table 1: Buttons in the team monitor window.

Button	Description
Add Agent	is reserved for dynamically adding agent to a team
Next Step	makes the agent proceed with one step
Unpause All	unpauses all the agents to let them proceed automatically; the button will change to Pause All after it is pressed

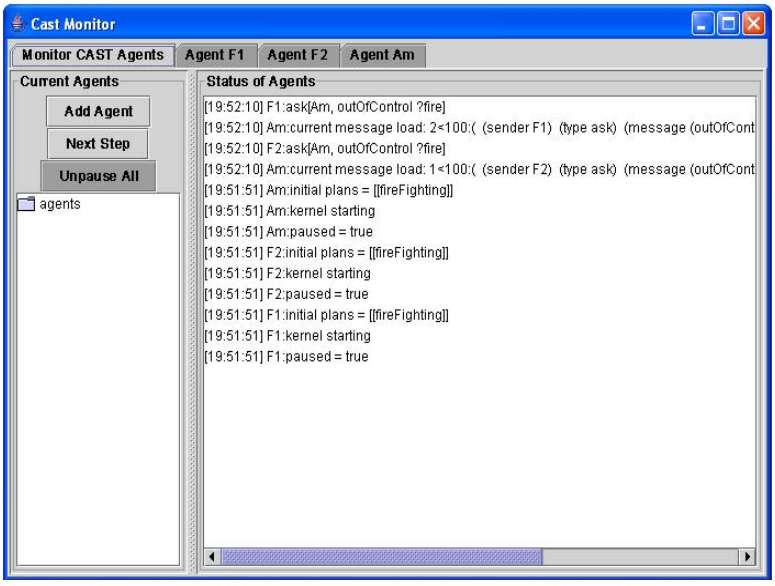

Figure 2: CAST team monitor.

6.2. Individual agent monitor

Under the individual agent tab, the information and control panels of the specific agent is displayed. The behaviors and status of this agent are displayed in the window on the left side. The buttons and their functions are listed in Table 2.

Table 2: Buttons in the individual monitor window.

Buttons	Description
Unpause/Pause Agent	unpauses this agent only, after unpaused, the button changes to Pause Agent
Show/Hide Process Nets	generates a tab for every plan of this agent, and under every tab is the process net for the plan; after it is pressed, the button changes to Hide Process Nets
Show MALLET Specifications	shows the MALLET specification in the window of *Knowledge Base Output*
Save as MALLET File	saves the MALLET specifications in the window of *Knowledge Base Output* as a MALLET File

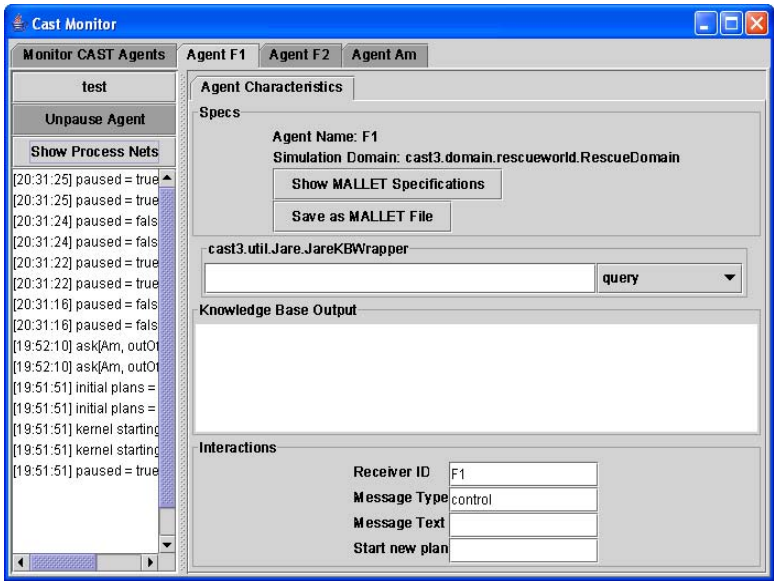

Figure 3: Individual agent monitor.

