A FRAMEWORK FOR KNOWLEDGE-BASED TEAM TRAINING

A Dissertation

by

MICHAEL SCOTT MILLER

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

DOCTOR OF PHILOSOPHY

August 2006

Major Subject: Computer Science

A FRAMEWORK FOR KNOWLEDGE-BASED TEAM TRAINING

A Dissertation

by

MICHAEL SCOTT MILLER

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

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Richard A. Volz
Committee Members,	John Yen
	Thomas R. Ioerger
	Winfred Arthur Jr.
Head of Department,	Valerie E. Taylor

August 2006

Major Subject: Computer Science

ABSTRACT

A Framework for Knowledge-Based Team Training. (August 2006)

Michael Scott Miller, B.S., Texas A&M University;

M.S., University of Houston-Clear Lake

Chair of Advisory Committee: Dr. Richard A. Volz

Teamwork is crucial to many disciplines, from activities such as organized sports to economic and military organizations. Team training is difficult and as yet there are few automated tools to assist in the training task. As with the training of individuals, effective training depends upon practice and proper training protocols.

In this research, we defined a team training framework for constructing team training systems in domains involving command and control teams. This team training framework provides an underlying model of teamwork and programming interfaces to provide services that ease the construction of team training systems. Also, the framework enables experimentation with training protocols and coaching to be conducted more readily, as team training systems incorporating new protocols or coaching capabilities can be more easily built.

For this framework (called CAST-ITT) we developed an underlying intelligent agent architecture known as CAST (Collaborative Agents Simulating Teamwork). CAST provides the underlying model of teamwork and agents to simulate virtual team members. CAST-ITT (Intelligent Team Trainer) uses CAST to also monitor trainees, and support performance assessment and coaching for the purposes of evaluating the

performance of a trainee as a member of a team. CAST includes a language for describing teamwork called MALLET (Multi-Agent Logic Language for Encoding Teamwork). MALLET allows us to codify the behaviors of team members (both as virtual agents and as trainees) for use by CAST.

In demonstrating CAST-ITT through an implemented team training system called TWP-DDD we have shown that a team training system can be built that uses the framework (CAST-ITT) and has good performance and can be used for achieving real world training objectives.

DEDICATION

To my parents, Joe and Jeanette, who have supported me in all of my endeavors.

ACKNOWLEDGMENTS

I would like to thank first my advisor, Richard A. Volz. I would also like to thank my committee members, John Yen, Thomas Ioerger, and Winfred Arthur, Jr.

I would also like to thank the following individuals as part of the MURI project, Wayne Shebilski, Shruti Narakesari, Ganesh Alakke, Jesse Plymale, Bhargavi Srivathsan, Kevin Walkington, and Shuang Sun. I have had a great time interacting with other graduate students, in particular, John Surdu, James Vaglia, Jeff Bourne, Greg Schmidt, Sen Cao, Yu Zhang, Jianwen Yin, Joe Sims, Ryan Rozich, and Cong Chen.

This research was funded by DOD MURI grant F49620-00-1-0326 administered through AFOSR.

TABLE OF CONTENTS

Page

ABSTRACT iii

DEDICATION v

ACKNOWLEDGMENTS vi

TABLE OF CONTENTS vii

LIST OF TABLES ix

LIST OF FIGURES x

1. INTRODUCTION 1

1.1 Underlying Assumptions 4
1.2 Goals of the Research 5
1.3 Accomplishments of This Research 7
1.4 Overview of Dissertation 9

2. LITERATURE REVIEW 11

2.1 Teamwork 11
2.2 Training Simulations 21
2.3 Intelligent Tutoring Systems 24
2.4 Modeling Human Behavior 33
2.5 Agent-based Teamwork 40
2.6 Agents in Intelligent Tutoring Systems 46

3. ISSUES IN DESIGNING A TEAM TRAINING FRAMEWORK 50

3.1 Integrating a Simulation Domain 54
3.2 Communications for Teamwork 59
3.3 Virtual Team Members 67
3.4 Monitoring Trainees 73
3.5 Performance Assessment 78
3.6 Coaching in Support of Teamwork 82

4. COLLABORATVE AGENTS SIMULATING TEAMWORK 87

4.1 Multi-Agent Logic Language for Encoding Teamwork 89
4.2 CAST Architecture 96

Page

4.3 Summary 109

5. TEAM TRAINING FRAMEWORK 110

5.1 Integrating a Specific Domain into CAST-ITT 117
5.2 Communications for Teamwork 131
5.3 Integrating a Virtual Team Member with a Simulation Domain 159
5.4 Monitoring Trainees 175
5.5 Performance Assessment 195
5.6 Coaching in Support of Teamwork 214

6. DOMAIN: DISTRIBUTED DYNAMIC DECISION MAKING 233

6.1 DDD Performance Support 236
6.2 CAST and DDD 240
6.3 MALLET and DDD 245
6.4 Agent-Human Communications in DDD 247
6.5 Summary 248

7. VALIDATION OF THE FRAMEWORK 250

7.1 Validating the Framework in a Team Training System: TWP-DDD 251
7.2 Validating Additional Features of the Framework 260
7.3 Domain Specific Assessment and Coaching for DDD 272

8. FUTURE WORK AND CONCLUSIONS 274

8.1 Significance of Research 275
8.2 Future Work 277
8.3 Conclusions 280

REFERENCES 282

APPENDIX A 290

APPENDIX B 292

APPENDIX C 295

VITA 296

LIST OF TABLES

Page

Table 1: DM assets 235
Table 2: Attack assignments with trainee as DM1 268
Table 3: Attack assignments with trainee as DM4 269

LIST OF FIGURES

Page

Figure 1: ITS with student model in operation 26
Figure 2: Beliefs, goals, and intentions 35
Figure 3: Petri net in action 38
Figure 4: Logical view of intelligent agent 71
Figure 5: CAST agent packages and classes 98
Figure 6: ProactiveTell and ActiveAsk algorithms 102
Figure 7: Example predicate transition net 104
Figure 8: CAST logger 108
Figure 9: CAST-ITT framework 112
Figure 10: CAST-ITT packages 116
Figure 11: Integrating a specific domain 120
Figure 12: ActorDomain abstract class 121
Figure 13: Flow of execution 123
Figure 14: Integrating domain sensing into CAST-ITT 127
Figure 15: Send/receive methods 134
Figure 16: KQML syntax 139
Figure 17: KQML objects 140
Figure 18: TraineeMessages interface 143
Figure 19: Second form of human to agent communications 149
Figure 20: First form of human to agent communications 150
Figure 21: Flow of communications 156
Figure 22: Observation-based communications 158
Figure 23: Virtual team member in CAST-ITT 160
Figure 24: Synchronizing agent to world 161
Figure 25: Template for MALLET domain plans for virtual team member 171
Figure 26: Skeleton executor and dispatcher plans 173

Page

Figure 27: Monitoring agent in CAST-ITT 176
Figure 28: Recording a monitored domain command 180
Figure 29: MonitorModule abstract class 192
Figure 30: Assessment support in CAST-ITT 197
Figure 31: Logical view of individual and team assessment support 199
Figure 32: TraineeAssessment and event classes 201
Figure 33: PerformanceModule abstract class 205
Figure 34: PerformanceResult abstract class 206
Figure 35: TeamModule abstract class 209
Figure 36: Generic support modules 210
Figure 37: Trainee visual action trace 212
Figure 38: TeamMetrics class 213
Figure 39: Training session timeline 215
Figure 40: Coaching agent 219
Figure 41: TeamModule abstract class 221
Figure 42: In-session feedback support 224
Figure 43: Direct feedback example 226
Figure 44: Indirect feedback example 227
Figure 45: After action review display 231
Figure 46: Decision maker screen in DDD 234
Figure 47: Intel report and planning tool 237
Figure 48: Task assignment panel and DDD 239
Figure 49: TWP-DDD integration 241
Figure 50: DDD with CAST-ITT 244
Figure 51: MALLET plans in DDD 246
Figure 52: Performance of partners versus no partner 255
Figure 53: Lead DMs in session 262
Figure 54: Modified operators in DDD 264

Page

Figure 55: Starting a plan as a team 264
Figure 56: Sample monitor agent log 270
Figure 57: Generic coaching agent 271

1. INTRODUCTION

Teamwork is required for many disciplines, from activities such as organized sports to economic and military organizations. Almost all human activities involve cooperation and information exchange to one degree or another. However teams go beyond superficial cooperation between third parties to a notion of common goals shared by members of a group. A group becomes a team when the cooperation required is essential to achieving shared goals (Morgan Jr. et al., 1986). This notion of shared goals provides a definition of a team as a group of entities (humans or agents) that are working together to achieve a goal that could not be accomplished as effectively (or at all) by any one of them alone.

Teams can be heterogeneous or homogenous in their composition. Some teams may have members that play unique roles, which require unique skills and resources, whereas other teams may have members who share roles and responsibilities. In terms of leadership, some teams are hierarchical, with a clear chain-of-command and leadership or authority roles; others are more flat and democratic. Yet, in all cases, all teams share a sense of unity among the team members even if those team members have other individual goals. Team unity is expressed as shared team goals that may evolve as the execution of the team progresses.

This dissertation follows the style of the *Journal of Artificial Intelligence Research*.

To achieve those shared team goals, teams have to work together by sharing information and cooperating in distributed decision making (also called collaboration). Expert teams have realized that to improve their proficiencies as a team they should follow the maxim "practice makes perfect." Therefore, not only do expert team members work together, but they also train together. In team training, the focus is not so much on each individual's task skills (which may typically be learned offline), but on improving interactions such as situational awareness, communications efficiency, and the effectiveness of group decision making skills (McIntyre & Salas, 1995).

Training teams has an additional complexity over the training of individuals. By definition a team requires multiple participants and each of those participants require resources. Each solution to team training has been a unique solution involving great cost in resources and training manpower. As one example, the US Navy has devoted considerable time and effort to understanding team failures, such as the Vincennes incident, and in developing protocols for training teams in its warship's Combat Information Center (CIC). Developing expert teams able to function under such stress is difficult (Cannon-Bowers & Salas, 1998). And developing and maintaining such expert skills sets are an ongoing challenge.

As with the training of individuals, effective training depends upon practice and proper training protocols. With the cost and risk of using the real world environment there has been a rise in the use of simulation for training. Simulation also allows the trainer to introduce a trainee to scenarios that entail to great a risk in the real world. However, determining what fidelity and capabilities should be provided by these

simulations is difficult. Even simulations have high costs because of their own development costs. Furthermore the expense of team training exercises and a limited experience with developing good team training protocols slows the development of such simulations with effective training protocols. Experimentation with protocols entails its own risk and, is limited to what experts believe works in order to maximize the benefit of the training.

In some high-risk domains, teams may practice their teamwork and individual skills more often than they actually perform those skills as they go about their day-to-day routine. The NASA Mission Control Center at Johnson Space Center only flies about four Space Shuttle missions in a year, with each mission lasting from 10 to 14 days. Therefore, the flight controller teams and space shuttle teams instead spend the majority of their time in front of workstation consoles and simulators training for future missions (NASA, 1998).

However, even normal operations may not fully exercise teams, as teams are not exposed to all situations that can potentially occur. A navigational team on a ship may enter and exit harbors many times without ever suffering the consequences of a loss of power and resulting need for a quick and correct response during those harbor operations (Hutchins, 1995). The cost of failure can become very high for both humans and material, therefore teams in high-risk domains practice in order to be able to learn the interactions of the team members and devise strategies to anticipate and handle emergencies.

In this dissertation, we look at defining a team training framework for constructing team training systems in domains that may be at high risk for the consequences of failure and that require high levels of skill in order to function. A team training framework will make the construction of training systems for high-risk environments easier and less expensive. Also, it will enable experimentation with training protocols and coaching to be conducted more readily, as systems incorporating new protocols or coaching capabilities can be more easily built.

1.1 Underlying Assumptions

In laying out the foundation for a team training framework we had four underlying assumptions for both the framework and the domains in which it would operate.

The first assumption is the characteristics of the teams being trained. These teams are the command and control (C2) team. This type of team is common in organizations that need to manage assets and resources within a restricted time and space, which involves coordination and communication between both the team and sometimes the assets, and the management of resources. A few examples of C2 teams are the Mission Control Center at NASA, Air Traffic Control centers, warships and their attendant operations centers, and the command staffs of military units.

The second assumption is the training environment. The expectation is that simulation technology will be available to model the command and control domain for training purposes. Having such technologies allows us to provide a software-based framework to support the development of training systems in regards to interacting with the training domain. However, there will not be a specific simulation system that applies

to all domains; therefore, a team training framework must be adaptable to multiple simulation systems.

The third assumption is that a model of teamwork will be incorporated into a team training framework. We have developed a model which includes a team-description language called MALLET (Multi-Agent Logic Language for Encoding Teamwork). This teamwork model allows us to facilitate the development of team training systems by providing a basic set of team oriented capabilities within the framework. The model is based on proactive information exchange (Yen et al., 2001). It includes an explicit representation of individual and team goals which can be useful for diagnosing problems with team interactions and supporting coaching. This language is parsed and executed using an associated intelligent agent architecture. We call this agent architecture, CAST, for Collaborative Agent architecture for Simulating Teamwork (Miller et al., 2000).

Our fourth assumption was that we could extend CAST to support team training with human trainees (Miller et al., 2000). The first version of CAST worked entirely with all agent teams. CAST became the underlying implementation for the model of teamwork and the support of virtual team members for our team training framework.

1.2 Goals of the Research

The use of intelligent tutoring systems for team training is still in its infancy, and still requires significant experimentation with both different forms of the use of agents and with different team training protocols. However, until now, each new experiment of this nature has required the design and construction of a great deal of specialized software,

and this is generally very expensive. The goal of this research is to create a framework within which a variety of training systems for different training objectives using intelligent agents in a variety of ways can be easily built (Miller et al., 2000).

We believe we can specify a framework that incorporates a unified view of the needs and functionality of a team oriented training system (each for a specific training domain) thus allowing a coherent approach to the specification of interfaces and mechanisms as needed to realize the implementation of such team training systems. Such an approach (framework) allows us to build a variety of systems more economically to test team training ideas, both through the use of intelligent agents and novel training protocols.

Specific objectives include:

- Easy incorporation of different simulation domains
- Use of intelligent agents as virtual team members interoperating with human trainees
- Support for adding domain specific assessment and evaluation
- Support for adding domain specific coaching via either humans or intelligent agents for both on-line and post training session
- Well defined interfaces that facilitate the use of a team training framework for including the above capabilities into a specialized team training system.

1.3 Accomplishments of This Research

The team domain in this research is based on C2 teams that are arranged in a hierarchical manner with clearly delineated roles for each team member. A team is modeled by representing each team member individually and not as an aggregate. The model of teamwork used should represent the shared responsibilities and goals that the individuals on the team have.

The model of teamwork within the team is maintained by the use of the supporting teamwork-oriented intelligent agent architecture. As part of this model a team training framework should include a language for describing team behaviors and information needs. During the course of training, intelligent agents within a team training framework use expert domain specific team plans and the goals of the chosen training scenario, expressed in this language, to simulate the activities of the virtual team members. The ability to replace those team members that are not human trainees by intelligent agents is a key idea in this research. At the same time, the team model within each agent drives team interactions between the agent and the trainees acting as other team members.

Provisions for a coaching agent are made within such a team training framework. These provisions support a generic coaching agent and the interfaces to add domain specific coaching solutions. Additionally, the framework provides access to the teamwork model to assist in evaluating the performance of a human trainee working as a team member in the team. The framework is able to support the construction of an overall model of the performance of the entire team and includes the capability of

supporting the development of multiple approaches to providing performance support to a trainee.

A team training framework should provide interfaces[1] that are designed in a manner that is easy to understand and extend over multiple training domains. The framework also provides specific interfaces to connect the intelligent agents to simulation domains. By connect we mean the agents are able to act and sense in order to execute their planned behaviors. The framework provides interfaces and dedicated agents to monitor human actions within the domain. This monitored data can in turn be accessed for assessment and coaching purposes. Although a generic level of assessment is provided based on the teamwork model used by the framework, the framework also provides interfaces that support the plug-in of domain specific assessment and coaching modules to achieve domain specific training objectives.

The novel contributions of this research are: 1), easy to use mechanisms for the integration of a simulation domain to a team training framework for interaction (sensing and acting) and monitoring purposes, 2) flexible generic mechanisms upon which a large variety of human/agent communication modes can be incorporated into a team training system, 3) easy to use mechanisms for development of both generic and domain specific assessment, 4) generic support for coaching, and 5) an approach for designing and executing team training systems. Underlying all of this is the use of a model of teamwork in a generic manner to support both the use of agents as virtual team members and as monitoring agents. In addition, significant contributions were made to the

[1] We mean interface in the broader sense as opposed to software such as the Java Interface.

teamwork language MALLET and the CAST supporting agent architecture for simulating virtual team members

1.4 Overview of Dissertation

This section provides an introduction to this research as well as the motivation and the ideas central to this approach.

Section 2 contains a review of the current literature relevant to this research. Areas of interest to this research are in the nature of teamwork in command and control teams, the use of intelligent training systems, and the intelligent agent architectures that focus on representing human teamwork.

Section 3 describes the issues for designing a team training framework. These issues are the integration of a team training system to a simulation domain, handling of team communications, defining virtual team members, monitoring trainees, support for assessment of individual trainees, and support for coaching of trainees in the context of a team.

Section 4 gives the details of the existing implementation of CAST. In this research, the underlying intelligent agent architecture is known as CAST (Collaborative Agents Simulating Teamwork) (Yen et al., 2001). CAST includes a language for describing teamwork called MALLET (Multi-Agent Logic Language for Encoding Teamwork) (Fan et al., 2006). Part of the development of CAST has been in support of this research (Miller et al., 2000). This discussion is divided into overviews of the team language, MALLET, and the agent architecture, CAST.

Section 5 describes the approach for the proposed team training framework. Section 5 also answers the issues introduced in Section 3 in specifying the interfaces of a team training framework.

Section 6 discusses the sample domain implementation used in this research. The sample simulation domain is known as the Distributed Dynamic Decision making (DDD) system.

Section 7 describes the validation of the proposed team training framework. Validation was done through both a real world training environment and a set of validation tests of the interfaces of the framework.

Section 8 covers the conclusions and significance of the current research. Also discussed are future directions for this research.

2. LITERATURE REVIEW

In this section, we will cover the following research topics as they pertain to this dissertation:

- The nature of human teamwork within command and control based teams
- Simulation domain systems for training
- Intelligent Tutoring Systems
- Knowledge-based agents for modeling human behavior
- Models of agent-based teamwork
- A review of specific Intelligent Tutoring Systems related to this research

2.1 Teamwork

In order to quantify the learning that an individual will gain in an intelligent team training system the first step is identify the types of teamwork processes to be learned. Teamwork can be divided into two views. The first view is the outcome of the work of a team (e.g. the Mission Control Center has a successful shuttle mission). The second view is of the team process of the team (e.g. the actions, coordination, and sequencing of activities of individual team member in the context of performing a successful shuttle mission).

In this dissertation we focus on the second view which is the view used by industrial/organizational psychologists in looking at group or team behaviors and interactions. However, we will capture the outcome of the team's progress in achieving its goals. Outcome based measures of team success are an essential part of this

dissertation in order to evaluate the final success of the team while also capturing the performance of the teamwork within the team.

2.1.1 Command and Control

The nature of the Command and Control (C2) teams that we are interested in supporting in this research has been studied in the military command and control literature (Builder et al., 1999). The C2 team has two functions: command and control. Command is the authority vested in the individuals of a team for the goals and responsibilities of the team. With its own authority the C2 team has limited need to depend on external higher authorities in order to carry out its own primary mission. Control is the coordination and arrangement of the use of needed resources to implement the desires as expressed by the command. With the required resources and ability to access the state of the team and its environment, a team should be able to conduct its operational role without calling on external resources. For training, the above definition means that with both of these functions together and a necessary simulation of the environment we can train the team in its mission without other external influences.

Lawson views command and control as a process of perceiving changes to the state of the environment, identifying the desired state of the environment, and taking actions to ensure that desired state is reached (Lawson, 1981). A primary consideration for Lawson is that a C2 system operates within a hostile environment that the C2 system must respond to and shape to its will. This desirable outcome is frequently mentioned by battlefield commanders as shaping the battlefield and acting within the decision cycle of their opponents (TRADOC, 2001).

The TADMUS study (Cannon-Bowers & Salas, 1998) gives an overview of US Navy command and control teams. The study was an outgrowth of the inquiry into the USS Vincennes shooting down Iran Air Flight 655 on July 3, 1988. In the Vincennes example, a combination of stress and a lack of training in areas such as cross skills, coordination, and maintaining situational awareness contributed to the factors that led to a disaster in how the team of the USS Vincennes responded to threats to the ship.

Command and Control teams have a unique combination of characteristics focused on managing limited resources in response to a changing environment. These characteristics are as follows (Pew & Mavor, 1998):

- A constantly changing and incomplete view by the team members of the current situation and the overall environment
- C2 teams follow what is called the Naturalistic Decision Making process for rapidly making decisions given limited time and options
- Each member of the team has a strict role and list of responsibilities.
- There is a hierarchical chain of command

Command and control teams reviewed in the literature consist of two broad categories. The first are heterogeneous teams in which each member is dedicated to fulfilling a unique task. One example of heterogeneous teams is the U.S Army battalion staff. In a U.S. Army battalion staff, each of the team members is given a designation (i.e. S-1, S-2, S-3, S-4, and S-5). The designations break down into functional areas within the team. They cover operations, intelligence, planning, supply, and personnel functions that need to be managed within the battalion (TRADOC, 1997).

The second type of command and control teams is homogenous in their roles. A homogeneous team example is in air traffic control teams where each controller has the same task of controlling flight operations. While their roles are the same, either an individual controller will be given geographical zones of control or some other system of dividing the airspace will be used in order to manage the workload (Nolan, 2003). A two man aircraft is similar in that while both pilots are capable of operating the aircraft, the more senior pilot commands the aircraft and the subordinate pilot backs up the senior pilot.

All C2 teams share the common characteristic that training is difficult (Pew & Mavor, 1998). Many C2 teams require extensive training on the job in order to expose a trainee to the complex situations and interactions required to become proficient. For those situations that do not occur routinely, training C2 teams requires dedicated resources and time allocated for all team members to build appropriate responses to emergencies (Cannon-Bowers & Salas, 1998). This subject will be covered in more depth in Section 2.1.3.

2.1.2 Naturalistic Decision Making

Naturalistic Decision Making (NDM) is a theory of how people use their experience to make decisions in a changing environment (Zsambok, 1997). The context in which the decision making happens is the key feature. It has been proposed as a model of how people behave in command and control teams when team members have to make multiple decisions in the face of an ever-changing environment.

The idea behind NDM is that as information is perceived, humans engage in choosing a course of action that corresponds to the known information taking into account prior experience. A NDM model for looking at military decision making differs from traditional utility-based models by changing the focus from the worth of an action to the context of an action. The context of the action has two steps. The first step is situation assessment and the second step is action selection. Situation assessment is information collection and deciding what is relevant and what is not. Action selection is done based on choosing the appropriate course of action to handle the situation as assessed. Choosing the appropriate course of action to take is based on the prior experience and training of the individual.

One model of NDM is the Recognition Primed Decision (RPD) model (Klein et al., 1986) which focuses on the recognition of the situation. The RPD model states that decision makers draw upon their previous experience to identify a situation as belonging to a particular class of problems. The decision maker is then able to select an appropriate course of action (COA), either based on similar prior situations, or by adapting previous approaches. The decision maker then evaluates the selected COA and adapts it or a suitable modified variant to handle the situation.

The Recognition Primed Decision (RPD) model does have a number of limiting factors. Experts use experience to see what novices do not; therefore RPD does not help in stating how novices would react to similar situations. Situations unknown to experts require a different decision making process. In such situations, the decision maker must fallback to other approaches such as drawing on analogies or through creative processes.

NDM extends beyond simple decision making about a single event and what action to take towards involving the complexity of the real world decision making as done by humans. Therefore, NDM is of interest to psychologists studying real human teams. Eight factors of NDM are as follows (Orasanu & Connolly, 1993):

1. Ill-structured problems
2. Dynamic environment
3. Competing goals
4. Multiple feedback loops
5. Time constraints
6. High stakes
7. Multiple players
8. Organizational norms and goals versus the individual's beliefs

The combination of the above factors poses complications for research into NDM. Ill-structured problems, dynamic environments, competing goals, multiple feedback loops all produce worlds of uncertainty and incomplete knowledge. Those worlds complicate the development of simulation models and assessment tools until those worlds are inspected in-depth. The last four factors add pressure to perform correctly and in conjunction with other team members and both internal and external pressures. Taken together, the above eight factors provide a challenging area of study that has only recently been attempted.

Given these eight factors, people wish to develop teams that are competent by using NDM as a guideline to predicting their behavior. To build expert teams using the

NDM paradigm the following needs to be done, according to the following guidelines (Zsambok, 1997):

1. Foster shared mental models
2. Train on teamwork skills involving communications, roles, and responsibilities
3. Allow teams to practice on guided scenarios
4. Allow practice on a selection of courses of actions
5. Train team members about other's roles and requirements
6. Train team leaders to manage tasks, delegate responsibilities, and maintain awareness of the team actions.

Based on the above needs, the US Navy and other government agencies have turned to simulator-based training technologies. Simulator technologies place individuals in the required scenarios or situations in order to gain experience in stressful situations and save training costs and dangers by reducing the need for on the job training.

In the next section we switch the focus from defining teams to how teams are trained.

2.1.3 Training Teams

Training teams involves teaching individual team members how best to function as part of a team. The performance of the team is based on both the expertise of the individual in their taskwork and in their ability to work with other team members towards shared goals. Therefore a team trainer must not only be able to model the performance of the individual trainee in respect to their taskwork but also their communication and

coordination skills as a member of a team. In this section we discuss what is required in modeling both the individual and team performance of a trainee.

We can differentiate the domain specific taskwork from the teamwork required to coordinate that taskwork among team members (Morgan Jr. & Bowers, 1995). Furthermore, such teamwork coordination skills can be carried across to other team level tasks outside of the first teamwork experiences of a trainee (McIntyre & Salas, 1995). However, finding an expert level of task proficiency requires access to expert teams. Therefore, research into teams tends to fall into two groups. The first research group is limited to reviews of the behaviors and identified characteristics of expert teams. The second research group is based on experiments using novices for which a reasonable proficiency is achieved by those novices during the course of the research. The separation between these two types of research allow for gaps in our current understanding of the relationship between teamwork and taskwork.

One attempt to close this gap is to develop performance metrics for teamwork. An important goal of training individuals as part of team is to devise how to rate an individual's teamwork skills as part of the overall team performance as how to evaluate the performance of the system as affected by the team members. The ATOM (Anti-Air Teamwork Observation Measure) studies developed a conceptual framework of assessing team performance. The studies linked the task outcome to the team process to provide a basis for performance. As part of the studies, the ATOM studies developed the following teamwork measures (Johnston et al., 1997):

1. Situation Assessment

2. Communication
3. Team initiative/leadership
4. Helping behavior

The four areas were used to develop a methodology for debriefing the teams after training exercises. The methodology for debriefing consisted of self critique and team member interviews in order to rate their performance in each of the four areas on a scale of 1 to 5. The guided aspect of the debriefing focused critical evaluation on the effectiveness of the team's teamwork as opposed to individual task performance.

TADMUS (Tactical Decision Making Under Stress) defines key factors to training teamwork skills (Cannon-Bowers & Salas, 1998). The approach of the program was to build a body of knowledge on decision making issues in high stress teamwork environments. The program also desired to develop measures of teamwork performance and principles for decision support and management of teams. TADMUS lists the following key factors for training.

- Training is scenario driven.
- Training focuses on teamwork skills.
- Training works on the responsibilities and monitoring of the leaders.
- Cross training in the roles of other team members is used to promote the development of the shared mental models.
- Specific training is done on the knowledge needs, roles, and responsibilities of team members.
- Enable self-correction within teams.

- Separate task and team training in the learning objectives.
- Train the trainees to know what is that is being taught and how to learn better.

The factors listed for TADMUS were used to derive several key principles for training teams.

First, automaticity is the idea that as skills are learned and practiced they become part of the unconscious operations of an individual (Shebilske et al., 1999). As the domain skills and behaviors become automatic, the expert is able to spend less time reasoning about the low level operations of a task and instead reason about issues that are more complex and therefore be able to better react to emergencies and high-stress situations then a beginner.

Second, pattern recognition is a key part of TADMUS in describing how command and control teams respond to situations. Such teams look for and recognize patterns in the environment that lead the team to react in the appropriate manner. In the case of the USS *Vincennes,* several key misidentifications of features were made by the crew leading to the mistaken assumption that an Iranian airliner was a military jet with hostile intentions and as the ship was operating in a hostile environment the crew reacted in a lethal and flawed manner.

Last, team-coordination and communication skills are essential in any team but gain a greater importance in a team comprised of heterogeneous team members in which decision making and information gathering are scattered. Since knowledge retrieval and

decision making can be represented as different roles and responsibilities within the team, the need for coordination grows.

The development of methods for assessing teams and for suggesting how teams should train such as described in the TADMUS study were used to guide the development of the team training framework discussed in this dissertation.

2.2 Training Simulations

Training simulations allow a trainee to experience real world conditions in a safe environment and help serve two training goals. First, trainees may attempt actions such as emergency procedures that would have too much risk in real situations and gain experience. Second, trainees are able to practice routine activities when real world resources are limited. One of the first training simulations were the Link Trainers modified to train military pilots in the skills of flying using instruments without outside visual cues, first developed during World War II (Singhal & Zyda, 1999). World War II saw the growth of training simulators for many fields as the influx of draftees to the armed services led to an explosion in the need for training. By the end of World War II training simulations had begun the transition from being mechanical to incorporating electronic systems for simulating the various subsystems. And training simulations had become an integral part of the training resources of the military.

Military and aerospace simulations have driven the development of the simulation field since the end of Word War II. From the mechanical nature of the early trainers such as the Link Trainers, modern flight simulators have seen a growth in their

fidelity and use. Modern flight simulators include a range of simulations from airliner crews to distributed networked individual fighters acting as squadrons.

In the next two sections we cover SIMNET, the prototypical and one of the most widely used simulation domains in existence, and we cover DDD, a testbed for Command and Control research.

2.2.1 SIMNET

The US Army began development of the SIMNET (simulator networking) in the 1980s (Singhal & Zyda, 1999). SIMNET was a distributed virtual environment simulation for crews of military vehicles operating as small units. SIMNET has three parts:

1. An event-driven architecture
2. Distributed independent simulation nodes
3. A set of predictive algorithms collectively referred to as "dead reckoning"

Collectively the three parts allow the construction of a heterogeneous simulation environment with varying capabilities as required that can be synchronized across a network without a central server. Throughout SIMNET, trainees play various roles and positions within the individual or crew simulators and can interact with other individual or crew simulators. Other aspects useful for the simulation world such as weather can be added as required.

The SIMNET architecture was formalized as the Distributed Simulation Protocol (DIS) (IEEE 1278.1 & .1a) in the 1990s (IEEE, 1997). The standardization of SIMNET fueled a growth in numbers and types of simulators that could be plugged into SIMNET.

The current follow on successor to SIMNET and DIS is the High Level Architecture (HLA) (IEEE 1516.x) (IEEE, 2000).

2.2.2 Distributed Dynamic Decision Making

The Distributed Dynamic Decision making (DDD) research tool was designed to meet the needs for empirical research for Command and Control technologies and environments. The DDD is implemented as a multi player, real-time simulation that provides a team of decision-makers with an air, sea and ground environment, a variety of task classes representing things to do, and controllable platforms that contain sub-platforms, sensors and weapons (resources). The DDD research tool provides the ability to conduct controlled experiments in a laboratory environment, using problems that are abstractions of "real world" command and control (Kleinman et al., 1996).

The design of DDD focuses on the dynamic/execution phase of a mission and allows for manipulation of key structural variables in task and organizational dimensions. DDD has the ability to constrain and/or to manipulate organizational structures such as authority, information, communication, resource ownership, and task assignment.

2.2.3 Scenarios for Training

The approach to a scenario driven exercise is to provide a structured training simulation that mimics real world events by having the trainees execute the actions and decision making in as realistic a setting as possible. The scenario is designed to exercise the

trainees in the situations most desired for training. Scenarios allow for multiple uses of a training environment to achieve alternate training goals as required.

With automation such systems have started to include computer generated forces as part of the scenarios. Computer generated forces are scripted to act with the appropriate behaviors and provide a more realistic learning environment to the trainee. Such scenarios may exist on top of a simulation system or a real world training environment.

Currently the development of such exercises is accomplished through trial and error by the developers of such scenarios. A description of scenario driven training exercises for the US Army is listed in their field manual on battle-focused training, FM 7-1 (TRADOC, 2003). In general, such training exercises use dedicated human coaches and self-critique to evaluate the performance of the trainees.

Self-critique works best when the trainees themselves are expert enough to be able to evaluate their own performance. But computer assisted instruction (CAI) systems have also been developed to assist in creating appropriate evaluation and feedback to trainees. Today such systems are more commonly referred to as Intelligent Tutoring Systems (ITS) or Intelligent Learning Environments (ILE).

2.3 Intelligent Tutoring Systems

Intelligent Tutoring Systems (ITS) are a realization of computing systems to provide instruction and guidance to human students in the learning of a task (Wenger, 1987). An intelligent training system offers one-on-one interaction with a learner in order to provide knowledge-based individualized instruction.

Artificial intelligence techniques have long been a part of Intelligent Tutoring Systems (Corbett et al., 1997). Such techniques have been used to move beyond just automated instruction systems towards realizing "intelligent" computer-based instruction. Artificial intelligence has been used to model a tutor that in turn can model the student and provide a customized learning environment to a trainee. Such a custom learning environment has been shown to provide a significant advantage over traditional learning methods such as classrooms (Bloom, 1984).

To be successful, the ITS must maintain state information about a trainee's progress in learning the domain being taught. Such state information is referred to as a user model. Or more specifically a student model when applied to learning.

2.3.1 Student Modeling

User modeling is the ability of a computing system to develop knowledge of a human user in order to facilitate interaction with that user (Wenger, 1987). User models can be used in training systems to represent a model of how the human learns the training objectives given the constraints of the problem to be solved. A student model is a specific case of a user model that focuses on building an understanding of a student's behavior and knowledge as it applies to the learning task and the student's performance at that task. A student model contains knowledge about the student's learning and knowledge of the training domain that may be used by the ITS to provide corrections to the student (Wahlster & Kobsa, 1989). In this dissertation, the phrases user model and student model represent the same thing.

The introduction of Artificial Intelligence (AI) to user modeling makes it possible for an ITS to build a user model that is based on inferring unobservable actions of the user's behavior (Wenger, 1987). The user model enables the ITS to build a representation of the user's knowledge that lead the student to take the observed actions. Since the ITS maintains a model of what the expected correct behavior is, the ITS can evaluate the performance of the student's mastery of the knowledge and adapt its instructions to improve those portions of the student's understanding that are known to be weak.

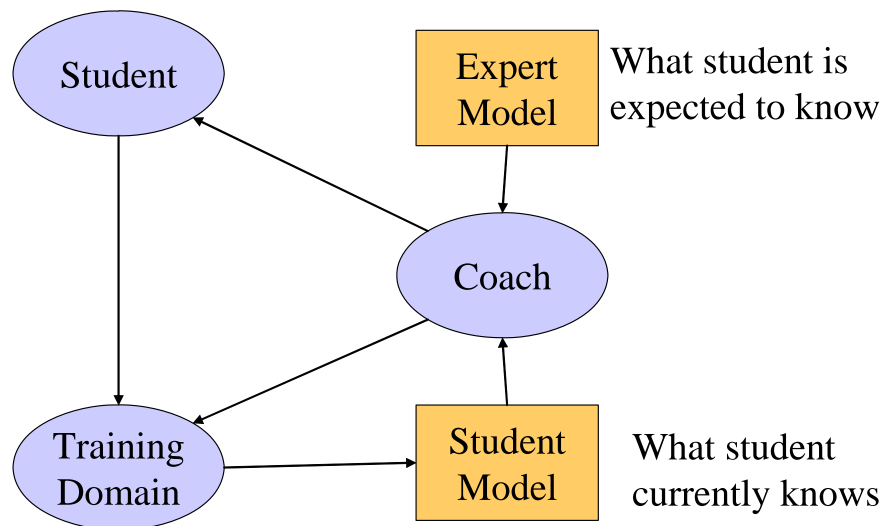

Figure 1: ITS with student model in operation

In order to evaluate the student model an ITS may have a model derived by a domain expert called the expert model as a basis for comparison. Shown in Figure 1 is a typical ITS with a student model and an expert model. The student model is constructed by the training domain based on a trainee's observed behavior in that domain. The coach evaluates the performance of a trainee by comparing the student model to the expert model. The coach is then able to provide a critique of a trainee's performance to a trainee.

One of the first approaches to user modeling in an ITS was the SCHOLAR system in 1970 (Carbonell, 1970). SCHOLAR provided a natural language interface for teaching geography. Carbonell's approach differed from previous module-oriented Computer Assisted Instruction systems in that he used a notion of a semantic network to encode knowledge in the ITS. The ITS may traverse the network in order to teach the student and be able to generate sessions of explanations and questions. Although SCHOLAR was simplistic in dealing with student errors, it did use its own representation of knowledge in the semantic network to model the student's performance. SCHOLAR kept a list of what nodes in the semantic net the student had asked, and if the node was answered correctly or incorrectly. It could thus track the student's progress through the material to be taught.

The overlay model is an extension of a simple user model which works by comparing an expert model to the user model and looking for gaps in the knowledge of the student. All such differences are viewed as a lack of skills on the part of the student. It is important that the ITS is able to instruct the student in ways that allow the student to

learn the knowledge using the same processes as the expert has done in order to get a match in the overlay once training is complete. An additional goal to consider is when to capture knowledge about the user. The approach taken by all student models is to adapt at run-time as information is learned about the behavior of the student acting in conjunction with the ITS.

One example of an overlay model was developed by Carr for use in WUSOR-II (Wumpus Advisor II) (Carr & Goldstein, 1977). WUSOR-II was a program developed to coach students in the skills to play the computer game Wumpus. The WUSOR-II system used a rule base and a critic (an evaluation of those rules) to detect inconsistencies in the student's behavior. The critic updated the overlay model when triggered by new evidence of the student's knowledge that could be related back to the rules used in WUSOR-II. Collectively the pieces were grouped as a module called the Psychologist that today would be called the coach.

A disadvantage of the overlay model is that it does not take into account information from the student that is either wrong or that is outside of the expert's domain. The ITS then no longer has a model of the student's behavior that matches the expectations of the system. Therefore the advice from the ITS becomes so out of context as to be meaningless to a trainee.

In order to correct for the incompleteness of the overlay model the differential model was introduced (Kass, 1989). The differential model instead compares differences in performance between the expert model and the user model in order to further subdivide the knowledge not known by the student. Unlike the overlay model, which

only knows what knowledge is not known by the student, the differential model also knows what the student could not know, and what the student does not know. The differential model is still not a complete model in that it still assumes the knowledge of the student is a subset of the expert's knowledge.

SHERLOCK is an example of an ITS that uses the differential model (Lajoie & Lesgold, 1992). SHERLOCK was an Air Force project to develop a computer-based coach that could be used to train technicians in trouble-shooting strategies. After classroom instruction, SHERLOCK could be used to provide situated on-the-job training in diagnosing the test equipment used by the technicians. In testing, performance of the trainees increased to match the skill levels of technicians who had been on the job for four years after only 20 hours of instruction with SHERLOCK. SHERLOCK took a simulation-based approach to training with the use of coaching as needed to improve on the cognitive task analysis abilities of the trainees.

SHERLOCK introduced the idea of the student trace. The student trace is a record of the student's action in the training system. The student trace became the input to the student model. The student model in SHERLOCK is an instantiated lattice of knowledge variables. This lattice is used by a knowledge system of rules to generate inferences about the student's ability.

Students may have different levels of prior knowledge and different styles of learning. Potentially the breadth of knowledge to be covered would require a student model that incorporates all that has been learned in cognitive science. However, John Self states that the complexity of the student model can be managed through focusing on

the training objectives and only supporting what is required to achieve the training objectives (Self, 1990). A super coach built to know and teach everything under the sun is not required.

According to Self, the question of building an appropriate student model can be made manageable by using four guiding principles.

1. Have the student show their work
2. Do not assess what you will not coach
3. Student models elaborate, not remediate
4. An ITS collaborates with the student

In essence John Self says that a useful student model does not have to know everything about a student and be able to provide an answer to every question. The first two principles point towards reducing the workload of the student model by reducing the problem space. Student actions map into the student model using aids such as computerized user interfaces. The student model only stores what is necessary. The last two principles illustrate a different approach to coaching based not on the belief that the instructor knows all and knows best. Instead, that the student is able to reflect on the problem and integrate such reflection into their understanding of the problem instead of being corrected. The forth principle casts the ITS in the role of promoting the student's understanding and not rigid instruction based on a fixed pedagogical style.

For this research the development of such student models for team training allows us to have a team-oriented student model that is supported by a team training framework for the set of individual trainees in a team. Although we take our own team-

oriented approach in developing such a student model, we do follow the basic principles laid out above.

2.3.2 Plan Recognition and Student Modeling

A further step in user modeling is to use plan recognition on the part of the coach or ITS. The ITS tries to identify what the student is doing and match it to one or more known plans of what needs to be done in order to accomplish the task. Such systems must have a library of plans or a way to generate plans based on goals and tasks that are part of the domain. These libraries are normally created by experts in the relevant domain. Detection of the current goal of the user by plan recognition can be difficult to accomplish (Allen et al., 1990).

Collagen updates a user model with plan recognition in order to reduce unnecessary communications and find the focus of attention of the user (Lesh et al., 1999). Lesh simplifies the plan recognition task by using partially elaborated plans and finding the current goals of the user in order to find out what the user is doing. Collagen will directly ask the user for further elaboration in order to keep the number of choices from becoming intractable. Collagen uses a tripartite model of intentions, focus, and conversation segments to narrow the scope of the plan recognition problem. Collagen predicts the short term focus of the discourse in order to provide appropriate responses and actions by the system to the user. The focus keeps the amount of extra dialogue to be generated lower than if no such predication was done. Therefore, the system is designed to reduce the amount of its own dialog in its responses from a human perspective. Whereas CAST searches in predicted future plans and finds essential communications to

pass among team members, Collagen focuses on the current activity to predict the next required communications to continue the dialogue.

A different approach is to introduce the idea of generating a user-tailored plan by the training system (Kupper & Kobsa, 1999). In such a planning framework, the training system will generate a plan based on the user's state in order for assistance to be given to the user that matches the user's preferences and needs. The intent is to be able to overcome the problem of maintaining a large library of plans for domains in which plans can have many variants. It also allows for adoption of a plan to suit user capabilities and different coaching methods. Their approach is to use a partial order planner in combination with stereotypes (a default initial view of a trainee) to build plans for giving advice to the user. Unfortunately they do not currently have a demonstration of their framework available yet.

2.3.3 Developing an ITS

In designing a team training framework based on an ITS methodology it would be useful to have some principles or guidelines for development. Corbett, Koedinger, and Anderson present in (Corbett et al., 1997) an overview on past developmental efforts with intelligent tutoring systems.

The overriding design principle for an ITS according to Corbett et al. is to "Enable the student to work to the successful conclusion of problem solving."

Corbett et al. go on to list eight principles for intelligent tutoring system design.

- Represent student competence as a production set
- Communicate the goal structure underlying the problem solving

- Provide instruction in the problem solving context
- Promote an abstract understanding of the problem solving knowledge
- Minimize working memory load
- Provide immediate feedback on errors
- Adjust the grain size of instruction with learning
- Facilitate successive approximations to the target skill

The first four principles relate to designing the learning environment with awareness to the cognitive issues of understanding the domain, knowing the training objectives, and educating the student to know the problems and solutions within the domain. The last four principles focus on learning by ensuring the student is not overloaded, recognizes errors, gains practice, and problem solving should reflect the actual final real-world environment.

The principles are focused towards designing a specific ITS for a specific domain. The team training framework as defined in this research will not be able to answer or meet all of these principles. Instead some of these principles or issues will have to be answered by the training system developer while integrating the framework into their specific domain.

2.4 Modeling Human Behavior

Intelligent agents have been used as a tool for representing reasoning about knowledge in a manner similar to human cognition. By giving an intelligent agent human-like

characteristics, domain developers and researchers can use such agents for modeling human behavior.

Modeling human behavior is typically done with mental models. Mental models are models of how human beings represent the state of the world in order to interact with the world (Johnson-Laird, 1993). Shared mental models extend the mental model to explain how humans maintain shared associations and observations of team members in order to predict the team members' behavior (Cannon-Bowers et al., 1995). Shared mental models are a form of implicit communication in that they allow team members to maintain awareness and understanding of each other's expected behaviors and understanding of the situation in the face of limited communications and high workloads. A simple example is that individual drivers on a road share a common knowledge of the rules of the road and so can anticipate the behavior of other drivers on the road. Drivers have both individual goals (e.g. go from point A to point B) and shared goals (e.g. avoid accidents with other drivers).

In this section we explore a common model (BDI) that intelligent agents use to represent human behavior. This approach is called Beliefs, Desires (or Goals), and Intentions (BDI) and is a model for developing intelligent agents (Rao & Georgeff, 1995). We give a brief overview of the BDI approach. We follow up with a discussion of one of the more influential models of a BDI agent, PRS (Procedural Reasoning System). Last, we discuss Petri Nets, a methodology for plan execution and representing teamwork. Petri Nets underlay the model of teamwork used by CAST.

2.4.1 Goals and Plans within BDI Agents

In the most common approach to developing intelligent agents (Rao & Georgeff, 1991), the behavior of a rational agent is driven by the beliefs, desires (or goals), and intentions of that agent. Intentions are treated as partial plans of actions that an agent intends to fulfill in order to achieve the agent's goals. An agent's beliefs govern the choices of goals and intentions undertaken in order to be successful. Figure 2 contains a summary of the salient points of the reasoning framework for a rational agent. Using the combination of beliefs about a world, goals to achieve in that world, and intentions that select towards those goals, an agent can act to change the state of a world.

Beliefs, Goals, and Intentions

- F is a state formula
- Belief(F) iff F is true in all belief-accessible worlds
- Goal(F) iff F is true in all goal-accessible worlds
- Intention(F) iff F is true in all intention-accessible worlds

- Action A results in state S being true
- An agent believes that if successful in doing action A, the agent will achieve state (goal) S, and so adopts an intention to do action A

Figure 2: Beliefs, goals, and intentions

For a BDI agent, the goals of the agent will select the intentions, which in turn drive actions and produce behavior. Given such a framework for rational behavior, an intelligent agent operates within a sense, decide, act loop within a domain. The combination of the sense, decide, act loop and beliefs, desires, and intentions allow a BDI agent to operate within an environment and act autonomously (Russell & Norvig, 1995). How an agent chooses to act is based upon plans that the agent either generates or that have been stored for use by the agent.