6.2.1. KB query panel

The KB query panel allows a user to access agent's knowledge base during runtime. Therefore it provides a powerful tool for agent developers to "debug" (watch and modify agent's knowledge) agent behavior dynamically. Figure 4 shows a sample out put and the list of query functions, which is explained below the Figure.

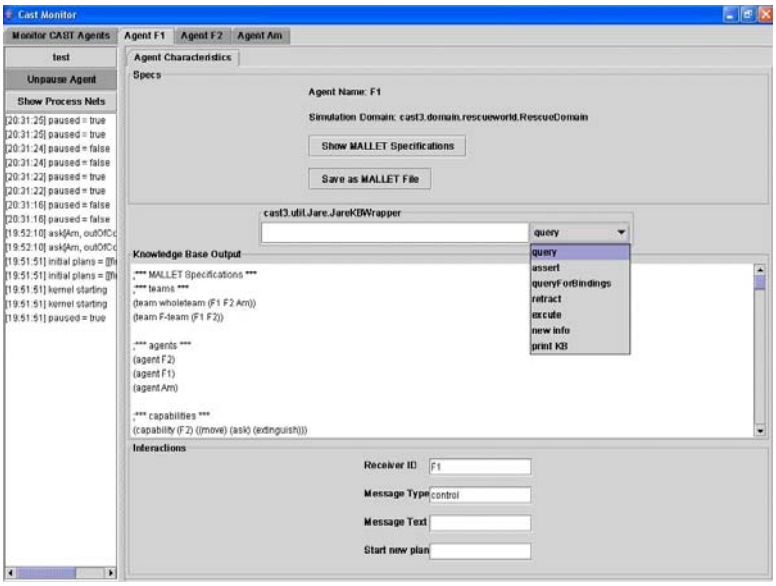

Figure 4: KB query panel.

- "*Query*" is for querying knowledge from the knowledge base. Input the sentence in the blank space with two brackets, and then press Enter. The output of the query will be displayed in the output window. If it is true, the queried sentence will be displayed again. If it is false, "*fail"* will be displayed.
- "*Assert*" is for inserting a new knowledge to the knowledge base. Input the sentence in the blank space with a bracket, and then press Enter. "*Asserting: S*" will be displayed in the output window, where S means the asserted sentence.
- "*QueryforBindings*" is for querying for agent bindings.
- "*Retract*" is for deleting a fact from the knowledge base. It is a reverse process of *Assert*.
- "*Execute*" is for dynamically loading a file.
- "*New info*" is for triggering a pro-tell, prefixed as "info…"
- "*Print KB*" is for printing out the knowledge base in the output window.

6.2.2. Agent communication panel

The part at the bottom of the window provides interactions between different agents. *Receiver ID* identifies the receiver. *Message Type* identifies the type of a message you want to deliver to other agent. *Message Text* is the content of a message. *Start new plan* identifies the other plan you want to start at this point.

6.2.3. PrT net display panel

When the *Show Process Nets* button on the left side is clicked, all the plans of this agent will be shown in the form of the *Process Net*. The process net of a plan is a PrT net as Figure 4 shows. A PrT net consists of *Places*, *Transitions*, and *Arcs*. A *place* (in oval) indicates a possible system state. A place can contain tokens to indicate current state of

the system. A *transition* (in rectangle) is an action which changes the system state from one place to another. A *transition* is connected to a *place* by an *arc*. The arcs starting at a place and ending at a transition are called *input arcs*, and those starting at a transition and ending at a place are called *output arcs*.

The display of a Pr. T net uses five different colors to indicate different states:

- Grey: the grey rectangle separated with the process net at the top of the window indicates the local binding of the variables of the agent states. E.g. ?FIRE=fire2 indicates the FIRE variable is bound to fire2 at the point.
- Red: some places will be colorized as red when they are active.
- Green: the places that are not active.
- Yellow: the transitions in yellow means they have already been passed through before.
- Blue: the transition in blue means they haven't been passed through yet.