A plan can be formally defined as a data structure consisting of four elements: A set of steps, a set of step ordering constraints (such as "Si before Sj", which means that step Si must occur sometime before step Sj), a set of variable binding constraints, and a set of causal links (Russell & Norvig, 1995). A causal link is written as Si $\xrightarrow{C}$ Sj and read as Si achieves c for Sj. Causal links record the effect of steps in the plan. Therefore the effect of Si is to achieve the precondition C of Sj. A plan can thus be viewed as an ordered set of steps that must be completed in sequence and for which specific conditions must be met by a prior step in order to initiate follow on steps.

One of the most influential developments of the BDI model is the Procedural Reasoning System developed by Georgeff.

2.4.2 Procedural Reasoning System

In Georgeff's PRS (Procedural Reasoning System) (Georgeff & Lansky, 1987), a plan is represented through a procedural language. PRS is based on the BDI model, in which

beliefs are maintained in the database, goals (subset of desires) appear in plans and goal stack, and intentions are stored into intention stack.

According to Georgeff, plans are represented in PRS by declarative procedure specifications called Knowledge Areas (KAs). Each KA consists of a body, which describes the steps of the procedure, and an invocation condition, which specifies under what situation the KA is useful. The KA in PRS does not consist of possible sequences of primitive actions, but rather of possible sequences of sub-goals to be achieved.

PRS can be used to create and execute plans in a dynamic environment. However, PRS is implemented as a single agent that has no awareness of teamwork. Therefore, PRS does not incorporate any analysis about information needs for the plan, which could help proactive teamwork. Instead such analysis would have to be added to PRS.

2.4.3 Petri Nets for Plan Representation in Agents

Petri Nets have previously been suggested as an appropriate implementation for both plan execution in intelligent agents (Moldt & Wienberg, 1997) and for representing teamwork (Coovert & McNelis, 1992). Petri Nets are a graph that has two types of nodes. Transition nodes are fired to represent actions and place nodes hold tokens to mark the flow of control. When a transition node fires the tokens in the preceding place nodes are moved to the successor place nodes from that transition node. A Petri Net is shown in Figure 3 in two steps. In step one transition 1 is eligible to be fired. In step two, transition 1 has been fired and the succeeding place nodes have a token. In step two,

transition 2 is eligible to be fired however transition 3 still lacks a token in one of its preceding place nodes.

Step 1

Step 2

Both place nodes not filled

Transition able to fire

Transition not firing

Place node with token

Place node with no token

Figure 3: Petri net in action

Predicate Transition Nets (an extension of Petri Nets) can represent the dependence of actions on pre-conditions in a very natural way, that is, by true/false conditions upon input places to a transition (Yin et al., 2000). Such conditional places therefore force the control tokens to take alternate routes (e.g., places that are not blocked). In intelligent agents the Predicate Transition Nets represent plans of sequences

of actions that can be enacted (e.g., connected nodes) and the beliefs about what actions can or cannot be taken (e.g., filled/unfilled places).

For colored Petri Nets the tokens have values (Jensen & Rozenberg, 1991). These values are typically used to track resources as the Petri net progresses (e.g., counting totals for containers at each stage of their progress at a loading facility). The tokens may in fact be typed and /or manipulated by a functional programming language in the course of evaluating the colored Petri Net. The Agent-Oriented Colored Petri Net (AOCPN) system introduced by Moldt uses colored Petri Nets to synchronize the behavior of multiple agents in the AOCPN system through shared transitions (Moldt & Wienberg, 1997). Each agent uses the values of the tokens to assist in maintaining the 'mental state' of that agent. The agent 'mental state' consists of the beliefs and commitments of the agent. In CAST the value of the tokens are the names of the agents involved in executing the transition following the place node holding the token.

A failing of Petri Nets lie in their static structure. The static nature of the Petri Nets makes it difficult for dynamic planning to construct Petri nets as required. Since our current version of CAST does not have dynamic planning, we did not concern ourselves with this issue. The Rob-CAST system by Sen Cao does dynamically build links between already written team plans for control and information purposes (Cao, 2005).

The rationale for our use of Predicate Transition Nets was the ability to model concurrency, represent and identify information flows, and use decomposition to model plans and sub-plans. By automatically translating a team language, MALLET, into a Predicate Transition Net structure, we avoided needing domain experts to develop and

understand the Predicate Transition Nets and instead use a higher-level language to represent team knowledge and processes (Yin, 2001).

2.5 Agent-based Teamwork

The traditional intelligent agent is a single agent system. When intelligent agents incorporate teamwork into their capabilities, they now have to understand role assignments within the team and how to manage team concepts such as cooperation and coordination. As introduced by Tambe (Tambe, 1997), several teamwork theories have been proposed in the literature by Jennings (Jennings, 1993) and (Jennings et al., 1998), Cohen (Cohen & Levesque, 1991), and Grosz (Grosz & Kraus, 1996) which provide a guide for the design and specification of team-based intelligent agents. We will briefly cover two of the major theories in teamwork which are joint intentions and shared plans.

2.5.1 Joint Intentions

Intentions, as defined by Bratman (Bratman, 1987), define a mental state of an individual based on attempting to achieve specific actions. In a multi-agent system, the intelligent agents need to coordinate joint actions by extending their individual intentions to a shared model of intentions for completing those joint actions (Cohen & Levesque, 1991). For Tambe (Tambe et al., 1999), one model of multi-agent teamwork is to explicitly characterize a team's mental state (called joint intentions). The ability to represent shared beliefs dynamically is particularly helpful in changing environments, where team members may fail in achieving assigned goals or team members may be presented with new goals.

Joint Intentions theory focuses on a team member's joint mental state, called a joint intention (Cohen & Levesque, 1991). A joint intention is defined to be a joint commitment that the agents have agreed to as a collective action. A joint intention of a team Q is based on its joint commitment, which is defined as a joint persistent goal (JPG) to achieve a team action p. A joint persistent goal includes an escape condition q that enables a team to drop the joint persistent goal if the team members all mutually believe that q is false. The joint persistent goal is denoted as JPG (Q, p, q). The joint persistent goal requires:

- All team members to mutually believe that p is currently false.
- All team members mutually know that they want p to be eventually true.
- All team members mutually believe that until p is mutually known to be achieved, unachievable or irrelevant they mutually believe that they each hold p as a weak achievement goal (WAG).

A weak achievement goal can be represented as WAG (m, p, Q, q), where m is a team member in team Q, implies that one of the following conditions holds:

1. The agent m believes that p is currently false and has a goal to make p eventually to be true.
2. The agent m believes that p is true, will never be true, or is irrelevant (that is, q is false), but has as a goal that all the team members mutually believe the status of p.

The joint persistent goal guarantees that all team members will hold the commitment until p is mutually believed to be achieved, unachievable or irrelevant.

Therefore, each team member holds p as a weak achievement goal. So whenever one team member realizes that p is either achieved, unachievable or irrelevant, that team member needs to communicate to all other team members to confirm a mutual belief in the team before that team member can drop it.

Tambe founded his STEAM model on the joint intentions theory. STEAM incorporates team synchronization to establish joint intentions, and monitoring and repair capabilities. To form a joint intention, all team members must establish certain mutual beliefs and commitments. Such mutual beliefs can be done through the request-confirm protocol in STEAM (Tambe, 1997). STEAM therefore requires communication in order to build and maintain joint intentions.

2.5.2 Shared Plans

Instead of basing the initiation and maintenance of coordinated team actions on a joint mental attitude, the Shared Plans theory (Grosz & Kraus, 1996) relies on a theory of collaboration that looks not only at the intentions, abilities, and knowledge about actions of individual agents, but also the agents' coordination in group planning and acting. For Grosz, in multi-agent activities participants not only do means-ends reasoning about their own actions, they also reason about how to coordinate with and support the actions of others in the team. These joint activities require plan-based reasoning that arises from the participants' attitudes of intentions toward the actions of others and of the team as a whole. These joint activities are defined as shared plans.

A shared plan could be either a full shared plan (FSP) or a partial shared plan (PSP) because agents may have either partial or complete beliefs and intentions. In a partial shared plan;

1. Some sub-actions may have not deployed to any agent
2. The agents may have only partial recipe for doing an action
3. The agents may have only partial shared plan for doing some of subsidiary actions in the recipe
4. The agents may have only partial shared plan for some of contracting actions.

If capable, an agent can fill in partial shared plans to promote them to the level of full shared plans. Taken together, full shared plans and partial shared plans define a set of plans that can be used in a dynamic environment as required to coordinate joint actions.

2.5.3 Communications between Agents

Communication is the important issue for multi-agent collaboration and coordination. There are several challenging issues in agent communication, such as: what to communicate, to whom to communicate, when to communicate, and how to communicate. Communication languages have been designed to answer the question about how to communicate. As a starting point we can divide agent communication into three different levels: content, individual intentions, and joint intentions (Barbuceanu & Fox, 1995).

The first level is concerned with information content communicated among agents. One method for doing so is through a standardized communication language

such as the Knowledge Interchange Format (KIF) (Genesereth & Fikes, 1992). KIF, as one example, offers a standard format to be used among heterogeneous agent systems.

The second level specifies the individual intentions of agents. Knowledge Query and Manipulation Language (KQML) (Finin et al., 1997) is designed as a standard language for expressing intentions such that all agents would interpret them identically. KQML provides an extensible set of *performatives* for communicative actions that could happen among agents, such as ACHIEVE, ASK-IF, ASK-ALL, TELL, DENY, and ERROR. KQML also defines a set of policies (protocols) that constrains the legal sequences of communication acts, which induce a set of inter-agent conversation patterns using the communication actions.

The third level is concerned with the conventions that agents share when interacting by exchanging messages. Coordination Language (COOL) (Barbuceanu & Fox, 1995) was introduced to serve the third level. In the third level, communication is not only used for exchanging information, but as an example could be used for forming joint intentions.

With the knowledge of the goals, roles, capabilities, responsibilities and plans, team-based intelligent agents can perform belief reasoning and decide what, when, and to whom to communicate given a uniform set of methods on how to communicate.

2.5.4 Collaborative Agents Simulating Teamwork (CAST)

This dissertation is based on CAST as its underlying teamwork model and intelligent agent architecture so a short introduction is in order. In CAST, our hypothesis for generating efficient teamwork is: “A good classification and distribution of

responsibilities, capabilities, and effective belief reasoning can help us to generate cost-effective and timely information-flows/communication (through anticipation) within/between teams in large-scale agent systems". Under our hypothesis, based on the agent's knowledge of the responsibilities and capabilities of itself and all other related agents in the team, the agent can detect the information needs of other agents so as to provide information in a timely manner (Yin et al., 2000).

In contrast, the interpretation of joint intentions by Tambe prevents STEAM from doing so. STEAM constrains the joint intentions to the goals related to the shared complementary responsibility defined by CAST but STEAM goes no further. CAST has two additional responsibilities called shared competitive and redundant responsibilities. Shared competitive responsibilities are shared by multiple agents such that any one of the agents can carry them out independently, but if multiple agents take actions toward them, there will be some risk of failure because of conflict. Redundant responsibilities are such that any one of the agents can carry them out independently and if multiple agents take actions towards them, there will not be any damage in accomplishing them except for unnecessary effort.

Even for the goal related to the shared complementary responsibility, STEAM needs to go through a request-confirm communication protocol, which is not necessary in certain cases. For example, in the defense stage of a volleyball team, everyone has the shared complementary responsibility of passing the ball and attacking. There could be two possible situations, the first situation is that everyone in the defense team observes that it is an airball (served high and easy to return), and in CAST, without

communications, the team members establish the joint intention of passing the ball and attacking with the ball. However, in STEAM, the team members need to go through the request-confirm cycle, which is hinders fast responses. A second situation is that the ball is spiked by the other side, then every team member in the defense team has the shared competitive responsibility of saving the ball, by shared competitive responsibility we mean that saving the ball is the goal of the team, but only one of the team members can do it. In our method, someone near the ball will say “I got it” by informing others and also try to save the ball simultaneously with the shared competitive responsibility reducing the chance that two or more players will bump into each other. However, in STEAM, there is no way of taking this kind of responsibility without communications and therefore resulting in a collision with each other.

2.6 Agents in Intelligent Tutoring Systems

A modern extension to intelligent tutoring systems has been to model the elements of the ITS as intelligent agents. Instead of a monolithic ITS, elements of the ITS such as the coach or other entities in the training environment can be modeled as individual agents.

In the first of the two systems described below an intelligent agent is used in the more common approach to represent a coach or tutor that can interact with a trainee. A novel aspect of the implementation in Steve is the visual representation of the coach as a human in the virtual environment that can be used to demonstrate domain techniques to a trainee.

In the second system, Revised Space Fortress, an intelligent agent is used as a partner to offload a portion of the taskwork from a trainee. Offloading some portion of

the taskwork allows a trainee to focus on the remaining taskwork as a step in their learning process.

In both systems agents play roles as determined by the system designers to provide the most efficient learning that the designers can conceive. The first advantage for both systems over a monolithic approach is the ability to upgrade the capabilities of the individual agents as required. The second advantage is a more natural perspective in designing the system for the developer. Both of these ideas are used in this dissertation.

2.6.1 ITS with Coach Agent in Virtual World: Steve

Steve (Soar Training Expert for Virtual Environments) developed is an agent architecture that has been used in intelligent tutoring systems that require a virtual environment in which the students can interact (Rickel et al., 2001). Steve can appear in the virtual world as a human figure, or as a floating hand that can point at objects in the virtual 3D world and manipulate them. Steve uses the Jack virtual human software developed at the University of Pennsylvania (Lee et al., 1989) for presenting its virtual elements. Steve is built upon SOAR (Laird et al., 1996) as is its reasoning and belief system. Steve provides a single agent architecture in a standard ITS approach to training a single student in the performance of a complex task.

In the course of instructions, Steve is used to demonstrate the operations a student needs to perform and to monitor that the student performs the actions in the right sequence in the virtual world. Steve acts as both a demonstrator of the tasks that need to be performed and Steve performs as a coach that monitors and provides feedback to the student. The virtual world itself is a software simulation that reacts and provides visual

feedback, as the student would expect in working with the real world equipment. The simulation is used for training students for shipboard operations in US Naval ships.

2.6.2 ITS with a Partner Agent: Space Fortress

Space Fortress is a system for developing training protocols for learning complex skills (Donchin et al., 1989). The participants each use a joystick and mouse to manage their resources in flying a spacecraft with the goal of destroying the enemy fortress and maximizing their score for a set of parameters. A number of different pedagogical approaches have been tested within Space Fortress in researching automated instructional systems.

Of interest to this research is a pedagogical approach in which each partner of a two-person team alternates in performing half of a task with the other partner. In the partner experiment for Space Fortress, one participant uses the joystick while the other participant uses the mouse. Such training works by having the participants each practice on a specific part of the skills needed for the game and then rotate the partners to the other partner's role so that the participants also gain understanding of their partner's role and their own skills improve for the skill component being trained (Shebilske et al., 1993). One note is that the training team protocol in the partner study is used to improve the individual participant's score in the task and not for any metric of teamwork. The goal has been to instead reduce the use of resources required for training by sharing a single computer with two trainees simultaneously.

However, with the availability of cheap fast computing resources, a second approach has been devised that uses an agent partner instead of a human partner. In

Revised Space Fortress (Cao et al., 2004) a partner agent is substituted for the other human player and only a single human player and the partner agent perform the experiment. In the experiment the strategies employed by the partner agent were developed through previous studies and the partner agent tries to execute the optimal strategy for the role that the agent was playing during the experiment.

The partner agent was extremely limited in its capabilities and what capabilities it did have were specific to the Revised Space Fortress domain. Nevertheless, it is an interesting demonstration of the use of an agent acting outside of the role as a coach in an ITS and instead acting as a training assistant through the use of offloading taskwork and providing the human trainee with an example in action of the partner's role.

3. ISSUES IN DESIGNING A TEAM TRAINING FRAMEWORK

Intelligent tutoring systems have been promoted as a solution to the needs of training by offering the ideal of a one-on-one tutoring technology that is adapted to the personal needs of a trainee. In the past, intelligent tutoring systems have focused on the training of an individual exclusively even if multiple people were being trained in the same session. If we wish to train individual learners to act as members of a team, we need more than a training system designed for a single user. One needs to model the elements of a team such as teamwork activities and multiple team members. However, the exact mechanisms needed to most effectively use intelligent tutoring to support team training are not yet known, and each effort to build a distinct experimental system is expensive. Our approach is through the use of a team training framework for developing such team training systems. Such a framework would incorporate a model of teamwork combined with interfaces for integrating to simulation domains and supporting team-oriented assessment and coaching.

In order to allow training system developers to experiment with different intelligent tutoring mechanisms for team training, we introduce several capabilities in the team training framework created in this dissertation. First, we incorporate the ability of intelligent agents to work as a team member with human trainees. To do this the team training framework incorporates the notion of virtual team member. Second, we incorporate the use of intelligent agents to monitor and assess the performance of trainees, both for individual task work and for cooperative teamwork. Third, we incorporate the capability for building various forms of coaching agents that can

automatically provide feedback to trainees. The provision of these capabilities by a team training framework raises a number of issues that must be addressed in this research.

To provide a virtual team member to replace the other members of the team being used, those virtual team members have to execute taskwork within a training domain and use teamwork to communicate with each other and the trainees. To provide monitoring, assessment and coaching of individual trainees acting as team members, a team training framework must support domain sensing, sensing of both virtual and human team member actions, the development of domain specific assessment of team actions, and the development of a variety of forms of domain specific coaching paradigms.

The issues in training for teamwork that must be considered in developing a framework span multiple types of teams and domains. However, in this dissertation we focus on training the types of command and control teams as discussed in section 2. This dissertation covers those issues in training with the teams as have been defined above and within the context of a workstation-based domain simulation. The team is composed of humans working towards common goals using computer-based resources for completing tasks and handling communications.

The above general issues expand into a substantial number of detailed issues that must be addressed. The issues that this research addresses are as follows:

- Replacement of actual team members with virtual team members
    - Agents as team members interacting in a simulation domain

    - Interfaces for both acting in a domain and reading environmental cues to connect a virtual team member to the simulation domain
    - Requirements on a training domain for providing the scenario capabilities and implementation of interfaces that can be used by a team training framework to allow virtual team members to interact with the domain in a timely manner
- Interfaces that permit the interoperation of human trainees and intelligent agents as virtual team members (naturalness of agent behavior from the human perspective)
    - Explicit communications such as messages between team members that might have a number of formats to support (e.g., speech, text, visual cues)
    - Implicit communications such as augmentations to the human interface by the simulation domain to incorporate communication acts between team members (e.g., domain mechanisms that allow transfer of information as opposed to explicit messages)
    - Extraction of desired interactions among team members for the purposes of coordination, synchronization, and information exchange from team plans by virtual team members
    - Extraction of the desired activities of each virtual team member from team plans
    - Assigning roles and responsibilities among virtual team members

- Monitoring the interactions of a human trainee with the other team members (human or agent) with respect to the team plan
    - Matching individual trainee actions to team needs as a related to achieving team goals
    - Identifying that information needs for other members of the team are provided by a trainee or acquired by a trainee
- Interfaces that permit the inclusion of a variety of performance assessment modules that can be used in a variety of coaching paradigms
    - Provision of a generic set of assessment modules and access to virtual team members (e.g., a trace of the actions undertaken by a trainee for use in other assessment modules)
    - Domain specific assessment of a trainee’s performance is supported to fulfill training goals
- Interfaces that permit the inclusion of an agent-based coach developed in accord with a team training framework
    - Interfaces to support developing an intelligent coach agent
        - To access individual trainee and team assessments
        - To present coaching evaluations to trainers or trainees
    - Interfaces for different forms of interactions among a coach agent, trainees and agent-based team members
        - Interaction with a trainee during or after a training session

- Interaction with a virtual team member agent to act on behalf of a coach agent

Each of these issues is explored in Section 3. In Section 5, the approach taken to resolve the issues is described.

A summary of the issues to be discussed is as follows:

- Integrating a simulation domain
- Communications for teamwork
- Replacing team members with virtual team members
- Monitoring trainees
- Performance assessment
- Coaching for teamwork

3.1 Integrating a Simulation Domain

The human team members expect to participate in a training scenario within a simulation. The need for interactivity between a trainee and a real-time exercise places a requirement on a simulation domain to be able to generate events that a trainee can then act upon and see the results of their actions. The trainee must believe that the simulation domain provides an experience rich enough to learn from actions taken and mistakes made during the training scenario.

It is a requirement that the simulation domain provide an application programming interface that can be used by intelligent agents to act and sense in the domain in an equivalent way to what a human does. A key issue is defining

requirements on any such API that allows its use by generic domain independent code provided by a team training framework.

For the virtual team member to function properly within a team training framework, a team training framework in conjunction with the simulation domain must address the following issues:

- Support issuance of domain commands by virtual team members
- Acquisition of domain knowledge by the virtual team members

3.1.1 Issues on Executing Commands in the Simulation Environment

The actions that can be executed by an intelligent agent acting as a virtual team member must include those that can be executed by a team member. In particular, actions requiring physical action by a human must be performable in some way by an agent. For example, an action accomplished by the push of a button or the click of a mouse in the team member's user interface to the simulation domain must also be doable by an agent.

For an agent we define an action as an external or internal operator. An external operator is a discrete domain command. Domain commands for an agent are based on equivalent human commands in the domain. An internal operator is a computable mental activity. Such mental activities are typically domain specific. An example would be calculating possible collisions in an air traffic control domain. An operator for an intelligent agent should match at a logical level (executing a command or a domain computation) an action taken by a human.

These operators can be put together as sequential or parallel actions in plans to be executed by team members. As an example, the actual command for the human operator

in order to move a vehicle on a screen from location A to location B might involve the following physical steps:

- Move cursor over vehicle
- Right click to show menu
- Move cursor to select the move command
- Move cursor and select a point for the vehicle to move towards

It may not be necessary to have an intelligent agent perform the same intermediate steps as the focus of the entire exercise is on the domain logical action; move a vehicle. Instead, an agent may execute a move command with the three parameters of the vehicle id and the destination x and y locations.

The expectation is that a team training framework should be able to provide a domain independent interface that a training systems developer can use to map API methods in a simulation domain into actions that an agent can execute. This interface should support both domain actions and logical operations that can be executed by an agent acting as a virtual team member. Additionally, domain actions may be continuous. By continuous we mean actions such as the movement of a mouse across a screen. Therefore, a framework must answer how it expects to handle such continuous actions.

3.1.2 Issues on Sensing the Simulation Environment

In the simulation domain, human team members acquire knowledge about the environment, are able to react to events, and monitor state changes as they occur. Therefore, acquisition of all domain variables pertinent to knowledge acquired by any team member must be accessible to a virtual team member in our case. Domain

variables include what a team member can observe. A team training framework must provide a sensing interface to that a training systems developer can use to enable the acquisition of knowledge about changes in the domain.

If humans obtain feedback visually, for example, by seeing where a track is, the agents replacing the humans need to obtain the same feedback by some other means. If there are sounds generated to the human, some abstraction of these sounds containing the same information content must be made available to the agent. Moreover, if one human can observe the actions of another, there must be a mechanism for the agent to do so also. The objective is that a team training framework provides the interfaces and mechanism to allow an intelligent agent acting as a virtual team member to sense the relevant domain variables that a human participant would be able to sense.

The actual acquisition of the domain data (knowledge of the domain and monitoring of changes) by a virtual team member is subject to four issues. The four issues are how, when and what the data is acquired, and the format for the data.

For the first two issues, data should be updated to a training system built using a team training framework in a manner appropriate to the limitations of a particular domain simulation. The "how" issue means that a generic mechanism for handling domain specific data must be developed so that each individual training system built using the team training framework does not have to reinvent such a mechanism. It should be the case that only minor work, such as identifying the specific domain variables to be sensed, should have to be done to tailor it to a specific system.

The second issue, “when,” involves a synchronization between framework provided capabilities and domain provided capabilities. There are two possibilities (from the perspective of the domain), push and pull. In a push mechanism, data is pushed out to a team training framework at a tempo established by the domain. In a pull mechanism, data is pulled on demand from the domain or at specified intervals as needed by a team training framework. It would be ideal that a team training framework is able to support building training systems that use either push or pull. Data acquisition should be done at a rate to allow the agents to reason and act within the same time frames as a human would.

For the third issue, what data is acquired is based on what knowledge of the domain simulation a human team member would require. A virtual team member must then be able to access that same set of knowledge, but no more than this. This issue places a requirement on the domain to provide exactly that set of knowledge as domain variables in a format that is recognizable by interfaces provided by a team training framework.

The fourth issue of data format becomes a burden on a team training framework. Since the framework is a consumer of the domain state data, the framework must be able to accept the data and manipulate the data into a format for use by the virtual team member agent architecture. This issue of data format requires a standard representation of the domain variables from what could be a multitude of domain simulations.

3.2 Communications for Teamwork

A team training framework must support a mix of humans and software agents acting together as peers. For training purposes, the primary communication concern for team domains is the communications between team members. While not all human communications are relevant, commands, coordination, or other team activity that requires communication within the training domain must be recognized by a team training framework. The vast range of communications types can be a challenge to support. Therefore, the training system developer must make a final determination of which communications types will need to be supported in a particular domain.

The issue for a team training framework then becomes that it must support a number of communication formats (e.g., speech, text, domain mechanisms that allow transfer of information, etc). A team training framework must provide interfaces that support team member to team member communications such as agent to agent, agent to human, and human to agent. A team training framework must also provide interfaces for supporting translation of human/agent communications to forms understandable by either entity.

Within communications we define two types, explicit and implicit. Explicit communications are those messages exchanged directly between team members. Explicit communications are identifiable as visible channels of communications provided for the team to interact. Examples are email, voice intercoms, and other send/receive channels.

Implicit communications are information exchanges for teamwork and coordination purposes that are not team member to team member messages. An example

of such information exchanges is the use of information systems in teams that allow team members to update knowledge about the situation into the information system. Such knowledge then becomes available to other team members who query for that knowledge in their interface to that information system. Such systems are common in the military such as AWACS, AEGIS, and other battlefield information systems. An example used in this research is the Task Assignment Panel (TAP) in Section 6.1.2 that was added to the DDD software.

Implicit communications may be more difficult to recognize as there is not a visible communications channel. Instead implicit communications are typically implemented as a post/query system within a domain. A team member will post information as to their intent or knowledge to the various databases or systems that exist within a domain. Other team members must then query (or observe) such information in order to infer the intent of the poster or to extract knowledge for use in their taskwork and teamwork.

For a team training framework five types of interactions need to be supported.

1. Agent to agent
2. Agent to human
3. Human to agent
4. Human to human
5. Observation-based

3.2.1 Issues on Agent to Agent Communications

Agent to agent communications are the most straight forward type of interaction to support but present issues for the other two types of direct interactions (agent to human and human to agent). Given a virtual team for a training domain, if the training domain has domain communications mechanism (e.g., text messaging or other built in messaging capability) it is useful to allow a team training framework to plug into these mechanisms. Using the domain mechanisms allows such a framework to exploit any recording or other capability of the domain communications mechanism. Conversely the framework must support being plugged into the domain.

Since a team training framework would replace human team members then the issue of handling the communications of these virtual team members must be answered. The ramification of communications for virtual team members is how these communications will interact with human trainees (next two subsections) and training domains.

In respect to training domains, a team training framework should provide solutions for both having its own communications channels to support agent to agent communications (may be necessary for messages additional to human needs) and to support what the domain provides to its human team members for communications. Both still require that the framework provide an interface for handling communications. This requirement, at a minimum, must have support for an interface with a send/receive capability between the agents.

Agent to agent communications also raises the question of what format and type of messages will be required. Agents may require other coordination or information exchange messages additional to human needs. If so then a team training framework must make it possible for the training system developer to choose how or whether to display these agent messages to human team members.

3.2.2 Issues on Agent to Human Communications

As part of training, agent to human messages should be displayed to humans in as natural a format as possible and as close as possible to the human-to-human communication that a message of same format would have. As an example, human may receive messages in forms such as text, displays, or verbal. Agents must then generate messages that use such forms of communications in order to maintain the naturalness of the training. Judicious use of one or more of these forms allows an agent to interact in as natural a manner as possible, which can have a dramatic impact on how favorably the agent team member is viewed by human team members. It can be argued that is desirable that not only does an agent act with a consistent and expert behavior, but that the agent also acts in a natural and realistic manner (Ioerger et al., 2003). The benefit of natural behavior is to keep the human trainee from being disconcerted by behavior that is unexpected (not human like).

A basic send/receive capability allows a team training framework to support agent to human domain-based communications. If both humans and agents can use a domain messaging capability then the framework supports message exchanges for agent to human and human to agent communications. However, the framework must then

support two additional interfaces. The first interface supports translation of agent messages to a human usable form. The second interface supports the reverse capability. Both interfaces depend on the training system developer to determine what messages are relevant and how messages will be translated from one team member to another (human or agent).

If domain messaging capabilities do not exist then the issue for a team training framework is to still support a basic send/receive capability that can be extended by the training system developer. The two additional interfaces are still required. In this case additional work would be required to add or mimic a communication channel within the team that is usable by the agents in communicating to the human team members.

3.2.3 Issues on Human to Agent Communications

In the reverse of the situation described in Section 3.2.2, human team members (trainees) will have the need to communicate to virtual team members (agents). In a training session communications to agents by humans will be required for the purposes of task coordination and information exchange. Human trainees will expect to use the normal communications channels within the training domain to send these messages. This leaves the requirement of forwarding those communications to the virtual team members to a team training framework. The bigger requirement is that the agent will also be required to parse and understand those messages to some degree in order to respond in an appropriate manner.

In turn these require a team training framework to be able to handle four issues with respect to the virtual team members. First, the framework should provide a means

to parse and understand messages in order for the agent to act appropriately. Second, the framework should provide a means to act in response to a coordination request. Third, the framework should provide a means to integrate knowledge gained from an information exchange for use in the receiving agent's knowledge system. Fourth, the framework should provide a means to handle requests for information by understanding the request and sending a reply. The main point here is that if the framework can provide these capabilities, then the training systems developer need not be defocused from his/her main objective in order to provide them.

The most important issue is the understanding of the human message. A team training framework must provide interfaces and/or mechanisms by which the training system developer can incorporate domain specific mechanisms for translating a human message into a format understandable by an intelligent agent. Ultimately the handling of such a translation is a domain issue but the interfaces provided by the framework must be flexible enough to incorporate whatever solution the training system developer chooses.

Integration of knowledge gained from an information exchange requires that a team training framework support the storage and maintenance of such knowledge. Such a requirement may be subsumed in the requirements for a virtual team member (discussed in Section 3.3).

Requests for information require that an agent send a reply through interfaces as described in the previous section. Satisfying such requests require support for querying

the knowledge base of a virtual team member and then using the interfaces of Section 3.2.2 to send the reply.

3.2.4 Issues on Human to Human Communications

Human to human communications within the team may exist as there may be multiple human trainees in the team. Such communications may use domain mechanisms or may be entirely free form such as verbal, and expressions (hand gestures or facial gestures).

At the current levels of technology, it will be impossible to provide a substitute for all communications (e.g., speech, gestures) that exist between human team members to allow agents to interact as naturally. More realistically, until natural language processing is a part of the every day computer experience, a team training system will need to restrict the communications acts to what is currently technically capable.

Messages are structured around their intent, their intended audience, their timing, their format, and their method of transmission. Given a specific domain, the nature of the messages sent is typically restricted to information needs of a domain specific type; for example AWACS controllers query other controllers and pilots on task specific information and coordination needs. The training domain therefore allows us to place limits on the requirements of the communications needed for the agents to interact with other humans.

For a team training framework the issue is the monitoring of human to human communications for purposes of assessment and/or observations by the virtual team members. There should be provision for supporting the training system developer in either recording such messages for later processing or for supporting the inclusion of

such messages into the observations of the agents to whatever extent the training system developer is able to provide an understanding of such messages. Whether or not to process such messages is very much domain dependent.

3.2.5 Issues on Observation-based Communications

Implicit communications are those communications that do not involve an explicit message sent directly from one team member to another. An example of implicit communications in the AWACS domain is the identification tag that is written by a controller for a newly identified track by that controller. The controller is then able to post such identification tags to the central database in the AWACS software. The identification tags then appear on other controller's screens without a specific action by the other controllers. The other controllers may then act upon the new knowledge as required by their role in the team. However, the controllers do know that the information was provided by another team member and not generated by the domain. How the domain generates such information may not necessarily match other display or detection mechanisms and therefore must have special consideration.

For a team training framework, implicit information exchanges must become visible to other elements within the framework. Elements such as virtual team members, monitoring agents, and assessment and coaching depend on an accurate representation of the information available to a trainee in a domain. In the example given above the first controller issues a command to tag the new track and the second controller is then able to read the visual radar screen to see the newly tagged track.

Such implicit communications occupy an important enough role in teamwork that due consideration must be taken by the training system developer to identify and represent implicit communications within a team training framework.

The specific issues for a team training framework are to provide mechanisms for incorporating such implicit communications into training systems built using the framework. For the framework such implicit communications are observations and therefore based on sensing. Thus, the issues overlap the sense mechanisms issues described in Section 3.1.2.

3.3 Virtual Team Members

Training domains traditionally expect that all team members who are participating are humans, and hence are generally built with human/computer interfaces to allow their participation. Introducing virtual team members adds to the challenges of the simulation domain. The virtual team members must be able to interact with the simulation domain in all major respects expected of a normal team member.

This interaction places two additional requirements on a team training framework to what was detailed in Section 3.1 on integrating with a simulation domain.

The first additional requirement for the framework is that an agent acting as a virtual team member must be able to act in a timely manner with the simulation domain. The reasoning and interaction capabilities of an agent must fall within the norm of the expected behavior of a human team member in order to function as a virtual team member. The framework is required to maintain a synchronization of time between the simulation and the agents maintained by the framework.

The second additional requirement of the framework is to provide a virtual team member who can operate with minimal domain specific software modifications for the creation of a specific training system. However, once a training system has been built that allows an agent to function in a domain, that agent must have goals and therefore plans to follow to achieve those goals which are not part of the training system, per se. Rather, there must be a language provided in which goals and plans for use by the agents can be encoded. Furthermore, that language should support notions of teamwork including coordination and communication.

Because of the above considerations, we can categorize the requirements for allowing a virtual team member to operate with a simulation domain into two categories:

- Requirements on the simulation domain time management
- Requirements on an intelligent agent for use as the virtual team member

3.3.1 Issues on the Simulation Domain Time Management

The training domain is a simulation of a real time environment. The real time aspect is necessary for posting events to human trainees in order for the training to approximate real world conditions. Simulations such as SIMNET (previously described in Section 2) are expected to be event driven simulations of real world environments.

We break the issue of time management into four requirements for a team training framework.

- The framework has a representation of time usable across multiple simulation domains

- The framework has interfaces to read/update the state of the simulation domain as desired
- The framework has mechanisms to regulate the response time of the agents acting as virtual team members

The first requirement is that a team training framework be able to process time events that integrate with the time representation used by the simulation (e.g., discrete or continuous, constant cycle or event driven). Simulations considered in this research are software-based and run on digital computers that are, by nature, discrete. Therefore the simulations are updated at points in time. The timing and nature of the updates must be accessible to the framework.

Regardless of how time is updated within the simulation, the time updates must be accessible to a team training framework. A common model is to post events at the time step in which the events occur and forward the information to the software that would use it. It is possible to either push the entire simulation state or only the state changes at every time step. Alternatively, the user could register for the view/events/state change the user is interested in receiving. The registered events may then be pushed by the domain simulation or pulled by the user. For the framework the requirement is to provide generic mechanisms able to support cyclic or event driven simulations, and an additional requirement is to support either a push or pull model of updates as provided by the simulation server.

The third requirement is that the agent must match in its execution of operators to the expected speed of a human in executing that same action. It would be inappropriate

for an intelligent agent to perform in one millisecond that which takes a human a second or more. A human being takes a measurable amount of time to react to sensory data and generate a response (Ashcroft, 1994). For an activity that will take cognitive effort, that response time may be measured in seconds, not milliseconds. Otherwise, the agent's performance may prove disruptive to the learning of a trainee. Conversely, if the agent is too slow (e.g., abnormally long time for reasoning) for the domain then the agent will not act as an expert team member should. It has been found that the naturalness of the agent's behavior can have an impact on the learning behavior of a trainee (Ioerger et al., 2003). The naturalness is in part determined by the speed of the agent versus human reaction times. The requirement on a team training framework is to provide mechanisms for allowing the training system developer to regulate the response time of the agent.

3.3.2 Issues on the Virtual Team Member

Modeling the desired teamwork of team members (whether human or intelligent agent) within a command and control team is a primary requirement of any team training system The approach taken in the agent-based teamwork literature is to model each individual team member as an intelligent agent. The intelligent agents are then given the capacity to reason about other individual team members and to act to coordinate and communicate as required to function as a member of the team.

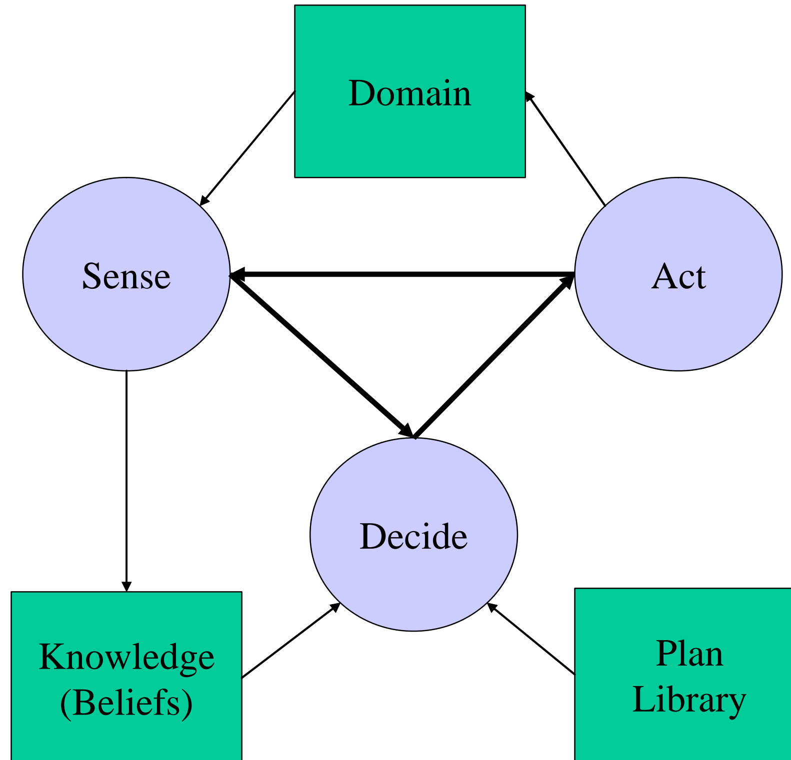

Figure 4: Logical view of intelligent agent

Figure 4 presents the standard logical view of an intelligent agent architecture as represented in the literature (Russell & Norvig, 1995). The agent uses a sense, decide, act loop to act within a domain following plans stored in a plan library. The sensing of domain changes combined with the beliefs (facts) stored by the agent are used to guide the decision making of the agent. Action within the domain is enacted by the agent once decision making is completed for that cycle.

The virtual team member should provide a basic set of capabilities that are generic to all domains. Ideally, the training system developer will be able to focus on those aspects of their domain they wish to capture (e.g., domain specific actions, plans, and interactions). A team training framework must provide the following capabilities.

- A structure within which a model of teamwork can be realized
- A decision cycle acting on a sense, decide, act paradigm
- Interfaces to support additional intelligence as required by the training system developer

A common approach is to represent the desired teamwork as a set of plans to achieve the goals of the team (Yin et al., 2000). These plans specify the individual and joint roles of each member of the team. These plans are then executed by the virtual team member agent following both individual and team intentions in order that the agent function as a member of a team. The specification of these plans is not part of building a training system using a team training framework, but is a necessary part of using a training system, once built. We assume that the training systems developer will develop suitable plans for describing both agent behavior and desired trainee behavior (expert model).

A key issue for a team training framework, then, revolves around the automated extraction from those domain specific plans of the interactions among team members for the purposes of coordination, synchronization, and information exchange in order to create believable virtual team members. When an intelligent agent acting as a virtual

team member executes these plans, that intelligent agent is able to assume the role of a human team member being replaced within the confines of a specific training domain.

The requirement of a decision cycle still leaves open to a team training framework on how to incorporate decision making within an intelligent agent suitable for team training. There is a requirement that an intelligent agent be able to act as a virtual team member. This implies a reasoning requirement on the intelligent agent. We can state that a virtual team member requirement imposes a minimum decision making requirement on the intelligent agent. The minimal decision making requirement is a knowledge system that can maintain the state of the agent's plans and goals with respect to the other team members (human or agent).

However the final intelligence (capability) of an agent acting as a virtual team member within a specific domain may not be fully answered by a team training framework. The level of expertise required by agents for instructional purposes is only beginning to be explored and thus an agent built now should be designed to be capable of a range of levels of expertise. The framework should be able to provide interfaces that can be extended to allow the training system developer to add the level of expertise required of a specific training domain.

3.4 Monitoring Trainees

Monitoring the actions of a human trainee and the other team members (human or agent) is required to allow for the collection of data necessary for a coach to determine success or failure for the team in a domain. These monitored domain actions are crucial inputs from a training domain for a team training framework. A list of these domain actions and

the state of the simulation at the time of each domain action are known as a student trace (Katz et al., 1994). The student trace is the time ordered set of trainee actions in the domain. The student trace provides the training system developer with a structured data set to be used to create a domain-based student model. For the framework monitoring a trainee's actions is separated into two requirements.

The first requirement is monitoring individual trainee actions and the capturing of other associated state information from the simulation. This monitored data must be made available for use by the assessment and coaching elements of a team training framework. For team training, trainee actions may be further subdivided into individual taskwork and team interactions. Individual taskwork are the domain actions a trainee undertakes to accomplish individual and team goals. Team interactions account for the communications and coordination acts that a trainee undertakes to fulfill that trainee's roles and responsibilities within the team. Therefore the individual student trace must be extended to account for the teamwork of an individual trainee in relation to their team members.

A secondary requirement on a team training framework is the additional capability to monitor the incorporated virtual team members. Additional to what was discussed in Section 3.3 on virtual team members, a team training system should be able to supplement the monitoring of a trainee with respect to the progress of the virtual team members by recording the actions taken by the virtual team members.

3.4.1 Issues on Monitoring Trainee Actions

Knowledge of what actions have been taken by a trainee allows an intelligent tutoring system to determine what has been learned and applied by a trainee within the domain. Domain specific actions may constitute both taskwork activities and teamwork activities. However, a prime requirement for a team training framework is for monitoring trainee actions within a specific domain in a generic manner. An additional consideration in monitoring a trainee is what the framework can do in a domain independent manner in providing generic team-level knowledge that can be readily utilized in a variety of training domains.

All of the following requirements must be done by a team training framework in a generic manner.

- Capture domain actions executed by a trainee
- Capture the state of the environment observable to a trainee
- Store the action and state information
- Provide access to the action and state information

Given the distributed nature of some simulation domains careful consideration must be paid to where the monitored data has been monitored, stored, and then must be accessed from. This is a burden on a team training framework which applies to all of the monitoring requirements.

One of the requirements for a team training framework in capturing domain specified actions in a generic manner is to provide an interface for use in monitoring actions. Only the domain system developer can determine what constitutes actions in the

domain and provide a method (or interface) for capturing those actions. However, the interface provided by the framework must specify a form for receiving such domain specific actions in a manner that is generic across multiple domains.

A team training framework has the requirement to provide an interface for use by the domain to send monitoring domain state knowledge as could be observable by a trainee. A training system requires that the state of the domain in which an action is enacted be known for assessment purposes. Therefore, a team training framework must provide an interface to capture the state of the domain in relation to the actions undertaken by a trainee in the training domain. It might be possible to relate this requirement on the framework in terms of the requirement on sensing by a virtual team member in Section 3.1.2.

Once captured, the monitored data must be stored by a team training framework for access by assessment and coaching support. This support may be generic or domain specific and therefore the data should be stored in a generic manner and accessed through generic interfaces.

3.4.2 Issues on Monitoring Interactions with Team Members

Monitoring is complicated by not only the actions of a single trainee team member but that of what the other team members' actions may have impacted on a trainee. Teamwork is both the individual actions of team members and how those actions interact (i.e., coordinate and communicate) with other team members. Therefore interactions by a trainee with other team members are a key component of team training and these interactions must be recognized and monitored.

Teamwork is divided into those acts (both communicative and domain) for coordination and communications. A team training framework must monitor these teamwork activities even if these activities have already been captured as domain actions. The significance of a domain action for teamwork (i.e., a communication action or an action for coordination) is that it involves other team members. As opposed to the single user focus of a traditional ITS, the team perspective involves having to correlate the actions of multiple team members. This focus expands the framework as a potentially multi-trainee ITS with correlation between individual trainee actions.

Team interactions initiated or received by a trainee entail the requirement for monitoring both a trainee and also the other team members. For a team training framework the trainee monitoring task has already been discussed in the previous subsection. A team training framework has the requirement to provide interfaces for the following activities:

- Monitor virtual team members
- Identify information exchanges and coordination events

A team training framework must ensure that the virtual team member logs its actions and the events that are observable by the virtual team member. The framework must then ensure the logged data are stored and are accessible to the assessment and coaching interfaces of the framework.

Individual team member information needs for accomplishing required sub-goals within the team must be detected for each trainee and virtual team member in the team. These information needs provide both the information exchanges within the team and

the redundancy that may be available to substitute for failures by a trainee in providing for those information needs. Sub-goals are both individual actions by a trainee and plans of expected actions that the team must fulfill. Therefore just capturing a trainee's actions is not enough, as a model of the teamwork must be used by a team training framework.

The requirement on a team training framework is to have methods for identifying and marking both information needs and coordination needs. This requires the framework to have a model of teamwork that can be used to both describe and identify these teamwork needs. Furthermore the framework must have interfaces for allowing assessment support to query the framework for these teamwork needs. However, the training system developer establishes the roles and responsibilities of the team members in relation to these teamwork needs. Therefore the framework must provide interfaces and/or mechanisms to allow the training system developer to encode these roles and responsibilities.

3.5 Performance Assessment

Simultaneous monitoring of all team members (human and agent) allows assessment not only of individual performance in regards to team activities but also assessment of a trainee in regards to the overall team performance. Given that the monitoring requirements of Section 3.4 have been met by a team training framework, the framework must support both generic team assessment and domain specific assessment needs. Both of these needs involve the individual trainee (or trainees) and the team involved in the training.

A team training framework should support provision of a basic standard set of assessment services. Although one cannot envision all of the assessment modules that might be built within the framework, a few basic modules should be provided as proof of concept. These basic modules will also support the generic team level knowledge. In addition to providing a proof of concept these basic modules may offer building blocks to the training system developer for two purposes. The first purpose is to offer a coherent view into the model of teamwork being used and the interfaces to both query and use that model. The second purpose is to give an initial set of assessment support to the human coach and the domain for testing and incorporation of a team training framework into the training domain.