The three buttons listed below the process net window are *Fire a Transition*, *Toggle Layout*, and *Print Net*. Fire a Transition provides a convenient way to test how the process will be executed. That means the agent developers can check if the agent can make a correct decision by comparing with the desired choices and the actual choices. *Toggle Layout* button is used for automatically toggling the layout of the Pr. T. *Print Net* is for printing out the Pr. T.

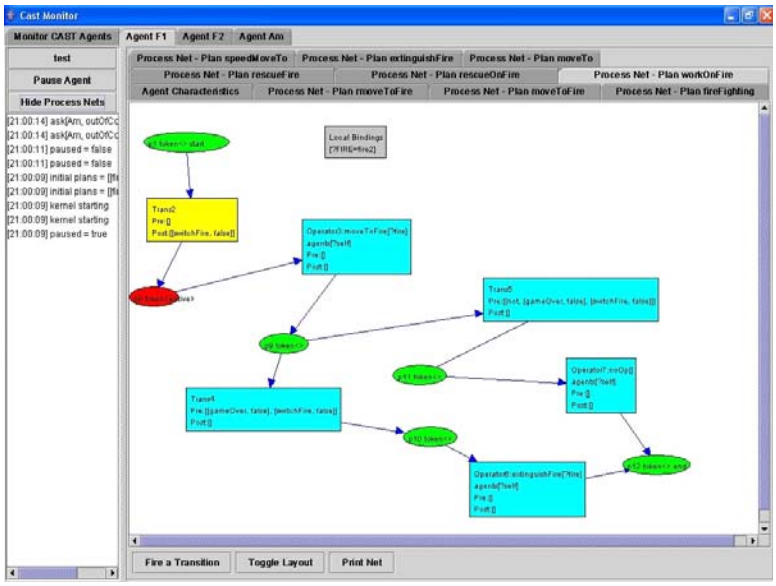

Figure 5: PrT net display.

7. How to program in CAST

7.1. CAST system overview

A team of CAST agents can interact with different types of domain or environments. First, CAST agents can be used as a model to simulate team behaviors. For example, CAST has been used to study how to enhance team performance by proactive information delivery. The experiments showed agents coordinate with different mental models on teamwork have different team performance. Second, CAST agents can be used as intelligent agents to model opponents in a computer game. For example, CAST has been used in dTank, a tank-game environment to compare the team performance of different agent architecture. Third, CAST agents can be used to simulator team-members in a training environment. CAST has been used in a simulated combat environment that designed for combat operational training. In the training, CAST agents are used to provide online feedbacks to trainees. Of course, CAST can be applied to other problem domains where teamwork or collaboration is required. In this section, we use dTank domain as an example to illustrate how to build an agent team with CAST.

Figure 6 shows the system architecture including domain, domain monitor, agents, and agent monitor. From the Figure, we can see it is a distributed system which is composed of two parts domain and agents. Agents interact with a domain: sense domain states and apply operation effects to the domain. Agents interact with each other by communication. A CAST monitor provides a display for users about the agents' knowledge, states, and operations. Normally, a domain monitor is linked to the domain or simulator to show the dynamics of domain states.

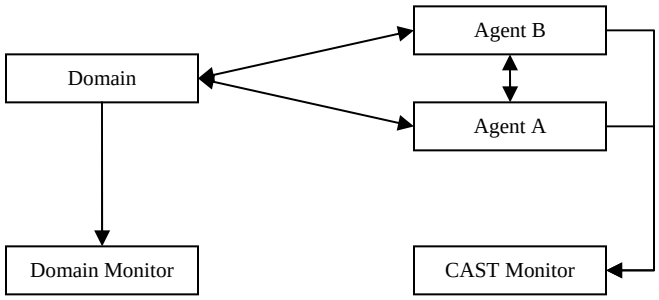

Figure 6: CAST system architecture.