Individual assessment assesses the taskwork performance of a trainee based on criteria set by the training system developer. This domain specific assessment requires access to the monitored data stored by the framework. Additionally, the framework must support the execution of assessment modules and the forwarding of assessment results to other assessment or coaching modules that requires those results.

Team level assessment is focused on analyzing the monitored data to determine quantified metrics of teamwork performance such as the amount, timing and appropriateness of communication and coordination activity. The actual assessments are again divided into two categories, generic and domain specific. It is desired that a team training framework provide certain generic domain independent teamwork analysis such as the amount of communication. In addition the framework should provide interfaces

by which the training system developer can incorporate domain specific team assessments.

3.5.1 Issues for Individual Assessment

Individual assessment is based on the taskwork. It is also based on those components of teamwork which are domain specific. For assessment to work a team training framework must have a number of interfaces and methods:

- An execution interface for execution of assessment modules
- A data interface to allow access to monitored data
- A result interface for assessment results
- A method for connecting assessment modules

A domain developer will expect to develop an assessment module as part of a chain of assessment and feedback to meet specific training objectives. A team training framework must therefore provide a structure for supporting such assessment. An execution interface supports execution of assessment modules by system built using the framework. The interface must be able to load and execute domain specific assessment modules in a generic manner. Execution may vary depending on the needs of the assessment module so mechanisms for timely execution of the assessment modules should exist. How the assessment modules access one another may be crucial such as a hierarchy of assessment modules building upon assessments of other modules. Therefore, a capability within the framework to specify the ordering of when assessment modules execute in relation to each other is also required.

A data access interface must be provided by the framework to support access to the monitored data. This requirement may be similar to the requirement in Section 3.1.1 on storing and accessing domain actions.

Providing access and storage interfaces for assessment results have two requirements. The analysis expected of the domain assessment modules will fall into two categories; real-time analysis and post-session analysis. These analyses must be stored. The results may be used by other assessment modules or coaching modules during the execution of the simulation. The need for generic access and storage of assessment results may require methods such as introspection. Post session analysis and storage implies that the second result requirement is to store the results in a file (permanent storage) for use after the training session has ended.

3.5.2 Issues for Team Level Assessment

As with individual assessments, team level assessments are categorized into generic and domain specific assessments. It is desired that the framework provide certain generic assessments. In particular, the number of times each team member communicates with another, the number of times a team member is involved in proactive communications (Fan et al., 2006), and the number of coordination messages can be identified in a domain independent manner and should thus be determined. Nevertheless, the most useful team assessments are likely to be domain dependent, and a significant issue to be addressed in the creation of a framework is the design of generic mechanisms that enable domain specific assessments to be easily incorporated into training systems built using the framework.

The key difference from individual assessments is that such assessments are done across the entire team and may therefore be based off all of the trainees and the virtual team members. Therefore, for both domain specific and generic team level assessment an additional consideration must be made to give such assessments access to the monitored data and individual assessments for each of the team members.

In many respects the issues for supporting domain specific teamwork assessment mirror those issues raised in the previous section for domain specific individual assessment support. In addition, for both individual and team assessments, modules may require access to one another. It is conceivable that a training system developer might develop a hierarchy of assessment modules that build on top of one another in providing more detailed (focused) assessment as desired. Therefore assessment modules should abide by a standard interface. It would be useful for the framework to provide an interface for allowing assessment modules to access one another through mechanisms such as reflection.

The framework may also have to impose requirements on the training system developer in the form of naming conventions for supporting a hierarchy of assessment modules. For such purposes the assessment modules should have a minimum of a unique name and support access mechanisms to allow other assessment modules to pull needed assessment data from other assessment modules as required.

3.6 Coaching in Support of Teamwork

Coaching is the expert system (or human expert) that provides pedagogically motivated instruction to a trainee in support of the learning objectives. How this is done is not the

critical issue for a team training framework. However, mechanisms to support the incorporation of a wide variety of coaching paradigms and the most common modes of operations and interactions for coaching do need to be addressed by the framework. In supporting coaching, we would like to include instructional feedback from the coach to a trainee that is generated either as in-session interactions or as a post-session review. Possible modes of interaction include: 1) generation of reports for review by a trainee, 2) real time generation of changes in behavior of the simulation, or virtual team members, or 3) direct interaction with a trainee.

In order to put some structure on the coaching support we desire to create, consider the operation of a very general coaching system (Goettl et al., 1998). A list of the typical steps in such a system is listed below:

1. Execution of next simulation step in session
2. Collection of monitored data about trainee
3. Performance of assessments of trainee (processed data)
4. Evaluation of trainee (generation of possible feedback/interactions to facilitate training)
5. Present feedback to trainee or perform interactions at appropriate time
6. Repeat 1 - 5 until session ends
7. Execute post-session coaching evaluation
8. Present feedback to trainee for post-session review

Steps 1 – 6 are referred to as the *in-session* phase, and steps 7 and 8, as the *post-session* phase. Not all of these steps may need to be accomplished with a single coaching module. And not all steps may be necessary for a specific domain.

Some of the requirements posed by the above list are already requirements that have been raised in this section such as integrating a domain, monitoring, assessment, and synchronization of the framework to a domain. Of interest in this section are steps 4 to 8. The focus for in-session coaching support is for interfaces to support the generation of advice and the relaying of that advice to a trainee. Post-session coaching has an additional requirement to store the generated results.

3.6.1 Issues for In-session Coaching Support

The first aspect of in-session support for coaching is the need for individual and team assessments discussed previously in Section 3.5. Assuming these can be incorporated into a system built with the framework, the framework should provide mechanisms to allow domain specific evaluation modules to be included; in the terminology of the framework, such modules should generate the appropriate forms of feedback or interactions with the training system. What is of primary concern to the design of the framework are the mechanisms by which the feedback or interactions, once created by the domain specific modules, are utilized, specifically, the interfaces that allow domain specific results to be incorporated into specific systems and used. In order to allow very general coaching paradigms to be developed, several methods of display or interaction should be allowed.

A team training framework should support explicit feedback from a coaching application to a trainee during or after a training session. Explicit feedback is one-on-one interaction or display of knowledge by the coach with a trainee. In-session support for explicit feedback by the framework requires interfaces be provided to display in the feedback to a trainee in a domain specific form determined by the training systems developer. By implication the coach module must also be able to send feedback to any trainee if there is more than one human trainee in the session.

Since the team training domain includes both a simulation domain and virtual team members, there is the possibility of another form of feedback by the coach. As an example, a coach might direct another team member to ask a trainee for information that the trainee should have already provided. Such indirect feedback through the actions of another team member may not necessarily be recognized by a trainee as coaching feedback. However, such indirect feedback offers the advantage of a natural way for the training system developer to deliver coaching feedback. The framework should support indirect feedback such as using virtual team members to coach a trainee. In order to meet this requirement the framework must provide interfaces to the training system developer to remotely access the virtual team members and alter their behavior. A training system developer alters the behavior of the virtual team members in order to deliver coaching feedback to a trainee in the form of actions or messages by that virtual team member.

3.6.2 Issues for Post-session Coaching Support

The significance of the post-session phase is the ability to collate the monitored data and provide coaching in regards to the performance of the trainee for the entire training

session. Provision should be made in a team training framework to support the use of both generic and domain specific assessment results for post-session coaching. For the post-session phase, there are thus requirements to execute summary assessments and to save the generated data at the conclusion of the in-session phase. First, the framework should detect and announce a change in the execution of the coaching (and assessment) from the in-sessions phase to the post-session phase. Upon this change, any summary assessment and evaluation modules should be executed.

A second issue is the storage of the monitored data and generated assessment data made during the in-session phase (and, perhaps, the beginning of the post-session phase). Storage results requirements are similar to those as described in Section 3.5.1. A secondary issue is formatting the stored data in a form amenable to being manipulated as desired by the training domain. This could be as simple as using an ASCII file format (such as is used in UNIX systems) to using formatted data storage forms such as XML. The key issue is how to create a generic structure by which domain specific information can be captured, formatted, stored and accessed.

4. COLLABORATIVE AGENTS SIMULATING TEAMWORK

CAST (Collaborative Agents Simulating Teamwork) is the agent architecture upon which the CAST-ITT (Intelligent Team Trainer) is built (Yin et al., 2000). Jianwen Yin and Michael Miller wrote the original version of the CAST software. CAST is a belief, desires, and intentions agent-based software architecture implemented for researching teamwork. The focus in CAST is the support of proactive information exchange and maintaining the shared mental model believed to be the best representation of how human teammates maintain the cohesion of the teams they work in.

CAST has two conceptual levels. The MALLET language provides a mechanism for describing teams, their plans, and the role and responsibilities of the team members. The CAST kernel provides the necessary algorithms to execute the MALLET team specification at run time. Incorporated into the CAST kernel are Java interfaces that allow modular support for other intelligence algorithms.

The software implementation of CAST used in this research has been extensively revised from the original software. There also exist multiple versions of CAST which all share the same underlying philosophy and concept. Each version will be given a brief overview later in this section.

The current domain independent multi-agent architecture CAST v3 (referred to simply as CAST in this dissertation) is designed to simulate effective teamwork by capturing the knowledge about team structure and teamwork process. A team structure specifies membership, roles, and capabilities of individuals on the team, whereas a teamwork process specifies plans for the team to accomplish its goals. The common

prior knowledge about the team structure and process enables the team members to establish a shared mental model (introduced in Section 2.5).

The shared mental model has two elements. The first element is the plans written in MALLET and their translated representation using a Predicate Transition Net for each plan and associating the current execution of the PT Net with the current state of the plan's execution. The second element is the knowledge base composed of facts on the current state of the team and the team roles. This knowledge base provides declarative knowledge on the structure of the team and how the different team members relate to each other in terms of roles and plans.

Team execution for an agent has two elements. The first element is the MALLET plans which encode the behavior of a team member in respect to goals, actions, and events. The second element is the algorithms within the CAST kernel for detecting and fulfilling information needs, plan coordination, and required communications.

Based on the information needs of other team members, agents may reason about the team state and the need of their teammates and adopt a coordination and communication strategy for determining whether and when they should provide to or request from teammates regarding information needs. Agents act and coordinate according to the plans provided to them. Furthermore agents are able to search the plans of other team members and find both information sources and needs of those other team members. Sending out information needs for other team members or requests for information to other team members is automated within the CAST kernel and occurs

implicitly as required while plan execution is being done. The key idea is this notion of explicit plan execution based on implicit information exchanges and role assignments.

CAST is designed to operate in a distributed fashion as each agent has its own thread of execution. Using distributed agents allows support for a distributed user simulations. Each user (operator) in a simulation is distributed to a specific location (e.g., their own terminal and/or specific physical location). Distributed agents can be co-located to access the same resources and operate under the same requirements as the operator they are replacing. The agents can also be centrally located on a single workstation for testing purposes and for non-distributed simulations.

A CAST thread of execution can be modified in its execution pace to allow for control of the speed of the CAST architecture in relation to the execution cycle of the domain simulation and/or human response speeds. Combined with a generic domain interface, CAST can be plugged into a multitude of teamwork domains.

4.1 Multi-Agent Logic Language for Encoding Teamwork

CAST provides MALLET (Multi-Agent Logic Language for Encoding Teamwork), a language for specifying membership, roles, agent capabilities, goals, tasks, operators, and team and/or individual plans. MALLET has three goals as a language (Fan et al., 2006):

- To allow experimentation with different levels and types of team intelligence
- To provide a mechanism for encoding teamwork knowledge about roles, responsibilities, and declarative knowledge

- To provide a mechanism for encoding teamwork processes through procedural knowledge

A complete description of the original version of MALLET can be found in Jianwen Yin's dissertation (Yin, 2001). A BNF representation of the current MALLET language (version 2) is included as Appendix A. A complete description of the semantics of the current MALLET v2 is available (Fan et al., 2006).

Operators in MALLET are assumed to be discrete commands that are issued by an agent. MALLET does not support continuous commands such as a joystick or the fine movement of a mouse. Instead commands are expected to be issued in order to achieve a discrete change in state within the simulation (e.g. launch a vehicle, shut off a valve).

4.1.1 MALLET and CAST

We separated the definition of the MALLET language from its implementation in CAST. This separation of definition and implementation allows us to explore a number of different approaches to modeling teamwork. We can use various models of communications, observability, synchronization, or collaborative decision making within CAST to implement the teamwork being invoked in MALLET.

There are currently five implementations of CAST for MALLET. The first implementation was the core of Jianwen Yin's dissertation efforts. The second variant is the successor to that original version that is used in this dissertation for the CAST-ITT. The third variant has a decision theoretic implementation for handling information value but does not completely implement MALLET (Fan et al., 2004). The fourth variant extends the CAST developed for CAST-ITT to add a component for observability

(Zhang et al., 2002). The final variant focused on a more restrictive implementation of the shared mental model and a different implementation of the process net in an effort to improve the performance of synchronization actions (Cao, 2005).

Knowledge encoded in MALLET can be divided into declarative and procedural. Declarative knowledge is the team and domain knowledge in MALLET. Plans are the procedural knowledge.

4.1.2 Declarative Knowledge

Declarative knowledge refers to representations of objects and events and how these representations are related to other objects and events.

MALLET encodes declarative knowledge about the team structure and about domain knowledge. Team structures such as roles and responsibilities describe capabilities of the individual team members. Additionally, team goals describe the intentions and desires of the team. All of this knowledge is encoded as predicates.

The specification of roles and responsibilities establishes the uniqueness of each team member and defines their capabilities. First, roles allow the team members to know their position in the team. Second, responsibilities establish what the team members should be doing or monitoring. A summary of the constructs that MALLET can provide for roles and responsibilities is listed below:

Roles:

- **Agent Name** – name of a team member
- **Team Name** – name of the team
- **Role Name** – name of a role in a team

- **Member Of** – a team by an agent
- **Plays Role** – what agent can play what role

Responsibilities:

- **Goal** – ending state
- **Start** – starting plan
- **Capability** – what plans and operators can a role do
- **By Whom** – what team member does what task in a plan

Some of the above constructs are duplicates and have been added to ease usage. The CAST implementation of MALLET is expected to provide an inference engine. Therefore in MALLET, rules about roles, responsibilities, behaviors, and conditional states may be specified as conjunctions of predicates.

(Team BLUE (DM0 DM1))

(Agent DM0)

(Agent DM1)

(Plays-role DM0 controller)

(Role controller (move transfer launch recover identify attack fusion))

(Capability DM1 (move launch recover identify attack))

The above examples illustrate a trivial team construct with a team called **BLUE** that has two members, **DM0** and **DM1**. **DM0** has a role of **senior** and that role defines the capabilities of **DM0**. **DM1** has its capabilities defined using the **Capability** predicate.

MALLET may also encode facts and rules about a domain for use in plan execution and decision making. Additional environment facts during execution about the

domain are provided for use to the inference engine for evaluation of the rules declared in MALLET.

4.1.3 Procedural Knowledge

The second element in MALLET is the procedural knowledge that the team will execute in order to operate as a team. Procedural knowledge is expressed as MALLET plans. The MALLET parser in the CAST variant used in CAST-ITT compiles team plans into Predicate Transition Nets, which are a partial representation of agents' shared mental model about the status of the actual execution of team plans.

Plans are a procedural description of teamwork processes, e.g., how team members will achieve the goals or perform the tasks. A goal defines the success state of a team. Goals are achieved by the execution of plans whose end state matches the goal. Plans may be either individual or team plans. An individual plan is executed by a single team member. A team plan is executed by more than one team member. An individual or team plan may consist of invocations of operators, sub-plans, or arbitrary combinations thereof using various constructs such as sequential, parallel, contingent, iterative, etc.

A team plan has additional information or hints as to the needed roles for a plan or the specific team members needed to execute some portion of the team plan. A team plan will have one or more team members who execute sections of the plan and must coordinate their actions because of synchronization needs contained in the team plan. The team plan can include individual actions and individual sub-plans.

Synchronization in team plans is based on responsibilities. Agents are assigned (or volunteer) to complete actions (or steps) within plans. Certain types of actions may

involve one or more agents in their execution. If this occurs, then the agents involved ensure that the execution ordering is maintained or joint actions are completed by using synchronization messages.

The team plans model the desired interactions of team members (whether human or intelligent agent) within a command and control team. Each team plan must list the conditions under which that plan is used, the effects of that plan, and the individual steps taken by each team member in executing the plan. Plans in MALLET have the following characteristics:

(Plan name (variable list)

(Preconditions) (Effects) (Termination-conditions)

(Process …)

)

The preconditions, effects, and termination conditions are all rules for allowing the intelligent agent to enter and exit plans and set the mental state of the agent as required.

The combination of executing plans, synchronizing team actions, and team knowledge foster a shared mental model for each agent. The shared mental model enables the agents to coordinate their plan activities by knowing when to execute specified sub-plans or when to wait until the sub-plans should be executed or are completed (if another agent is executing it). However, each agent maintains its own view (copy) of a shared mental model of the team.

For plans a process is constructed to allow execution of the plans as steps. Steps do not always describe commands in the domain and instead may specify cognitive activities. Examples of cognitive activities are computations that are time consuming to implement in the inference engine of CAST. Following is a summary of the major process constructs in MALLET:

- **Seq** – takes a list of processes and executes them in sequence
- **Par** – takes a list of processes and executes them in any order
- **Choice** – takes a list of processes and executes them in order until one completes successfully
- **Joint-do** – allows coordinated execution of a list of processes by a team of agents
- **Agent-bind** – dynamically selects team members to satisfy various constraints
- **While** – a conditional loop
- **If** – a conditional branch
- **Do** – execute an operator or plan

Plans do depend on the changes in domain predicates and execute domain commands. Domain predicates are those predicates that are provided by a simulation domain and represent the state of the domain as viewed by an agent. Although the kernel executes plans in a generic manner, MALLET and a specific domain implementation work in conjunction to provide domain specific plans and domain specific actions.

4.2 CAST Architecture

The CAST architecture is designed to explore issues in representing and simulating teamwork using agents. By combining the declarative and procedural knowledge specified in MALLET plans with the domain knowledge, CAST agents are able to execute plans in accordance with the coordination and synchronization needs of the team in a domain. CAST agents are individually capable of acting as team members and communicating with other CAST agents while working within a domain as specified by plans encoded using MALLET.

CAST has the following key features:

- An intention structure that executes the agent's individual role within a given MALLET team plan for the kernel
- A process execution system that controls the selection of plans and determines which agent can accomplish what part of a team plan to execute
- Communication mechanisms in the kernel that do proactive information sharing between teammates
- Coordination mechanisms in the kernel manage coordination of tasks that require the combined individual work of multiple team members
- A domain interface for using CAST within a domain simulation

In the next four sections we explore the internal structures of the CAST kernel and how they guide decision making, the representation of communications and the information flows as generated by the IARG, the process of executing plans using

Predicate Transition Nets, and the visual and programming aides provided by this version of CAST.

4.2.1 CAST Kernel

Each CAST agent has a kernel. The kernel implements the sense, decide, act loop of an intelligent agent. In the decision step of the kernel are invoked the algorithms and data structures listed above that allow the agent to function as a team member.

The basic agent cycle follows the sense-decide-act paradigm as follows:

1. Mark new cycle, this provides for housekeeping related to sense updates and expiration.
2. For all plans in progress, check the preconditions of all actions next in the each plan's processing order.
3. Randomly select one action whose preconditions are true. If no preconditions are true then select a null action.
4. Issue a command to execute the action selected.
5. Go to Step 1.

There are two aspects to the interaction of the agent simulation and the domain simulation: 1) obtaining inputs from the environment, and 2) providing commands to the domain. Inputs from the environment are acquired in step 1. If a command is issued in step 4, that command is executed in an environment through the domain specific implementation of the ActorDomain abstract class (shown in Figure 5 below under the cast3.dynamic package).

Illustrated in Figure 5 are the packages and classes of the CAST agent. The primary packages are the CAST kernel, the MALLET parser, the Predicate Transition Net, and utility classes. The CAST kernel includes all necessary algorithms for maintaining the representation of teamwork and the activities of the team members. The MALLET package includes the declarative knowledge and the decomposition of the team and individual plans into the Predicate Transition Net.

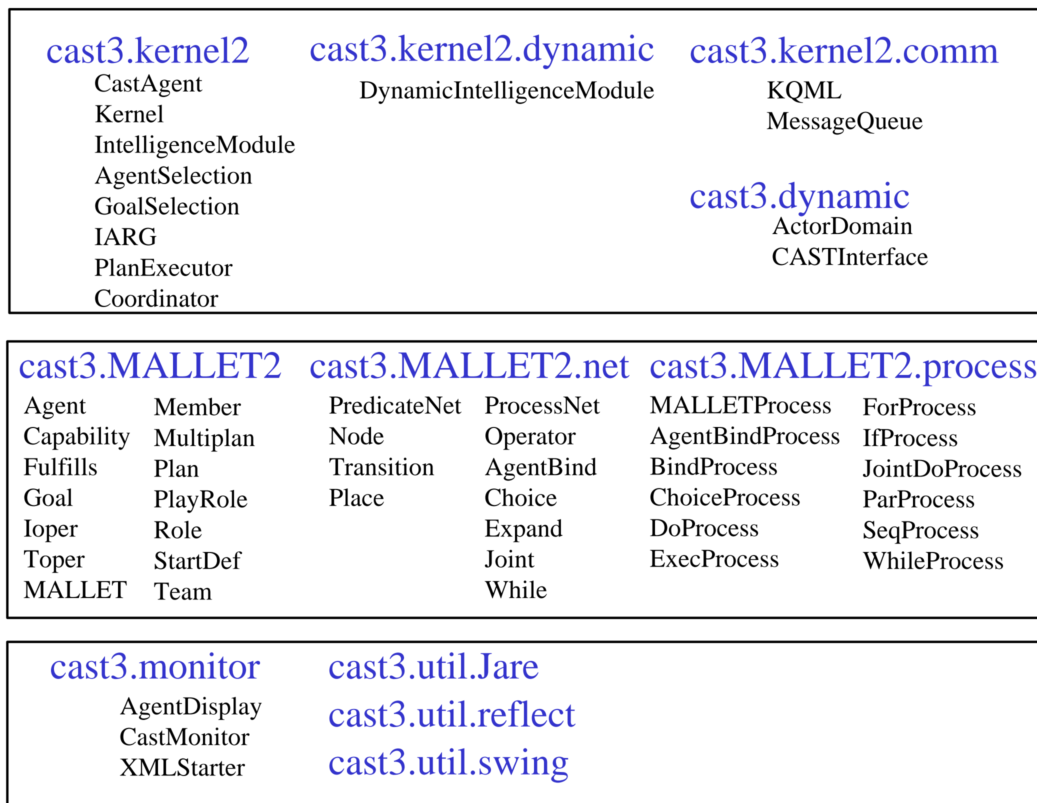

Figure 5: CAST agent packages and classes

Each CAST kernel has its own copy of a backward-chaining inference engine called JARE (Java Automatic Reasoning Engine) (Ioerger, 2003). JARE uses the knowledge base of an agent to evaluate logical conditions and constraints in a MALLET condition as part of plan execution.

Within the CAST kernel is the IntelligenceModule base class. The idea behind the IntelligenceModule is to separate the various capabilities of the intelligent agent into a set of services that can be used by other intelligence modules as required in an easy to access manner. Algorithms within the kernel are derived from the IntelligenceModule class. The basic algorithms in the CAST kernel are:

- PlanExecutor – Executes the steps of a plan implemented by a Predicate Transition Net for the current plan the agent is using to achieve its goals.
- GoalSelection – Simple implementation of a goal regression algorithm to determine what goals the agents must achieve. Based on these goals the algorithm selects plans to be given to the PlanExecutor to execute.
- AgentSelection – Selects the appropriate agent for a step in a team plan when the choice of agents is not bound to a specific agent. In addition, it handles role assignments and multiple agents by ensuring each agent either participates if required in a plan or is given the correct responsibilities within a plan. The basic behavior is that once an agent is selected for a plan, that agent should execute that plan.
- IARG – (Inter-Agent Rule Generator) Implements the proactive information exchange algorithm. It searches the Predicate Transition Net

structures versus the knowledge base to find when to provide information based on need between team members as defined in Jianwen Yin's dissertation (Yin, 2001).