Domain / Domain simulator

dTank (acs.ist.psu.edu/dTank) is a simulator created by the Applied Cognitive Science Lab (ACS) at the Pennsylvania State University. It provides an adversarial real-time test bed for conducting experiments with cognitive models in a tank combat setting. The simulator constructs a 10 by 10 grid-world composed of stone barriers and grassland. A simulated tank can move forward on grassland but cannot cross a stone or another tank. Tanks are able to navigate their environment by moving in the forward direction or turning either clockwise or counter clockwise. A tank can rotate its turret freely (360o) and the visual field is centered on the turret angle. It can observe world objects (other tanks or stones) if the objects are within its 100o visual field without being blocked by other game objects. A tank aims its gun at an enemy tank by turning its turret. Initially, a tank has a certain level of health, represented by health points, which decrees after the

tank is shot by others. When health points drop to zero, the tank dies. A tank can communicate with its teammates through a simulated radio function. A first-person driving view that can log behavior is also included, which will support human vs. human and human vs. agent games, as well as gathering data to model human users. Figure 7 shows a plan-view interface for monitoring a tank game. The dTank manual [6] provides more information.

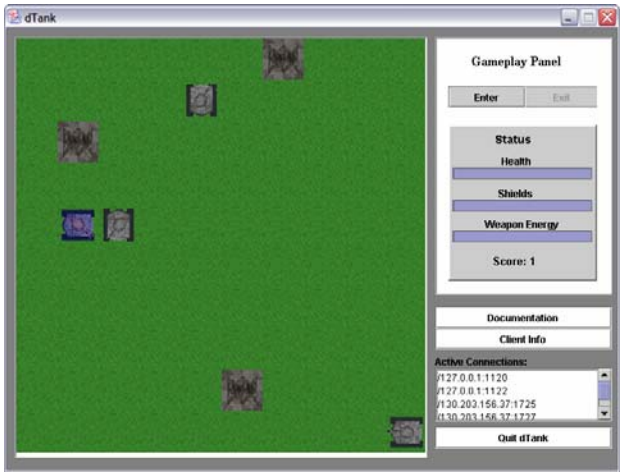

Figure 7: dTank plan-view monitoring interface.

Domain monitor

Domain monitor is normally provided for human to monitor the online domain situation. Therefore, the information coming to the domain monitor may not be appropriate to feed as the agent's sensory input. For example, one of the goals of dTank is to provide a testing environment in which agents and humans can interact on equal footing: both synthetic and human players should have similar sensory input from the game environment. If agents are able to view the entire game board along with the positions of all tanks while human can only see other tanks within their visual field, agents would have an unfair advantage over their human opponents.

CAST monitor

CAST monitor is useful to monitor the state of agents. It also provides log functions that can keep track of the agents' operational behavior. However, CAST monitor is not essential for applying the CAST agents in a domain.

CAST source code

MALLET2: includes the code for approaching MALLET files.
Domain: is the world simulator. The operators in the MALLET file should have been defined in domain.
Kernel2: the key part of CAST agent, which realized the main features of CAST, such as Dynamic Task Allocation, Detection of Information Needs of Teammates, Proactively Information Delivery.
Monitor: is to display the interface to the user for dynamically controlling the process of CAST agent.
Util: include some modules which are necessary for dealing with knowledge base.

7.2. Basic steps of building a CAST team

We adopt an agent oriented system analysis and design approach for building teams in CAST. The main tasks of creating teams with CAST include knowledge engineering for agents and facilitating functions that handle connections between agents and a domain.

1. **Study domain and get rules, APIs etc.**

The first step is to study the domain. A domain (simulator) defines the rules of the world such as the sensory (visual, audio inputs), operation effects (agents' outputs), and world state updates (time, resources). As an agent designer, you need to be clear about the environments which the agents will interact with.

Then, you need learn about the interaction protocols that are defined as API functions or network messages. For example the dTank defines its protocols as specified in table 3. By study the interactive protocols, you should identify how to handle agents' sensory inputs and apply agents' operators. In the dTank example, we may identify that agents can sense information about a tank including id, color, etc. We can also learn to apply operators such as move and turn.

Table 3: dTank communication protocol.

```
^io
        ^input-link
                ^dtank
                        ^id <<string>>
                        ^tank-color <<string>>
                        ^hit yes
                        ^died yes
                        ^status
                                ^health <<int 1-10>>
^output-link
                ^dtank
                        ^move
                                ^status complete
                        ^turn
                                ^direction clockwise | counter-clockwise
                                ^status complete
```

2. **Build domain adaptor**

According to the interaction protocol, you need build a domain adaptor that specifies basic communication with the domain. In the dTank example, the adaptor is composed of four classes:

- cast3.domain.JTankWorld.TankClient (defines the socket connection functions that facilitate the basic communication between the world simulator and agents.)

```
public TankClient(String host, int port) {
   try {
      this.host = host;
      this.port = port;
      tankSocket = new Socket(host, port);
      out = new PrintWriter(tankSocket.getOutputStream(), true);
      in = new BufferedReader(new
InputStreamReader(tankSocket.getInputStream()));
```

```
        } catch (UnknownHostException e) {
            System.err.println("Couldn't connect to host|port: " + host + "|" +
                               port + ".");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("Couldn't get I/O for the connection.");
            System.exit(1);
        }
        send("agent join");
        this.start();
    }
```

- cast3.domain.JTankWorld.AgentJTankWorldAdapter (defines how to apply the operators such as move and turn and how to handle the incoming messages)

```
    public void doMove() {
        operatorID++;
      //set command $base$dTank_delim$tail$dTank_delim$id
        sendWorldMessage("moveForward| |" + operatorID);
    }

    public void doTurn(String dir) {
        operatorID++;
      int direction = 1;
      if (dir.equalsIgnoreCase("left") )
          direction = 0;
        sendWorldMessage("rotate|" + direction + "|" + operatorID);
    }

    public void handleMessage(String message, String host, int port) {
      super.handleMessage(message, host, port);
      handleWorldMessage(message);
    }

    public void handleWorldMessage(String msg) {
        if (agent != null)
            agent.handleWorldMessage(msg);
    }

    public void sendWorldMessage(String msg) {
        send(msg);
    }
```

- cast3.domain.JTankWorld.JTankAgent (defines how to handle/parse the incoming messages)

```
public void handleWorldMessage(Object message) {
        String strMeg = trimBye((String)message);
        StringTokenizer strtok = new StringTokenizer(strMeg, stop);
```

```
    String type = new String();

    if (strtok.hasMoreTokens())
       type = strtok.nextToken();
    else
       System.out.println("error getting world message:" +
                          message.toString());

    if (type.equalsIgnoreCase("InitialSettings"))
       handleInitialMessage(strtok);
    else if (type.equalsIgnoreCase("ACTION"))
       handleActionMessage(strtok);
    else if (type.equalsIgnoreCase("ACK"))
       handleAckMessage(strtok);
    else if (type.equalsIgnoreCase("STATUS"))
       handleStatusMessage(strtok);
    else if (type.equalsIgnoreCase("INPUT"))
       handleInputMessage(strtok);
    else if (type.equalsIgnoreCase("SCAN"))
       handleScanMessage(strtok);
    else if (type.equalsIgnoreCase("EVENT"))
       handleEventMessage(strtok);
    else if (type.equalsIgnoreCase("VISUAL"))
       handleVisualMessage(strtok);
    else if (type.equalsIgnoreCase("STOPAGENT|"))
       setPaused(true);
    else if (type.equalsIgnoreCase("STARTAGENT|"))
       setPaused(false);
    else if (!type.startsWith("join"))
       handleUndefinedMessage(type, strtok);
  }
```

- cast3.domain.JTankWorld.JTankDomain (defines how to process the sensed information, and how to apply operators (the operators will be called by the agent execution module).)

```
  public void processSense() {
    if (!setTeam){
      jTankWorldAdapter.sendWorldMessage("setTeam|" + getTeam());
      setTeam = true;
    }
    updateTime();
    KBEnv solution = new KBEnv();
    String find = "((sense ?sense))";
    solution = getAgent().getKB().queryForBindings(find);

    while (solution != null) {
       Vector sense = (Vector)solution.lookup("?sense");
```

```
        getAgent().proactiveTell(sense);
        solution = getAgent().getKB().next();
    }

    ((JareKBWrapper)getAgent().getKB()).retract("(sense ?sense)");
}

/*
 * operators
 */
public void move() {
    jTankWorldAdapter.doMove();
}

public void attack() {
    jTankWorldAdapter.doAttack();
}
```