- Coordinator – Handles coordination of team plans among multiple agents.

In addition to these five modules, other intelligence modules may be added at run-time using the DynamicIntelligenceModule abstract class. The purpose of this abstract class is to support extension of the basic reasoning capabilities of CAST.

While all of the algorithms are essential to making team coordination work in CAST, the IARG algorithm is supports efficient teamwork by reducing unnecessary information sharing. Therefore, the next section will be devoted to this algorithm.

4.2.2 Inter-Agent Rule Generator Algorithm

Efficient teamwork relies heavily on information sharing, especially in dynamic environments. Having humans in the loop places an additional constraint on agent-based team members that they must interact with human teammates in a natural way (e.g. not overload the humans with repeated information). The key is to try to supply only the most relevant information, and identifying this information need requires reasoning about their goals and responsibilities on the team.

Dynamic information exchange in CAST is accomplished by an algorithm called Inter-Agent Rule Generator (IARG) that operates within the CAST kernel to identify potential information requesters and providers. The IARG engages in the following activities:

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

IARG uses two algorithms, ProactiveTell and ActiveAsk (Yen et al., 2001). ProactiveTell sends information acquired by an agent to other agents as the provider agent determines is necessary. ActiveAsk queries other agents that are expected to know information that is required by the needing agent. They are listed in Figure 6 as they are a crucial piece of how CAST is designed to work.

```
ProactiveTell
If I is a newly-sensed piece of information, or I is a
post-condition of the last action P taken by self,
then
     If I is not in this agent's knowledge base,
     assert I into knowledge base
     For each information-flow <predicate, needers,
     providers>
          If I matches predicate name and self is
          included in the providers, then
               For each agent x in the needers,
                    If agent x plays a role in an
                    active step,
                    Do TELL(x, I)

ActiveAsk
For each active step s in the plan in which self is
involved
     Let o be the operator to which s refers
     For each pre-condition I of s
          If not (know (self, I)), then
               Let Info-flow = <predicate, needers,
               providers> be the information flow in
               which I matches predicate name
               Select an agent y from providers
               (active agents first),
               Do ASK(y, I)
```

Figure 6: ProactiveTell and ActiveAsk algorithms

The IARG algorithm detects the information needs of other team members and the information needs of itself (as the algorithm is running within an agent's kernel). The IARG algorithm identifies information needs in the plans, roles, and responsibilities of each agent as defined in MALLET before and during execution. The IARG then uses the current state of its agent's knowledge base to find if new information has been added through the completion of tasks and changes in the environment as observed by the agent. When the IARG identifies information needs, these needs are provided to other

team members or if a need for the agent itself is identified then a list is complied of other team members that can provide the needed information to the agent.

The IARG algorithm depends on the structure of the plans and the knowledge (preconditions and effects) associated with each plan. The next section details the execution structure of the MALLET plans in the CAST kernel which is done by the use of Predicate Transition Nets.

4.2.3 Predicate Transition Nets

The procedural knowledge represented in MALLET is compiled into a Predicate Transition Net as a computable model of the agent's mental state. The Predicate Transition Net serves as an execution path for each of the agent's plans. Predicate Transition Nets are enhanced with control nodes that represent the agent's beliefs about which team member agents are performing the current actions.

The Predicate Transition Net is a natural representation for parallel action and synchronization in a multi-agent world (Sowa, 2000). Transitions can represent actions, with input places corresponding to pre-conditions and output places corresponding to effects. We extend the standard Predicate Transition Net formalism with special kinds of places called control nodes and belief nodes. Control nodes represent the belief an agent has about the current goals and activities of others in the team. Belief nodes represent the belief an agent has about the world, when coupled with an inference engine such as JARE, can represent first-order knowledge, including dynamic facts and inferences about the world. The Predicate Transition Nets play the dual role of monitoring (beliefs) and tracking (control) the execution of team plans.

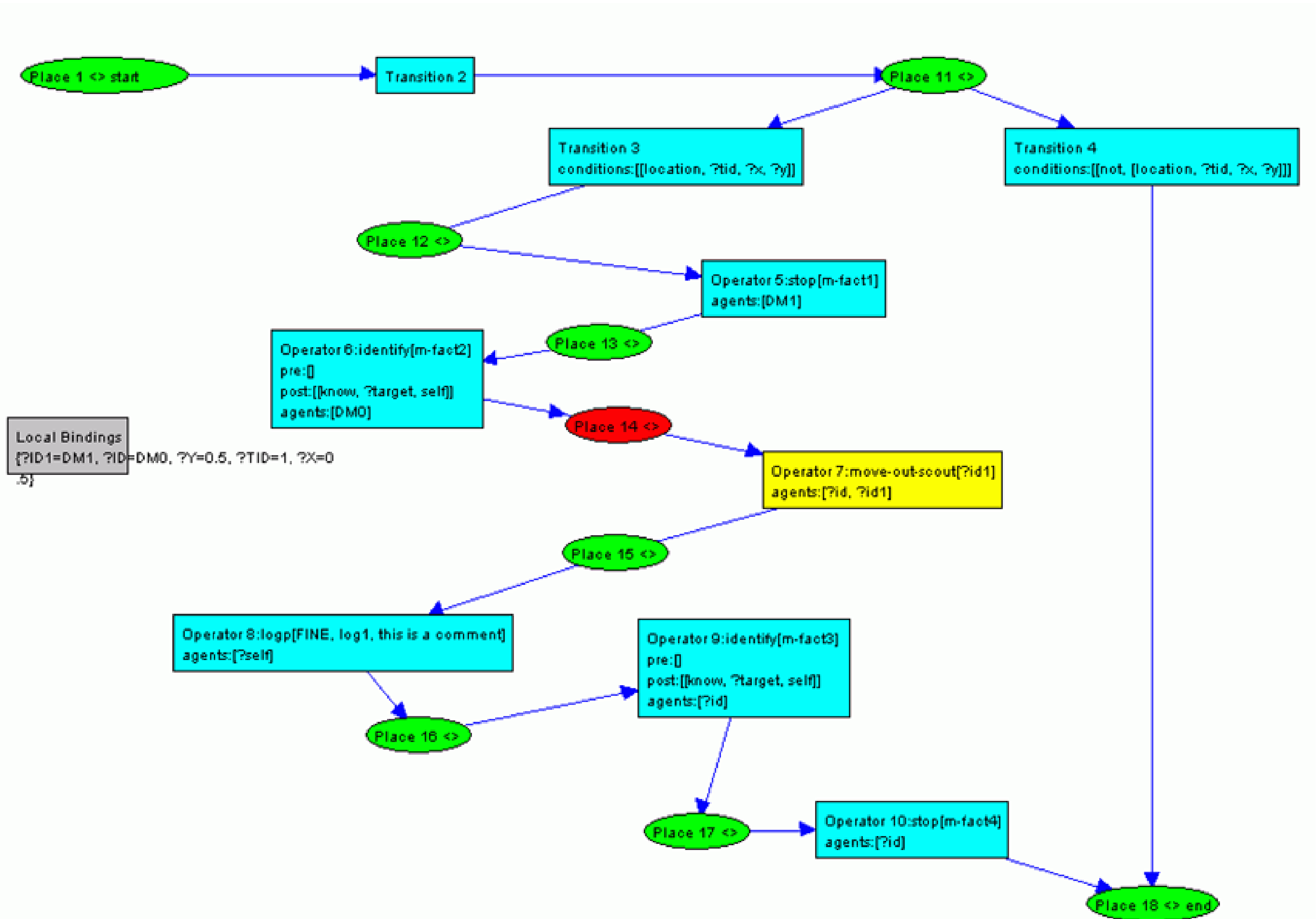

Figure 7: Example predicate transition net

Figure 7 is a snapshot of an example Predicate Transition Net using a test plan called **move-out**. A Predicate Transition Net generation algorithm constructs a transition for each operator in a team plan, connects them with control nodes, and links their inputs and outputs to appropriate belief nodes based on pre- and post-conditions. Since plans can be hierarchical, the sub-plans are expanded by calling the Predicate Transition Net generation algorithm recursively, and linked into the main Predicate Transition Net through control nodes down to the level that all the action nodes (transitions) ground out in operators. In the current implementation sub-plans are initially identified as operators until the sub-plans are expanded. By firing the tokens whenever steps are completed, agents can keep track of the progress of themselves and the team. Though not every agent is involved in or responsible for every step, this Predicate Transition -Net model of the overall team plan forms a common understanding of the team's goals and process, which agents use to determine how their individual actions fit together.

The test plan **move-out** is encoded in MALLET as listed below.

```
(plan move-out (?id ?id1)
  (effects (phase three))
  (process
    (if (cond (location ?tid ?x ?y))
      (seq
        (do ?id1 (stop m-fact1))
        (do senior (identify m-fact2))
              (do (?id ?id1) (move-out-scout ?id1))
              (logp FINE log1 "this is a comment")
        (do ?id (identify m-fact3))
        (do ?id (stop m-fact4))
      )
    )
  )
)
```

In the example plan **move-out**, Transition (Operator) 7 (move-out scout) is currently being executed. In this case the operator is a sub-plan that will be expanded. Place 11 marks a branch for an If construct. Places 12 through 17 mark a sequence of operators. Each plan is represented by a single Predicate Transition Net and each Predicate Transition Net has a single start and end node. Termination conditions exist if required to end plans.

A complete description of how the Predicate Transition Nets are constructed can be found in Jianwen Yin's dissertation (Yin, 2001).

4.2.4 Other Aspects of the CAST Software

The CAST architecture as developed for this dissertation and used in the test domain DDD, described in Section 6, includes visual displays and logging services to facilitate the development, testing, and execution of the software.

CAST provides the visual displays primarily through the central logging process, the CAST Logger. Each agent has a display associated with the agent in the CAST Logger. The agent display generates the visual view of the Predicate Transition Nets for real-time monitoring of each agent's progress. The agent display also allows for examination and manipulation of the inference engine and of the execution of the agent to a limited degree. The CAST Logger display may be used to pause, slow down, or speed up the agents' execution cycle. For development of CAST-ITT, the CAST Logger was embedded into the Coaching Agent. The CAST Logger loads a CAST configuration file (see Appendix B) during start up and uses this file as a reference for what agents are

supposed to connect to the CAST Logger via RMI. Figure 8 shows the trace from a team consisting of team members DM0 and DM1.

The logging facilities in CAST build upon the Java Logging API with modifications to the storage of logging data (see Appendix C). Logging is done at multiple levels, starting from the highest level of the MALLET plan execution via Predicate Transition Net execution, to CAST kernel execution, and down to the level of selected highlights of algorithm execution. Storage of logging is done in memory at each agent locally with a file written to hard storage at the end of the session. Selected logging statements (by level) are also sent via RMI to the CAST Logger. The CAST Logger stores and displays these logging statements.

Cast Logger

Monitor CAST Agents | Agent DM1 | Agent DM0

Current Agents

Pause All | Add Agent

View MALLET | Remove All

agents
DM0
cast3.dynamic.demo.Default
DM1
cast3.dynamic.demo.Default

Time Step = 500

Status of Agents

[0.801] DM1:paused = true
[0.821] DM1:CAST Agent starting
[0.821] DM1:initial plans = [[move-out, DM0, DM1]]
[0.961] DM0:paused = true
[0.991] DM0:CAST Agent starting
[1.001] DM0:initial plans = [[move-out, DM0, DM1]]
[5.378] DM1:entering plan move-out
[5.408] DM1:stop[m-fact1]
[5.458] DM0:entering plan move-out
[5.488] DM0:waiting on (move-out 5) by DM1
[5.718] DM1:0 received message from DM0[controlask,(move-out 5)]
[5.738] DM1:waiting on (move-out 6) by DM0
[5.939] DM0:1 received message from DM1[controltell,(move-out 5)]
[5.949] DM0:DM1 has completed (move-out 5)
[5.959] DM0:controltell node (move-out 5) in current plan true
[5.959] DM0:0 received message from DM1[controlask,(move-out 6)]
[5.989] DM0:DM0:IARG:[know, m-fact2, self] declare [DM1, DM0]
[6.009] DM0:entering plan pro-inform
[6.039] DM0:identify[m-fact2]
[6.219] DM1:0 received message from DM0[controltell,(move-out 6)]
[6.219] DM1:DM0 has completed (move-out 6)
[6.229] DM1:controltell node (move-out 6) in current plan true
[6.279] DM1:entering plan move-out-scout
[6.299] DM1:DM1:IARG:[know, ms-fact11, self] declare [DM1, DM0]
[6.339] DM1:entering plan pro-inform
[6.369] DM1:identify[ms-fact11]
[6.480] DM0:entering plan move-out-scout
[6.510] DM0:send[DM1, declare, (know m-fact2 self)]
[6.720] DM1:0 received message from DM0[declare,(know m-fact2 self)]
[6.740] DM1:stop[ms-fact12]
[6.940] DM0:completed plan pro-inform with [DM0]

Figure 8: CAST logger

4.3 Summary

In this section we have covered the underlying teamwork architecture, CAST, that CAST-ITT uses. CAST is important to CAST-ITT for three reasons:

- The CAST teamwork model is used by CAST-ITT.
- CAST agents are the virtual team members.
- The CAST-ITT monitor agents are derived from CAST agents.

5. TEAM TRAINING FRAMEWORK

The team training framework known as CAST-ITT is a knowledge-based approach to providing a methodology and software-based toolset to a training system developer to incorporate an automated ITS into their team training simulation domain. Understanding and developing a knowledge-based system for use in team training is a complex endeavor that requires knowledge across several disciplines such as computer science, psychology, and educational fields. The goal of the CAST-ITT framework (from here on addressed as the Framework) is to ease the development cycle by providing a generic team training framework that is adaptable to multiple team training domains.

The Framework contains several elements that are necessary for constructing an intelligent team training system, specifically; virtual team members, surrogates for human team members, individual and team assessment, and a coach. Our approach uses intelligent agents for each of the virtual team member and surrogate types. Interfaces are supported for generic and domain specific assessment modules. The coaching shell is provided to support interfaces for and execution of a coaching agent. In addition, the Framework contains a number of interface specifications that allow it to be used with various domain simulations, domain specific teamwork plans, and domain specific assessment modules. The Framework allows either software-based coaching agents or human coaches to be integrated into a system built in accord with the Framework.

The Framework is built upon our version of CAST, described in Section 4. CAST Agents are used for the virtual team members and extended CAST Agents act as

surrogates assisting human trainees. The Framework is designed to provide a limited amount of built-in generic team-oriented data collection.

To build a specific team training system using the Framework, a training system developer must provide a domain specific interface implementation, domain specific team plans, domain specific assessment modules (both individual and team) and, if desired, an intelligent agent acting as the coach building upon the coaching agent shell. A generic display interface is always present which, through introspection of the external domain specific modules, can present domain specific information to the human coach.

On the left-hand side of the diagram are grouped both the virtual team member agents, based on CAST, and the monitor agents, extended from CAST. These agents share a common command and sense interface. These interfaces are encapsulated in the generic domain interface. The domain specific interface is a realization of the generic domain interface for a particular domain. The domain specific interface will contain those commands that can be issued by virtual team members following plans encoded in MALLET. These commands form the basis of which commands a trainee will issue that the Framework will monitor. An instance of the overall Framework for a single trainee is shown in Figure 9.

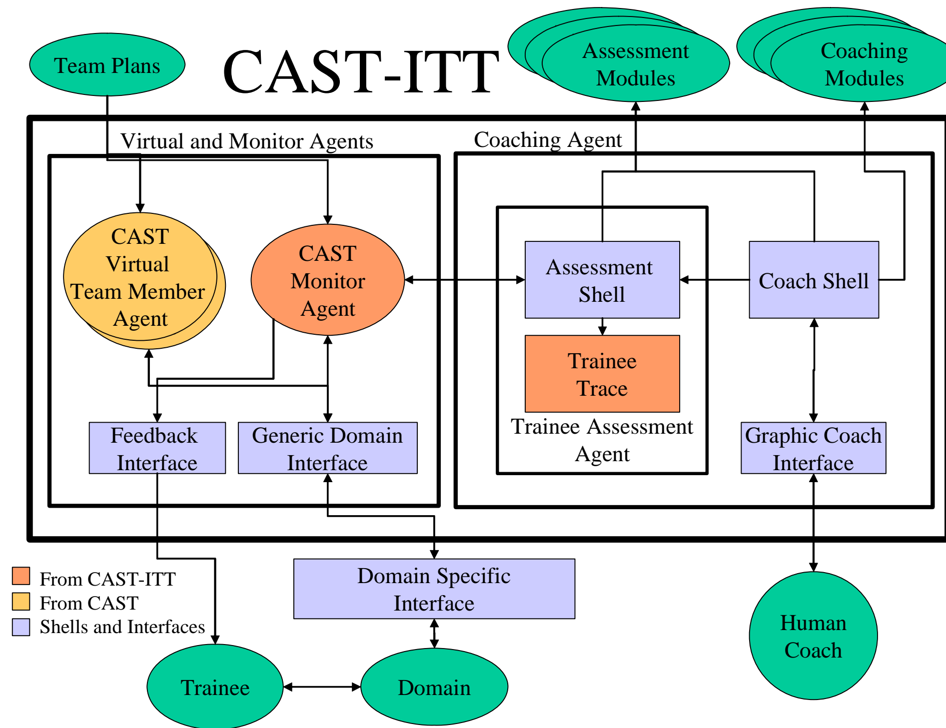

Figure 9: CAST-ITT framework

Using the Distributed Dynamic Decision making (DDD) system as an example (introduced in Section 2.2.2), the domain specific interface allows a CAST Agent to read the DDD client user interface and send commands (e.g., attack hostiles, or launch assets) to the DDD simulation. DDD is also discussed in Section 6 as a test domain that has been used with CAST-ITT.

At the top of the diagram in Figure 9, the team plans that are specific to a training domain are used by the virtual team member agents to perform as a member of the team and by a monitor agent to provide a model of the expected behavior of the human team members. In combination with a domain specific command and sense interface, the team

plans allow CAST Agents acting as virtual team members to act in that domain specific environment. The virtual team member agents may also communicate with each other and with the trainees during the training session. The team plans are written in the MALLET language, which is discussed in Section 4.1. As an example in DDD, the team plans provide the behavior of the CAST Agents in order that they function in the roles of a Decision Maker (DM) in the DDD simulation.

A monitor agent observes a single trainee's actions. The monitor agent provides the collected data of these domain actions for use by the assessment modules and by the coaching agent. The monitor agent also provides additional communication mechanisms for allowing the virtual team members and human trainees to interact with each other. For example, in a training domain such as DDD, those commands that have been implemented in the domain specific interface are monitored and a trainee's use of those commands is recorded. For training purposes, commands issued by a trainee should match those commands that exist in the domain specific interface to allow the Framework to support teamwork assessment in a generic manner.

The right hand side of Figure 9 refers to general support for coaching provided by the Framework. Data about domain activities by a trainee flows from a CAST Monitor Agent to the assessment interfaces. The monitored data is stored within the coaching shell (described below) for access by the assessment modules. Assessment is viewed at two levels, generic and domain specific. Each level is further divided into individual and team assessment. A human coach or a coaching agent is then able to use the assessments to give feedback to a trainee.

Generic assessment is based upon observations and analyses that can be made for any domain (e.g., amount of communication among team members) or that can be determined through analyses of the MALLET plans for specific activities in specific domains. It also includes assessment of what a trainee could (but might not) know from observations of the environment. Such generic assessment can be provided for any domain by capturing commands executed by a trainee and what the CAST Monitor Agent observes in the environment as viewable by a trainee. Please reference Section 5.4 for details on the monitoring interfaces and Section 5.5 for details on the generic performance assessment.

It is expected that every domain with which the Framework is used will have domain specific assessments that are needed. Obviously, these cannot be part of a generic framework for building team training systems. However, the Framework can, and does, provide mechanisms for incorporating domain specific assessment modules into systems that are built upon the Framework. The Framework specifies certain interfaces that must be accommodated by domain specific assessment modules. Once that has been done, the domain specific assessments will be performed and available to other parts of systems built using the Framework in the same manner as the generic assessment modules. For example, in the DDD domain, we have built domain specific assessment modules that determine both individual and team scores during the execution of a scenario. These scores can be accessed by the generic coach (described in Section 5.6) or by a domain specific coach built by the training system developer through generic framework interfaces.

Finally, the Framework provides a coaching shell for use by either a human coach or a coaching agent that a training system developer might create. In either case, the generic coaching interface provides access to the results of the assessment modules (both generic and domain dependent) by the use of introspection. The coaching shell is provided to give a basic platform (set of interfaces) on which domain specific coaching agents can be built. This basic platform gives access for the coaching agent to the assessment modules and their assessment results. The coaching shell is a central repository for monitored data collected about the trainees and the team for assessment and coaching.

Coaching feedback is based on interfaces available through the coaching shell and monitor agent to enable the coach to interact with a trainee. The first step for coaching support to a trainee is based on interfaces within the coach shell to access assessment results and build a coaching evaluation to be used as desired. The second step for the training system developer is the development of domain specific mechanisms that can be interfaced with the Framework through the feedback interface of the monitor agent. The feedback interface (described in Section 5.6) allows for the use of individual coaching user interfaces to be displayed to each trainee for providing domain specific feedback, e.g. reminders of missed objectives or expected behavior in a predetermined situation. A second mechanism for feedback to a trainee is through the manipulation of the virtual team members by the coaching agent/human coach.

The development approach has been to extend and embrace the current CAST architecture to support the additional ITT components. This approach is facilitated by the

open-ended nature of the current CAST architecture. Below in Figure 10 is a diagram of the major packages of the ITT part of CAST-ITT (the Framework). The Framework interfaces are marked in bold. The plus sign is used to mark sample implementations of specific interfaces to enable a basic level of generic assessment support in the Framework.