3. Get domain knowledge

An agent can be viewed as being composed of two parts: architecture and knowledge. The agent architecture is fixed, you should not change CAST architecture, unless it is necessary. Most of the agents' behavior should be captured as knowledge. We can group the knowledge part further into domain-dependent knowledge and domain-independent knowledge. The domain independent knowledge is the knowledge that is needed to compliment certain missing features from architecture. In the dTank team, for example, knowledge about how to aim a gun at an enemy is domain-dependent; knowledge on how to choose an indifference operator is domain-independent. We need to capture agent's knowledge before the agents know how to collaborate in a domain.

The procedural knowledge contains a two-phase plan: search and attack, as illustrated in Figure 8. Initially, a team of agents wander around and search for enemy tanks. Once an enemy is found, the team-members communicate to inform each other of the enemy location and to coordinate their attack. Next, team members attack the target together and destroy it. This process iterates until all the enemies are destroyed. Declarative domain knowledge includes moving directions, stone locations, and so on.

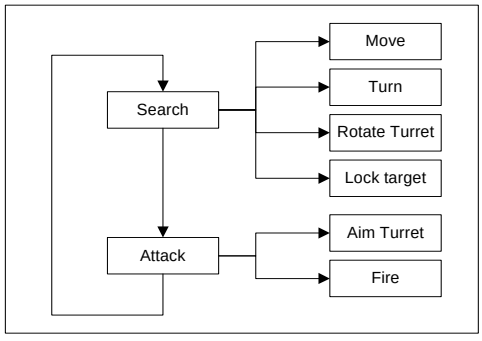

Figure 8: Agent plan decomposition.

4. Teamwork knowledge

In addition to domain knowledge, the designer also needs to capture teamwork knowledge, which defines the team composition, roles, and capabilities. For example, the following MALLET specifies a team for dTank domain.

```
(agent F1)
(agent F2)
 (team tankteam (F1 F2))

(capability F1      ((move) (turn) (attack) (rotateTurret) (raiseShields) (scan)))
(capability F2      ((move) (turn) (attack) (rotateTurret) (raiseShields) (scan)))

(plays-role F1      (fighter_tank))
(plays-role F2      (fighter_tank))

(role fighter_tank ((move) (turn) (attack)))
```

Moreover, the agent should also specify how the tasks should be allocated among the team members.

```
(plan attack_enemy ()
  (pre-cond  (at ?tank ?x ?y)(is enemy ?tank))
  (process
    (agent-bind
      (?f)
      (constraints (playsRole fighter_tank ?f))
    )
    …
  );endproc
);endplan
```

5. Starters

A CAST team is normally configured in an XML file, which defines the agent name, domain, MALLET plan, cast monitor, knowledge-base type, world address, port, and communication type.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CAST SYSTEM "jTank.dtd"><CAST>
<AGENT>
  <NAME>F1</NAME>
  <DOMAIN>cast3.domain.JTankWorld.JTankDomain</DOMAIN>
  <MALLET>teamplans/jTank/casttankteam.mlt</MALLET>
  <MONITOR>localhost</MONITOR>
  <KBTYPE>cast3.util.Jare.JareKBWrapper</KBTYPE>
  <WORLD_HOST>localhost</WORLD_HOST>
  <WORLD_HOST_PORT>3400</WORLD_HOST_PORT>
  <COMM_FEATURE>PROTELL</COMM_FEATURE>
```

```
</AGENT>
<AGENT>
 <NAME>F2</NAME>
 <DOMAIN>cast3.domain.JTankWorld.JTankDomain</DOMAIN>
 <MALLET>teamplans/jTank/casttankteam.mlt</MALLET>
 <MONITOR>localhost</MONITOR>
 <KBTYPE>cast3.util.Jare.JareKBWrapper</KBTYPE>
 <WORLD_HOST>localhost</WORLD_HOST>
 <WORLD_HOST_PORT>3400</WORLD_HOST_PORT>
 <COMM_FEATURE>PROTELL</COMM_FEATURE>
</AGENT>
</CAST>
```

An agent starter class (cast3.domain.JTankWorld.JTankAgentStarter) is responsible for instantiating the agents:

```
    while (it.hasNext()){
      Element agent = (Element)it.next();
      String name = agent.getChildText("NAME");
      String domain = agent.getChildText("DOMAIN");
      String mallet = agent.getChildText("MALLET");
      String monitor = agent.getChildText("MONITOR");

      domainhost=agent.getChildText("WORLD_HOST");
      domainhostport=agent.getChildText("WORLD_HOST_PORT");

      String feature=agent.getChildText("COMM_FEATURE");

      JTankAgent cast =
         new JTankAgent(name, mallet, domainhost, domainhostport);

    cast.setFeature(feature);