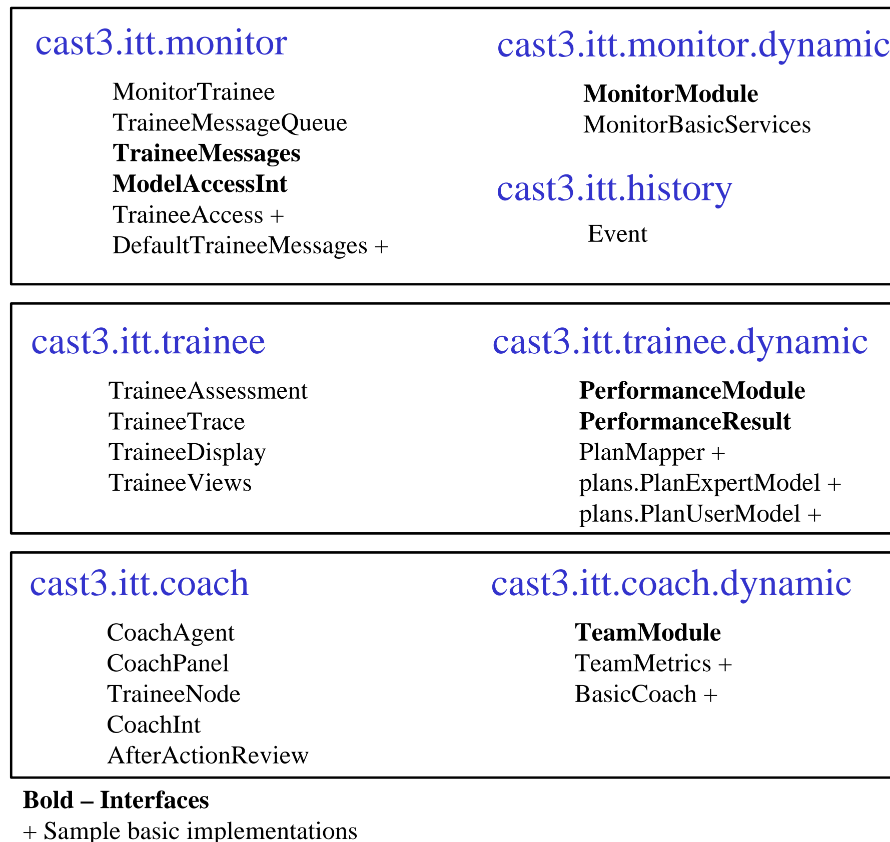

Figure 10: CAST-ITT packages

In the rest of this section, each of the issues discussed in Section 3 will be addressed. In order to do so, the Framework introduces a number of conventions and

requirements on the training system developer that must be observed in order for the Framework to usable as subsequently described. The sections in this section present these general requirements and then address the issues raised in Section 3. In particular the topics of this section are as follows:

- Integrating a specific domain into CAST-ITT
- Communications for teamwork
- Integrating a virtual team member with a simulation domain
- Monitoring trainees
- Performance assessment
- Coaching for teamwork

5.1 Integrating a Specific Domain into CAST-ITT

The highest level considerations in integrating a specific domain into the Framework are to create a correspondence between the elements addressable within MALLET and the elements that are executable within the domain. These elements fall into three categories, the execution of commands, the sensing of domain state, and the handling of communications (discussed in Section 5.2). As the Framework must provide mechanisms for handling all in a generic manner, it is necessary that some conventions be specified and observed by anyone using the Framework to build a specific training system.

The use of a system built upon the Framework by a training system developer is based on the existence of an expert team plan which explicates both what software

agents should do and what trainees are expected to do. Actions in MALLET plans are expressed, ultimately, in the form of MALLET operators. There are two forms of operators, internal and external, though they are not syntactically different. Internal operators may be any computation that can be performed without the issuance of explicit commands to the domain, e.g., calculating the distance between two points. External operators must be commands that are executable within the simulation domain.

Thus, the following requirement on the plans used with a domain must be observed:

1. There must be a set of operators in the MALLET plan that corresponds to executable operators in the domain.
2. Every operator in the plan that does not correspond to a command in the domain must be an internal operator, i.e., there must be an underlying software implementation of the operator.

Given this requirement, there are a number of issues that arise:

1. How does a framework know what commands are valid and executable by the domain?
2. How does a framework handle, in a generic manner, domain specific operators in MALLET plans and interact with the domain to cause execution of the corresponding command?
3. What level of error checking is provided by a framework?

Decisions in MALLET plans are based upon evaluation of logical predicates. All impact of information sensed from a domain is through predicates. Thus, all sensed

information from a domain must appear as a fact in the knowledge base underlying the CAST Agent. The issue then is maintaining the domain facts in the knowledge base. There are two basic approaches that one can take, pull the fact from the domain every time it is needed, or have the domain push the fact every time it changes or based upon some other condition, e.g., periodically. The decision on whether push or pull is best is often domain specific. Accordingly, the Framework provides mechanisms for implementing either and leaves the choice up to the training system developer.

A set of questions analogous to those for issuing commands arise with respect to sensing:

1. How does a framework know what sensing is provided by the domain?
2. How does a framework handle, in a generic manner, domain specific sensing?
3. What level of error checking is provided by a framework?

In the following subsections an overview is provided to the approach used by the Framework to address these two requirements of command execution and domain sensing.

The key to addressing the questions introduced above lies in the interfaces between the Virtual and Monitor Agents of the Framework and the domain highlighted in Figure 11 below. In order to understand the mechanisms used to answer the questions, we need to examine the Generic Domain Interface and Specific Domain Interface in more detail.

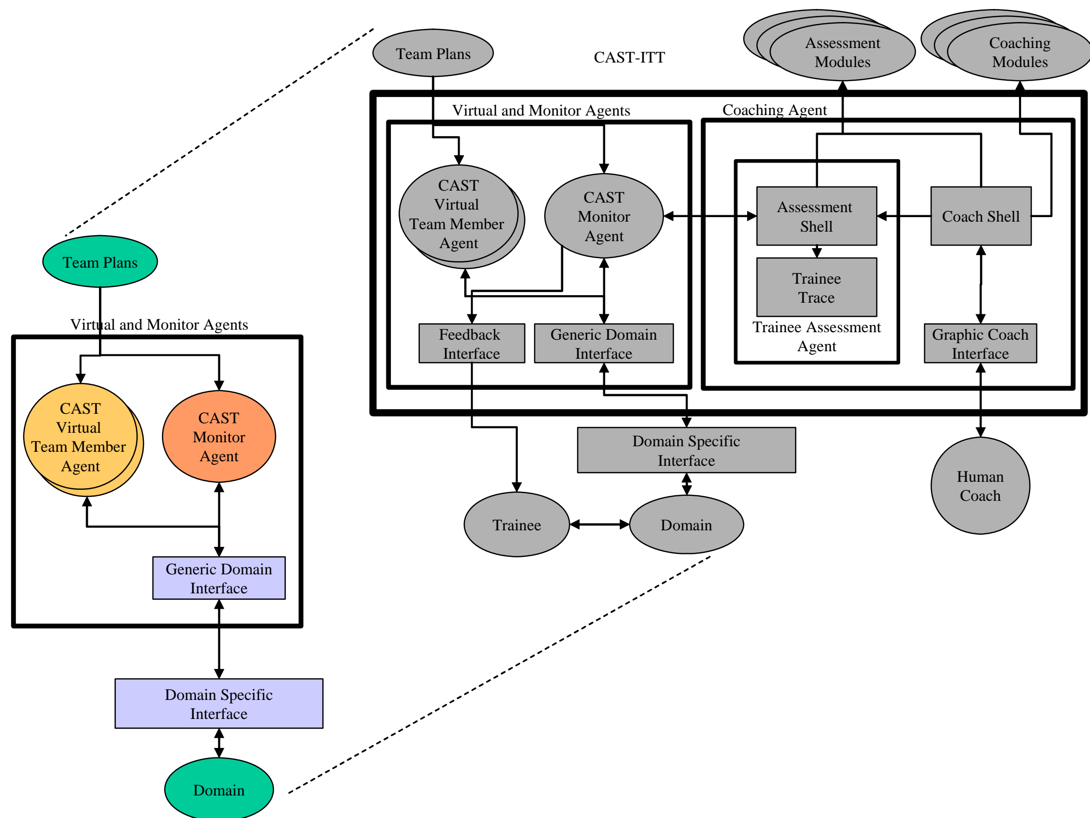

Figure 11: Integrating a specific domain

The Generic Domain Interface (which is an interface in the general sense) is comprised of two distinct Java classes, the ActorDomain abstract class and the CASTInterface interface. The Domain Specific Interface, which the training system developer must create, extends and completes the ActorDomain class and uses the CASTInterface interface to access needed elements in the agent (both Virtual Team Member Agent and CAST Monitor Agent). Only the ActorDomain class is used for

handling execution of commands, while both classes (interface and abstract) are used in the management of sensing.

Each of the different uses of these interfaces uses different, but sometimes overlapping, aspects of them. In order to make the presentation most easily understood, it is organized in terms of the utilization of the interfaces, and the only the portions used for a given purpose shown in the section being presented.

5.1.1 Command Execution

Command handling is generally quite straightforward from the perspective of the training system developer. From his/her perspective, he/she must simply extend the ActorDomain class, which for this view, appears as in Figure 12 below, to include domain specific methods which result in the execution of each of the domain commands.

```
public abstract class ActorDomain
{   /*   framework   supplied   utility   routines   and
     irrelevant abstract methods */
   /*public void command1( String arg1 … ){    } */
   …
   /*public void commandN( … ){    } */
}
```

Figure 12: ActorDomain abstract class

Each command must be specified with a unique name and has zero or more string values for arguments. The CAST kernel receives a string derived from a MALLET plan with the name of the MALLET operator (command) to be invoked. It follows that every external operator that appears in a MALLET plan must have a corresponding method in the implemented domain interface.

More specifically, the training system developer must extend this class by adding a method for every command the domain will recognize; the method must cause the command to execute. When a CAST Agent begins execution, it loads the class implementing this interface (the class is specified in a configuration file – See Appendix B). When a CAST Agent must execute an external operator, it uses introspection to find the proper method, dynamically creates the proper method call, and then invokes the method.

Figure 13 highlights the levels of execution within the Framework for handling the multiple requirements on the virtual team member in order to interact within a training domain. In the figure can be seen the flow of execution that comprises the Framework in representing plans, making decisions, issuing commands from plans, and executing the commands in the domain. The solid lines show direct execution (or compilation) from one level to the next. The dotted line shows the need for a one to one correspondence between actions in a domain and domain operator definitions in MALLET.

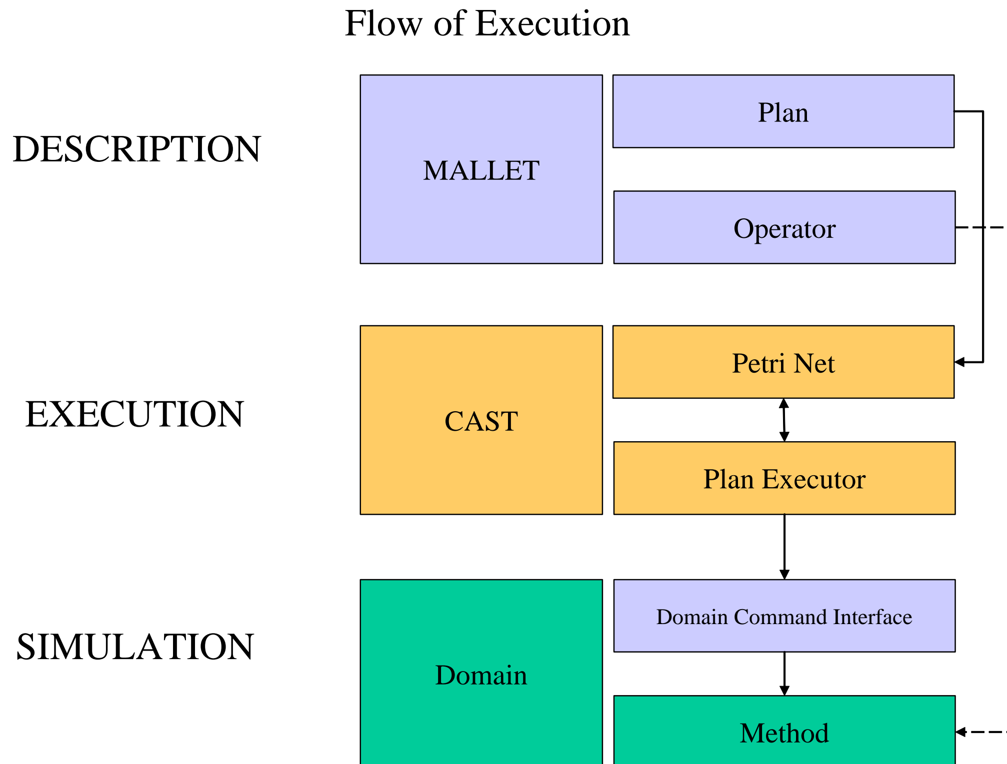

Figure 13: Flow of execution

Taken from the top of the diagram, the agent starts with a MALLET plan. The MALLET plans have been created by a domain developer for a specific domain. The MALLET plans consist of sub-plans, operators, and control constructs. The CAST Agent, acting in the role of the virtual team member, parses the MALLET plans to find the goals, roles, and other elements required for teamwork in the simulation domain.

For execution of selected MALLET plans, determined by goals, role, and responsibilities, the CAST Agent compiles the individual MALLET plans into Predicate Transition Nets. The Predicate Transition Nets provide an executable form of the team plans that when used in conjunction with other teamwork algorithms (detailed in Section

4) is used by the CAST Agent to control the flow of execution. The CAST Agent selects the plans as appropriate and executes the control flow of the selected plan until an operator is reached. That operator is sent as described above.

Error detection is built into the Framework. Error detection is based on two approaches; notification of errors when possible and fail safe execution. The complexity of the integration of the various parts of the Framework requires concise messages to be provided as soon as an error is encountered. Fail safe execution is based on the idea that the Framework will try to continue to run even when errors are being flagged.

Logging has been incorporated to the CAST architecture and is used for both finding errors and collecting performance metrics on the CAST agent architecture itself.

In CAST, the MALLET parser attempts as much as possible to find syntactic and semantic errors before run time to assist in the development of the domain specific MALLET plans. If such errors are found, the parser prints an error message and terminates.

As the domain specific implementation of the ActorDomain class is loaded by CAST, other checks are possible such as if the operators defined in the MALLET plans exist in the instantiated domain class. In the future other possible checks might exist such as testing of operators before session start, semantic checking of MALLET based on domain guidelines (rules), or extending logging to recording domain errors and/or needs.

5.1.2 Handling Sensing

The Framework provides a set of interfaces to allow the sensing of the domain environment by the individual CAST Agents. Each CAST Agent uses its own instance of the same set of interfaces. The purpose of these interfaces is to translate the current state of a domain simulation to an agent as sensed knowledge for use in that agent's knowledge base. Each fact about the state of the simulation domain that is viewable by the CAST Agent is represented as a predicate within the Framework. Such predicates are called domain predicates. Domain predicates are those predicates that are provided by a simulation domain and represent the state of the domain as viewed by an agent. Thus, the sensing problem becomes one of putting the relevant domain facts into a knowledge base accessible by the agent, i.e., conditions involving these facts can be expressed in MALLET and evaluated by the CAST kernel.

As noted earlier, the sensing process uses both the ActorDomain abstract class and the CASTInterface Java interface, and the process is significantly more complex. The reason for the additional complexity is driven by the desire to support either push or pull mechanisms for acquitting domain data, and the need to have efficient mechanisms within the Framework for managing the sensed data. In order to describe how the Framework should be utilized by a training system developer, it is necessary to describe, at least at a high level, some of the internal mechanisms created within the Framework.

Figure 14 below expands the high level Generic Domain Interface and Domain Specific Interface shown in Figure 11 and shows some detail that is internal to the Framework, but is central to understanding how to build the necessary sensing

mechanisms. In particular, the sensing process uses an extension to the agent's knowledge base called the DomainPredicate interface and an extension to the reasoning engine that allows predicates to be queried as either DomainPredicate or through the normal knowledge base. The introduction of DomainPredicate allows two key goals to be accomplished. First, it was desired to separate the mechanism for managing sensed information from the implementation of the reasoning engine. Second, it was desired to allow either push or pull mechanisms for managing sensed data. To achieve this, there is a thin layer below the reasoning engine (not shown in the figure and not relevant to the current discussion) that checks to see whether or not the name of a predicate matches one of the DomainPredicate objects. If there is a match, that domain predicate object is invoked to provide the fact; if not, the normal knowledge base is used.

Since both push and pull methods of sensed data acquisition are supported by the Framework, it is necessary both for the DomainPredicate objects to be invoked by the domain and for the DomainPredicate objects to be able to request updates of the domain. The former is made possible by the CASTInterface. This interface is implemented as part of the CAST kernel, and an instance of it created when an agent is initialized. CASTInterface provides a handle to the DomainEnv (see Figure 14), which contains the DomainPredicate objects; thus allowing the Domain Specific Interface to access the objects. As mentioned previously the Domain Specific Interface extends the ActorDomain abstract class.

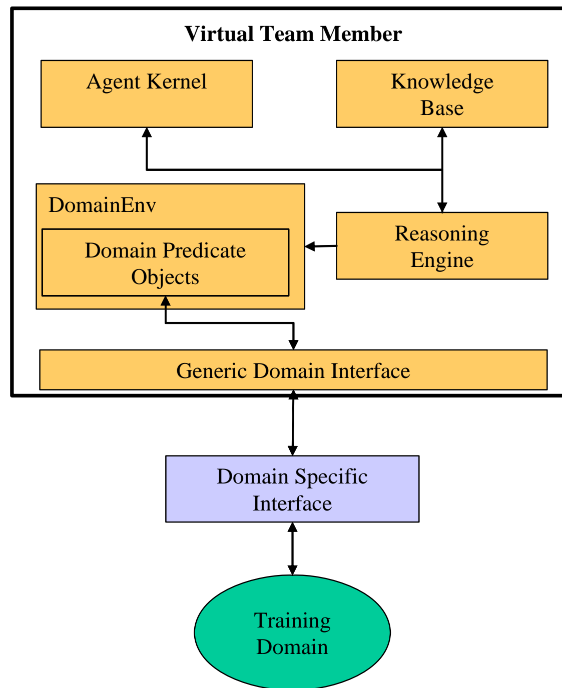

Figure 14: Integrating domain sensing into CAST-ITT

To allow the DomainPredicate objects to issue commands to the domain, the domain specific implementation of the ActorDomain abstract class must simply provide the methods needed to obtain sense information from the domain in addition to the action commands noted in the previous section.

A key aspect of the sensing mechanism is the creation of the DomainPredicate objects for use within the inference engine of the CAST Agent. As we have said, the domain specific extension to ActorDomain is granted access to the DomainEnv class through the CASTInterface, which, in turn, is accessible through a method provided by the ActorDomain class. The domain extension of the ActorDomain class is required to

create a predicate type for each sensing fact to be available for use by MALLET plans. To support creating these predicate types, a class named Predicate is provided by the Framework. The domain specific extension of ActorDomain must create an instance of the Predicate class, i.e., a DomainPredicate object, for each fact type. Each instance has two parameters; the fact type name and the number of variables for that fact type; the corresponding constructor profile is.

public Predicate(String name, int params)

The DomainPredicate object also contains a buffer for storing the domain facts of the corresponding type. This buffer is an object comprised of a set of unique facts for that fact type.

As we have said, the domain specific extension to ActorDomain is granted access to the DomainEnv class through the CASTInterface, which, in turn, is accessible through a method provided by the ActorDomain class. In order to make the DomainPredicate objects created as above accessible through DomainEnv (which is necessary because the may also be accessed via the reasoning engine), the addPredicate method in DomainEnv must be invoked to pass the object reference; this method takes a single parameter which is a handle to a DomainPredicate.

It is useful to consider the domain predicates in a bit more detail. Each predicate type can be described in the following manner.

(Type valueIdx+)+

where

- Type is a domain type name, e.g., asset, task

- Each valueIdx is a placeholder for a value or set of values associated with the type.

Essentially, one can view each of the entries as sort of a record specification with names for the record and the fields that describe the record. Multiple records may exist of the same type to describe individual domain facts within the domain.

For example, consider a predicate type definition for an asset in the DDD domain. It would have the form

(Asset id name type DM{#} x y vx vy power time_remaining state),

where Asset is the name of the DomainPredicate object, id is a parameter specifying a unique asset, name is a string name for the asset, type is the kind of asset (e.g., base, jet, etc.), DM{#} is an identifier for the owner of the asset, x, y, vx, vy are the x and y positions and speeds of the asset, respectively, power is the strength of the asset, remaining is the amount of remaining fuel, and state indicates the kind of activity the asset is engaged in. Obviously, all of these parameters have meaning only within the domain under consideration.

It must be remembered that a DomainPredicate is actually a type of predicate. There can be zero or multiple specific facts that satisfy the predicate. For example, the following are examples, which could all simultaneously be true, of the above predicate type.

(Asset A1 base1 BA DM1 0.4 0.3 0 0 0 0 launching)

(Asset A2 tank4 TK DM1 0.2 0.3 –0.1 0.2 5 33 attacking)

The buffer in the DomainPredicate for Asset would contain each of the above facts, which would provide information that the reasoning engine could retrieve if needed.

A further consequence of this representation is that the facts about a domain that can be referenced within a MALLET plan dealing with that domain must be of the format given by the form above. Thus, this form provides the link between the sensing implemented for a domain and the use of that information within MALLET plans.

The manner in which the predicates are updated depends upon the simulation domain. The choice of whether a push or a pull should be used depends upon the response time characteristics of the domain simulation and the ability of the domain simulation to support the respective modes.

For a pull mechanism the extended ActorDomain class has two choices. The first choice is to use a timer method to periodically invoke a method that pulls the state knowledge from the domain. The second choice is to enable the CAST kernel to do a pull before its next decision cycle. The ActorDomain class does the appropriate call and update before notifying the CAST Agent using the notifyAgent method that the next decision cycle is to be made.

For a push mechanism the domain class uses an event handler to accept the push from the domain and load the pushed state information into the DomainEnv object.

The update method for the domain predicates is part of the creation of the ActorDomain class by the training system developer. In this method is where the developer must decide what predicates to have represented from the domain. The domain developer must also create the translation of the domain sense data (in a

database or other accessible form) and translate that data into a string form in the associated predicate type definition.

In examining error checking for the domain predicates the Framework looks for a correspondence between domain predicates used in MALLET plans and that such predicates are provided by the domain.

5.2 Communications for Teamwork

Communications, and in particular human communications, is a vast area for research. Humans use communication in order to support teamwork. Unfortunately human communications is a complicated affair with the burdens of natural language recognition and the many kinds of communications which extend beyond strictly verbal, e.g., gestures, facial expressions, etc. However this should not stop the training system developer through assuming that handling team communications is something to simply postpone until a more complete solution exists.

As discussed in Section 3.2, the Framework does not try to solve the issue of what team communications is relevant. The question of what communications are required becomes an issue for the training system developer. Instead the Framework focuses on providing mechanisms for enabling communications to be a part of the Framework.

Communication in teamwork serves the needs of coordination and information exchange. To serve these two needs we have both explicit and implicit communications as defined in Section 3.2. How a team training framework supports these two types of communications is based on the framework's underlying model of teamwork.

For our Framework we use CAST as the underlying agent architecture and to support our model of teamwork. CAST expects that communication is an integral part of teamwork. CAST agents send and receive messages from other CAST agents. CAST algorithms do analysis of communications needs of team members based upon MALLET plans (Xu et al., 2003). When human trainees are incorporated into the team, communications between virtual team member agents and those human trainees need to be considered. Therefore, handling communications with humans is a critical part of the Framework. Most simulation domains provide communication mechanisms among trainees, and CAST is designed to take advantage of such mechanisms when present, but also provides hooks by which a variety of communication mechanisms can be added by the training system developer if desired.

For communications the following questions must be addressed.

- How does a framework utilize domain communication mechanisms (if present) to realize inter-agent communication?
- How does a framework handle human/virtual team member communications?
- How does a framework incorporate human to human communications
- How are implicit (observation-based) communications handled by a framework?

The Framework is not a complete or final solution to handling communications within teamwork. Instead the Framework provides the understructure that the training system developer can use to address communication training needs in his/her domain.

While we realize natural language translation is ultimately desired, as discussed in Section 3.2, it is not a part of the Framework. Rather, the Framework allows a variety of textual and visual communication mechanisms to be implemented.

The Framework assumes that, at the lowest level, all explicit communication is handled by the sending and receiving of strings. Such sending/receiving mechanisms are expected to be asynchronous in their execution. Strings are the textual representation of information to be exchanged. Different purposes are achieved via interpretation of the strings.

A training system developer has two levels of responsibility with respect to this. The first is the implementation of low level send and receive methods. This must be done through the implementation of the domain specific implementation of ActorDomain. More specifically, the ActorDomain specifications include the send and receive abstract methods shown in Figure 15 below. Once these methods have been implemented (discussed in detail below), the fundamental communication among agents is in place.