     AgentDisplay ap = new AgentDisplay(cast,"cast3.util.Jare.JareKBWrapper");
     manager.addAgent(name, ap);
    }
```

6. Test

Observing the agent's behavior, checking agent's knowledge base with query commands, and monitoring the state of the agent's process net are basic methods for testing and debugging. The followings are general guidelines for testing:

- Test the agent adaptor before testing the agent.
- Test the declarative knowledge (predicates) offline before testing the plans.
- Test plans with simple conditions before testing with complex conditions.
- Test single agent before testing the team.

8. Appendix A: the syntax of MALLET

CompilationUnit ::=(AgentDef | TeamDef | MemberOf | GoalDef | Start | CapabilityDef | RoleDef | PlaysRole | FulfilledBy | IOperDef | TOperDef | PlanDef | RuleDecl | LoadDecl)* <EOF>

PlanName ::=<IDENTIFIER>

OperName ::=<IDENTIFIER>

PlanOrOperName ::=<IDENTIFIER>

AgentName ::=<IDENTIFIER>

TeamName ::=<IDENTIFIER>

AgentOrTeamName::=<IDENTIFIER>

RoleName ::=<IDENTIFIER>

IdentifierListReq ::="(" (<IDENTIFIER>)+ ")"

VariableListOpt ::="(" (<VARIABLE>)* ")"

VariableListReq ::="(" (<VARIABLE>)+ ")"

MixedListOpt ::="(" (<IDENTIFIER> | <VARIABLE>)* ")"

MixedListReq ::="(" (<IDENTIFIER> | <VARIABLE>)+ ")"

Invocation ::="(" PlanOrOperName (<IDENTIFIER> | <VARIABLE>)* ")"

AgentDef ::="(" <AGENT> AgentName ")"

TeamDef ::="(" <TEAM> TeamName ("(" (AgentName)+ ")")? ")"

MemberOf ::="(" <MEMBEROF> AgentName (TeamName | "(" (TeamName)+ ")") ")"

Pred ::="(" (<IDENTIFIER> | <NOT> | <EQUATION> | <LT> | <GT> | <LE> | <GE>) (<IDENTIFIER> | <VARIABLE> | Pred)* ")"

Cond ::=Pred

| "(" <NOT> Cond ")"

AssertDef ::="(" <ASSERT> (Pred)+ ")"

RuleDecl ::="(" <RULE> (Pred)+ ")"

LoadDecl ::="(" <LOAD> <IDENTIFIER> ")"

RetractDef ::="(" <RETRACT> (Pred)+ ")"

GoalDef ::="(" <GOAL> AgentOrTeamName (Cond)+ ")"

Start ::="(" <START> AgentOrTeamName Invocation ")"

CapabilityDef ::="(" <CAPABILITY> (AgentName | "(" (AgentName)+ ")") (Invocation | "(" (Invocation)+ ")") ")"

RoleDef ::="(" <ROLE> RoleName (Invocation | "(" (Invocation)+ ")") ")"

PlaysRole ::="(" <PLAYSROLE> AgentName (RoleName | "(" (RoleName)+ ")") ")"

FulfilledBy ::="(" <FULFILLEDBY> RoleName (AgentName | "(" (AgentName)+ ")")

")"

PreConditionList ::="(" <PRECOND> (Cond)+ ((":IF-FALSE" | ":if-false") (<FAIL> | <WAIT> ((<IDENTIFIER>)+)? | <ACHIEVE>))? ")"

EffectsList ::="(" <EFFECTS> (Cond)+ ")"

TermConditionsList::="(" <TERMCOND> (<SUCCESS> | <FAILURE>)? (Cond)+ ")"

NumSpec ::="(" <NUM> (<EQ> | <LT> | <GT> | <LE> | <GE>) (<IDENTIFIER>)+ ")"

IOperDef ::="(" <IOPER> OperName VariableListOpt (PreConditionList)* (EffectsList)? ")"

TOperDef ::="(" <TOPER> OperName VariableListOpt (PreConditionList)* (EffectsList)? (NumSpec)? ")"

PlanDef ::="(" <PLAN> PlanName VariableListOpt (PreConditionList | EffectsList | TermConditionsList)* "(" <PROCESS> MalletProcess ")" ")"

ByWhomSpec ::=AgentOrTeamName

| MixedListReq

| <VARIABLE>

MalletProcess ::="(" <SEQ> (MalletProcess)+ ")"

| "(" <PAR> (MalletProcess)+ ")"

| "(" <IF> "(" <COND> (Cond)+ ")" MalletProcess (MalletProcess)? ")"

| "(" <WHILE> "(" <COND> (Cond)+ ")" MalletProcess ")"

| "(" <FOREACH> "(" <COND> (Cond)+ ")" MalletProcess ")"

| "(" <FORALL> "(" <COND> (Cond)+ ")" MalletProcess ")"

| "(" <CHOICE> (MalletProcess)+ ")"

| "(" <JOINTDO> (<AND> | <OR> | <XOR>)? ("(" ByWhomSpec MalletProcess ")")+ ")"

| "(" PlanOrOperName (<IDENTIFIER> | <VARIABLE>)* ")"

| "(" <DO> ByWhomSpec MalletProcess ")"

| "(" <AGENTBIND> VariableListReq "(" <CONSTRAINTS> (Cond)+ ")" ")"

| (AssertDef)+

| (RetractDef)+

9. References

[1] J. Yen, X. Fan, S. Sun, R. Wang, C. Chen, K. Kamali, M. Milller, and R. A. Volz, "On Modeling and Simulating Agent Teamwork in CAST," presented at 2nd International Conference on Active Media Technology,, Chongqing, P. R. China, 2003.

[2] D. Xu, R. A. Volz, T. R. Ioerger, and J. Yen, "Modeling and Analyzing Multi-Agent Behaviors Using Predicate/Transition Nets," *International Journal of Software Engineering and Knowledge Engineering*, vol. 13, pp. 103-124, 2003.

[3] J. Yen, X. Fan, and R. A. Volz, "On Proactive Delivery of Needed Information to Teammates," presented at AAMAS 2002 Workshop of Teamwork and Coalition Formation, Italy, 2002.

[4] J. Yen, X. Fan, and R. A. Volz, "Proactive Information Exchanges Based on the Awareness of Teammates' Information Needs," presented at AAMAS 2003 Workshop on Agent Communication Languages and Communication Policies, Melbourne, Australia, 2003.

[5] T. Finin, R. Eritzson, D. McKay, and R. McEntire, "KQML as an agent communication language," presented at Third International Conference on Information and Knowledge Management (CIKM'94), 1994.

[6] I. G. Councill, G. P. Morgan, and F. E. Ritter, "dTank: A Competitive Enviorment for Distributed Agents," The Pennsylvania State University, University Park, Technical Report ACS2004-1, 03/30/2004 2004.