It is worth noting that on the receive side, the training system developer must reference the queueMessage method in CASTInterface (interface to the CAST agent) when implementing the receiveMessage method in the extension to ActorDomain. The queueMessage method places messages into a queue for agent kernel processing. The training system developer must invoke this method, passing it the received message, as part of the implementation of the receive method of ActorDomain.

```
public abstract class ActorDomain
{ /* framework supplied utility routines and irrelevant abstract
      methods */
  public abstract void sendMessage (String receiver, String
      message) ;
  public abstract void receiveMessage (String message);
}
```

```
public interface CASTInterface
{ /* access to CAST agent for domain */
      public void queueMessage (String message);
      /** other access methods to CAST agent **/
}
```

Figure 15: Send/receive methods

The second responsibility of the training system developer is to interpret the message and utilize the message strings as appropriate for the domain and specific training system being developed. What has to be done depends upon the extent to which the domain supplied an inter-trainee textual communication system, the extent to which this was used in the implementation of the send/receive methods and the range of communication mechanisms to/from trainees. The development might be as simple as passing text strings to the domain for it to display to the trainees. Or, it might understand the message formatting used with the agent to agent communication

(discussed below) and strip off unrelated messages. Or, it might interpret the string for display in some specific visual form (see the TAP in Section 6.1.2 for an example). A more sophisticated domain might support conversion of text strings into speech generation, in which case the developer might have to do some simple parsing and calling of speech generation routines. Correspondingly, limited speech recognition systems exist in which a limited domain specific grammar could be used to interpret trainee generated speech and convert it into strings to be sent. In any event, this aspect of the communication is domain specific and the Framework simply provides the hooks by which any such mechanisms can be incorporated into a system built utilizing the Framework.

In order for a developer to utilize the Framework properly, it is necessary to understand the internal formatting used for the basic messages communicated that allow various parts of the Framework to understand the messages. It is also useful to understand how the Framework addresses the more important issues discussed in Section 3, as that provides a rationale for the choice of formats. It is most useful to discuss these in terms of the various combinations of explicit agent and human communication, i.e.

1. Agent to agent, (addressing the first question above)
2. Agent to human, (addressing the second question above)
3. Human to agent, (addressing the second question above)
4. Human to human, (addressing the third question above)

and the implicit communications, i.e.

5. Observation-based (addressing the fourth question above).

5.2.1 Agent to Agent Communication

The approach taken by the Framework in handling agent to agent communications is based on an underlying model of teamwork as embodied in MALLET and implemented in CAST. In this section we only discuss agent to agent communication between virtual team members. Other communications may exist between the coaching agent and the monitoring interfaces of the Framework but are not a part of this section.

In explaining agent to agent communication using CAST, it is important to recognize that MALLET does not require any form of explicit directed agent to agent communication, e.g., a write or read placed explicitly in a plan as a step for the agent to invoke. This does not mean that a developer is prohibited from creating MALLET operators for writing and reading messages. However, the Framework will only recognize them as operators and call whatever methods implements them, which will be completely distinct from the internal agent to agent communication discussed here.

The agent to agent communication discussed here arises from proactive communication in which an agent decides on its own (not explicitly part of a plan) to send or ask for information to/from another agent, or from the need to coordinate team progress through a team plan. Accordingly, CAST agents use a set of specific message performatives to communicate information coordination needs. These messages use a KQML format (Finin et al., 1997) and have seven types of performatives.

The seven CAST message performatives are as follows.

- **ask** – a predicate queried from the asking agent's KB
- **reply** – predicate that is asserted into the asking agent's KB

- **assert** – predicate that is asserted into the receiving agent's KB
- **retract** – predicate that is retracted from the receiving agent's KB
- **controltell** – current operator step in the active plan of the sending agent
- **controlask** – query on if an operator step has been completed in the receiving agent's active plan
- **unachieved** – failed active plan of the sending agent

Ask, **reply**, **assert**, and **retract** are all standard KQML messages for allowing agents to communicate state information and exchange agent beliefs. **Controltell** and **controlask** are used with CAST to coordinate completion of steps within active team plans. **Controltell** is for an agent to inform other agents of task completion in active plans. **Controlask** is for querying other agents as to their progress through active plans. **Unachieved** is used by an agent if an active plan that agent was executing was terminated or failed.

The first four performatives are all used as part of the proactive information exchange that the MALLET model of teamwork supports. The last three performatives are issued automatically by CAST Agents acting as virtual team members to support the "shared mental model" of each CAST agent.

Controltell and **controlask** each have two parameters, the plan name and the node number. The plan name will be a plan being executing by the agent. The node number will correspond to an operator in that plan. Since each agent has a copy of the original MALLET plans and translates those plans using the same algorithm into Predicate Transitions Nets each node number in a specific plan will match across the

individual agents. An agent executing a plan may use **Controltell** to tell another agent also participating in that plan of each completed operator in the plan. An agent may query another agent about the completion status of individual nodes (operators) in a plan by using the **controlask** message performative. **Unachieved** is only used if a plan terminates. Termination is a special precondition on a MALLET plan.

These messages are sent and received by CAST agents using the domain implementation of the sendMessage method and the use of the receiveMessage method described in the introduction of this section. Using these mechanisms a CAST agent will send and receive messages using the seven performatives during the execution of MALLET plans. A training system developer does not need to be concerned about the use of these performatives for agent to agent communication, as the Framework manages this automatically. However, with human/agent communication, the training system developer needs to understand and use at least some of these performatives. Moreover, the training system developer will have to create and interpret these KQML messages for some of the domain functionality that must be added.

When viewed in string form, a KQML message has the syntax as shown in Figure 16.

```
KQML        = (Performative Sender Receiver Message
            Number)
Performative   = (<PERFORMATIVE>
            <ask|reply|assert|retract|controltell|
            controlask|unachieved>)
Sender      = (<SENDER> Name)
Receiver    = (<RECEIVER> Name)
Message     = (<MESSAGE> (Predicate))
Number      = (NUMBER Integer)
```

Figure 16: KQML syntax

The Predicate in the message body is used as either a fact or a query (**ask** or **controlask**).

To facilitate creating and interpreting these messages, the Framework provides a KQML class. A KQML object is be used to build a message by using the getText (part of every object) method to produce a string representation of that message. The KQML class offers a number of constructors for constructing the message in the appropriate format for use by CAST agents. Each non-terminal in the KQML BNF is a name value pair that can be individually set by the training system developer. Each object has getter and setter methods for each of these components. Objects can be constructed by creating a empty object and then adding the components via the setter methods, by passing the constructor a string for the KQML message or by copying from one object to a new object. Normally, a sender would create an empty object and fill in the components via the setter methods and a receiver would use the received string to construct the object followed by getter calls to obtain the individual components.

The methods (getters/setters) of the KQML class are:

- Text – creates object from string or produces string representation of object
- Performative – sets/gets one of seven types listed above
- Message – sets/gets body of message in predicate form
- Sender – sets/gets name of sender
- Receiver – sets/gets name of receiver
- Number – sets/gets unique incrementing id for message by sending agent (automatically done by agent when sending)

In Figure 17 are three examples of constructing new messages. The first example is created by using the string that would typically be sent from one agent to another. In CAST this constructor is used by receiving agents. The second example is a copy constructor. The third example is object construction by individual setter methods. This construction method of a KQML message is used in CAST by the sending agent.

```
KQML object construction from a string
KQML K1 = new KQML("((performative assert) (message
((agent tim ready))) (sender tim) (receiver bob))");

KQML object construction from another KQML object
KQML K2 = new KQML(K1.getKQML());

KQML object construction by individual methods
KQML K1 = new KQML();
K1.setSender("tim");
K1.setReceiver("bob");
K1.setMessage("((agent tim ready))");
K1.setPerfomative("assert");
```

Figure 17: KQML objects

In summary, from the perspective of the training system developer, there is nothing that needs be done beyond providing an implementation of the send and receive methods of the ActorDomain class. The Framework handles all of the agent/agent communication by sending/receiving KQML messages via these methods. However, the training system developer needs to either create or interpret at least some of these messages in domain specific ways for handling the agent/human interaction.

5.2.2 Agent to Human Communication

Ultimately agent to human communications must support the training to be conducted in a specific domain. This requires that virtual team members participate in the teamwork with human trainees. To support the teamwork the virtual team members must communicate as required for the training domain. An agent will send two types of messages. Coordination messages based around the use of team plans in MALLET, and information messages composed of facts exchanged based on the IARG algorithm described in Section 4.2.2.

From the point of view of the Framework, the issue is then what interfaces the Framework provides the training system developer to take agent communications intended for a human trainee/Monitor Agent pair[2]. The Monitor Agent actually receives all such messages and decides which to handle alone and which to translate into a form presentable to a trainee. Every such message will be of the form described in the previous section.

---

[2] Recall that there is a Monitor Agent for every human trainee to assist in the communication between the human and other agents, as well as to maintain information that may be useful for coaching.

The first step in handling the agent to human communication is to decide what is to be done with the performatives identified above, since a virtual team member may send (or expect to receive) any of these kinds of messages to (from) the human because it has no knowledge that it is communicating with a human. On the surface, it would seem that some of them would have no meaning to a human, e.g., **controltell,** and with others, it may be questionable, e.g., how easily could a trainee understand an "(assign task-123 DM2)" message with performative **assert**. However, the Monitor Agent (see Section 5.4 for more details) has need of messages such as **controltell** even if the human does not.

In particular, a Monitor Agent tracks the progress of the team through a team plan and maintains a knowledge base corresponding to the team member role the human is playing. Therefore the Monitor Agent actually receives all agent to agent communication intended for the role (position) for which a trainee is receiving training. Accordingly, when **controltell** messages are received, unbeknownst to the human, the Monitor Agent updates its version of the Predicate Transition Net (discussed in Section 4.2.3). When **assert** or **retract** performatives are received the values are placed in or removed from this knowledge base. This updated knowledge base is useful for performance assessment and is discussed in depth in Section 5.4.1.

More specifically, communication from an agent to a human trainee is actually received by a Monitor Agent. The Framework implementation of Monitor Agent will automatically take care of maintaining the Predicate Transition Net and knowledge base the human trainee. In particular, it handles the following performatives automatically:

- **assert**
- **retract**
- **controltell**

In addition, the Monitor Agent makes all incoming message available for display to the human trainee. It does this through an interface named TraineeMessages for which the training system developer must build an implementation to handle agent to human messages. For each of the seven performatives the TraineeMessages interface provides the specification of a method whose purpose is to convert the corresponding KQML message into a string and then pass that string to the receive method of the ActorDomain extension. It is up to the training system developer to decide whether or not and how to display each of the performatives to the human trainee. Typically, some methods, e.g., **controltell**, may do nothing.

```
public interface TraineeMessages {
      public void ask(String sender, String message);
      public void reply(String sender, String message);
      public void declare(String sender, String message);
      public void retract(String sender, String message);
      public void controltell(String sender,String message);
      public void controlask(String sender, String message);
      public void unachieve(String sender, String message);
}
```

Figure 18: TraineeMessages interface

The Framework does not specify the manner of presentation to the human trainee. That is domain specific and must be designed as part of the design of a specific training system. In Figure 18 is shown the TraineeMessages interface. The Framework has a default implementation called DefaultTraineeMessages that provides a default Java message dialog that displays each of the seven performative types. The training system developer may replace DefaultTraineeMessages by specifying a domain specific implementation class. This implementation class is loaded by the Monitor Agent using the TRAINEEMESSAGE parameter in the agent configuration file.

Within the replacement class the training system developer starts the task of displaying one of the TraineeMessages methods by creating a KQML object using the string from the message. Once constructed the KQML object has getter methods for the counter, sender, receiver, performative, and message. The counter is unique to each message sent by an agent. This may be useful for tracking messages for performance assessment support by the training system developer. The message is a predicate encoded as either a fact or a query. The training system developer may easily access the message fields and build an appropriate domain specific presentation of the message content to a trainee. The implemented TraineeMessages interface is invoked whenever a message is received.

We note that the Framework gives the training system developer great freedom in generating the display to the human. For example, if speech synthesis were available, it could generate voice outputs for some messages. As example would be that when a proactive tell sends information needed to fulfill a precondition on an operator that the

human trainee should do, the information could be turned into a voice representation (e.g. "A level 3 threat just entered your zone."). Or, more mundanely, it could utilize existing domain communication mechanisms for message passing to transmit the information.

5.2.3 Human to Agent Communication

Human to agent communications can be the most difficult form of communication to handle as it exposes a number of complex issues involving communications through domain mechanisms, the possibility of natural language recognition support, domain specific restricted grammars, and the integration of the Framework communications mechanisms with the trainees' communications.

Many training simulation domains provide methods for trainee to trainee communication. For example, DDD provides an email like mechanism that allows trainees to send messages to each other. Also, many training simulation domains allow verbal communication among trainees. The Framework assumes that some such mechanisms are available in the domain. This results in a need to place certain restrictions on the use of these domain mechanisms.

The most fundamental requirement made by the Framework is that any human generated communication must be reducible to a form that can be understood by machine. Such reduced communications must be in one of two forms: 1) one of the performatives introduced above, or 2) a form that can be understood and acted upon by domain specific MALLET operators or code placed in the domain specific extension to ActorDomain. The former requires the use of a structured syntax (could be either verbal

with speech recognition or textual) that can be parsed into the proper forms. The latter allows the Framework to support anything for which the training system developer can create interpreting routines. We discuss the latter situation first, as it is mostly outside the realm of the Framework, but does impose some restrictions on the training system developer.

To handle human generated information of the second form, the first requirement is that the training system developer must be able to reduce the input to a bit string. The system developer may choose to do this either at the destination, if the domain supports transmission of the information in its original form, or at the point of origin. In either case there are three critical things that must be done with the string at the receiving end: 1) it must be obtained from the domain in some way, 2) it must be made available to the agent for use, and 3) it must be possible to trigger the execution of an MALLET operator as a consequence of the receipt of the string. There are multiple ways in which each of these might be accomplished by a training system developer; we discuss two ways of approaching these that might be used, as they will illustrate the necessary interactions with the Framework.

As a first example, we consider the case in which the string is constrained to be in accordance with some restricted grammar from whose sentences predicates to be inserted into the agent's knowledge base can be readily composed and in which the information is sent through mechanisms supported by the domain, e.g., the e-mail facility of DDD (Kleinman et al., 1996), or treated as sensed information. In this case, an instance of a Predicate object (see Section 5.1.2) must be created corresponding to the

receipt of the string. The training system developer must implement a method to parse the string received so that the appropriate translation can be created from the received sentence and then create a predicate from the message and insert it (discussed in Section 5.1.2 on handling sensing) in the agent's knowledge base.

As a second example, consider the case in which the information is transmitted from a sender to a receiving agent via special write/read routines, and it is desired to simply invoke a domain specific MALLET operator, passing the received string as a parameter.

In this case, the developer must include, in his/her extension to ActorDomain, a method to convert the information to a bit string (unless, of course, the information is transmitted in string form, e.g. by using something like the email facility of DDD (Kleinman et al., 1996)).

In order to transmit this bit string to an agent, the training system developer must create in the extension to ActorDomain: 1) some form of write method that will cause the input string to be sent to a designated agent, and 2) some form of read method that can receive these special messages and take appropriate action with them. These methods will then be identified as operators in MALLET plans for use by the agent. Each of these operators, though, must have methods added to the domain specific extension to ActorDomain to support them. Some out-of-Framework send/receive mechanism must be built, with the write operator invoking the send and the receive invoking the read. It is on the read side that the training system developer must primarily interact with support provided by the Framework.

First, one must consider how a read method could be invoked from a MALLET plan. Typically, in the MALLET plan this would be done by having a precondition that waits on the presence of the input string. The training system developer would need to do two things in the receive routine that is added to the ActorDomain extension; 1) assert the received bit string into the knowledge base as a predicate (e.g., x=<bitString>), and 2) set the read operator precondition to true by asserting a predicate to the agent's knowledge base. This should be done when the message is received by the domain specific extension to the ActorDomain. Once the precondition is true, the read method may then process the bit string. This sequence of events (receive message, store the bit string in the knowledge base, set the precondition to true, and invocation of the associated read operator) allows an agent to also do additional processing through MALLET, such as enacting effects on associated operators and plans. The benefit of this approach is that the agent is able to incorporate the act of reading the communication into its own reasoning activities. This sequence of events is shown below in Figure 19.

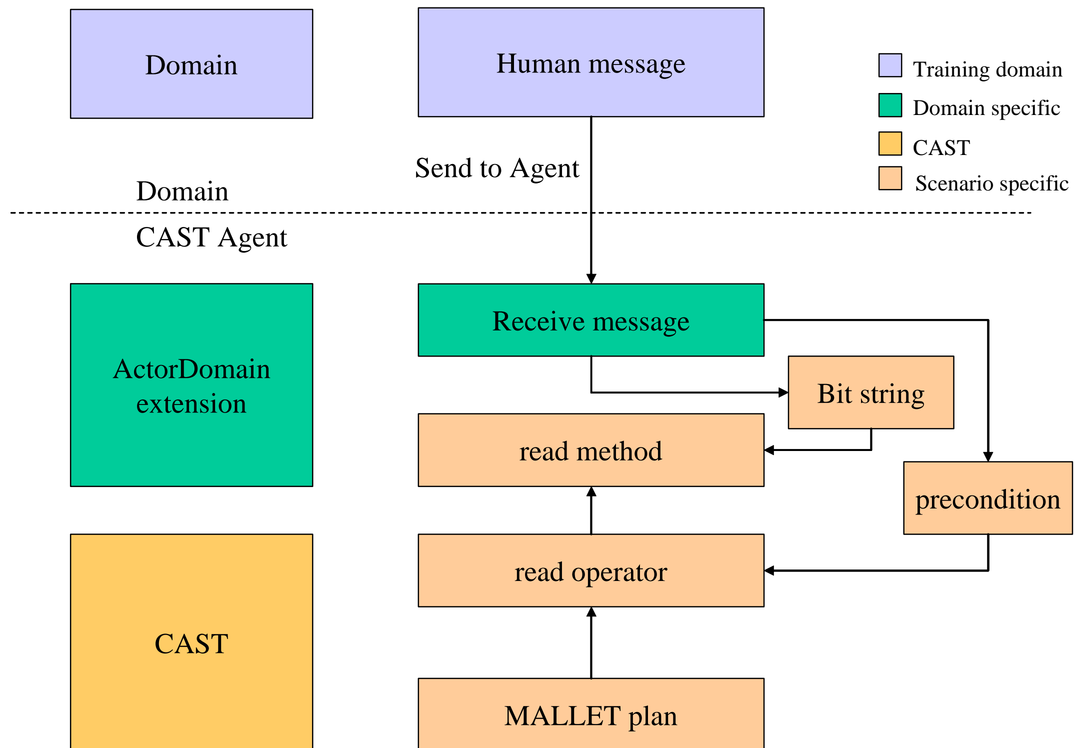

Figure 19: Second form of human to agent communications

In the case of the first form (use of one of the KQML performatives), the central requirement of the Framework is that every communication from a human to an agent be transformed into one of the seven performatives described in Section 5.2.1. The reduction of the human communication to one of these performatives is domain and training systems implementation specific and not specified by the Framework. An overview of the first form of human to agent communications is illustrated in Figure 20.

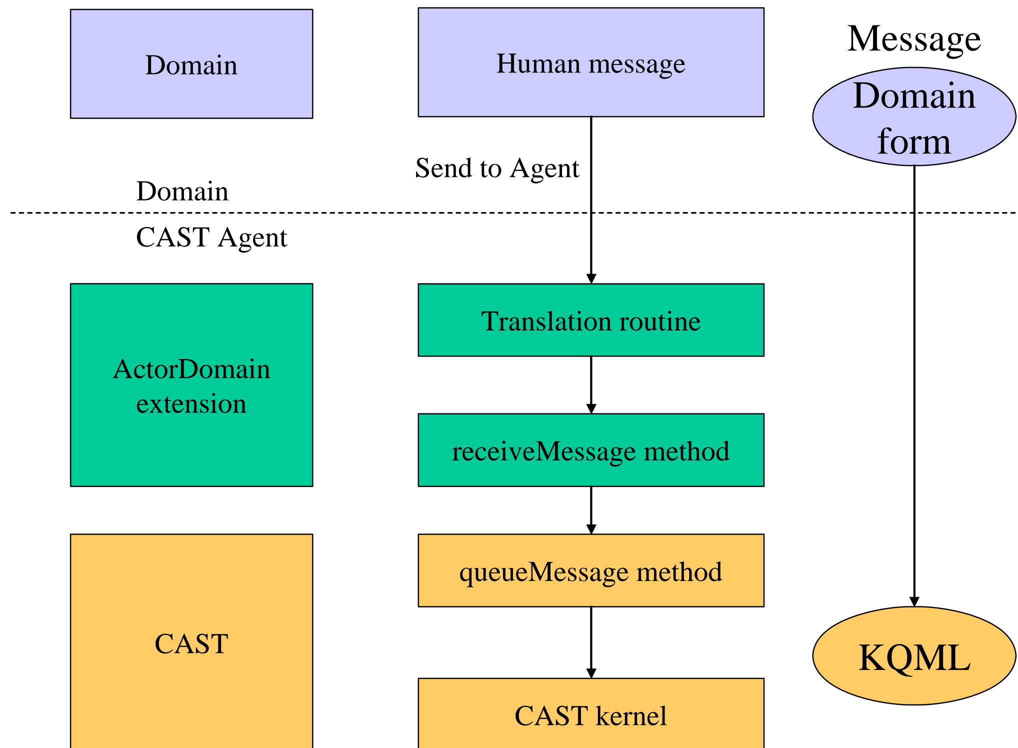

Figure 20: First form of human to agent communications

In order to accomplish the delivery of a message once the proper format has been created, the training system developer has to extend his/her receiveMessage method to provide the required performatives to the CAST agent. For the message to be delivered to an agent the required fields in the KQML object are sender, receiver, message, and performative. For each message the training system developer wishes to convey to the CAST agent the message needs to be encoded using an object instance of the KQML class and delivered with the queueMessage method. In order to structure the message for use by the Framework the KQML class is provided to format each message (described in Section 5.2.1).

Having described what a training system developer would have to do to deliver message to agents using the Framework, it is useful to describe a bit more the use of communication by a human trainee. Much of the work in our research group has been centered on various notions of proactive communication, especially the idea of providing information when it is needed without being asked. Providing needed information proactively is one form of helping behavior that is considered very important. The **assert** and **retract** performatives would typically be used for this purpose. Our concepts of proactive information exchange also include asking for information when needed from the agent or agents most likely to have it. The **ask** and **reply** performatives would be used for the respective ends of this kind of communication.

The first four performatives; **ask**, **reply**, **assert**, **retract** are the most straightforward performatives to implement. These messages either query the agent's knowledge base or change the state of that knowledge. Once formatted, they can be inserted directly into the message queue and the agent will receive the knowledge. However, the developer need only handle the performatives that a trainee in his/her domain might generate. The decision on what, when and to whom to issue them, however, is a complex decision process and is discussed in greater detail in Section 5.4 as part of the discussion on the Monitor Agent.

The last three performatives; **controltell** and **controlask**, and **unachieved** are different in that they are used for coordination within the MALLET team plans. They may be issued automatically by a Monitor Agent (see Section 5.4), rather than a trainee.

A final approach the training system developer may choose is to instead design the operations of the CAST agents in such a way that the agents do not depend on direct human communications in order to function. This requires that agent MALLET plans be encoded with minimal or no team plan interactions with human trainees. Essentially each agent does its own thing in the performance of its duties.

5.2.4 Human to Human Communications

Human to human communications offer the potential to expand what can be monitored by the Framework. Given that a team is being trained there is the possibility that more than one human trainee is present in the training session. Additionally the design of the training session may require a level of expertise that the training system developer cannot encode into a virtual team member and therefore a human domain expert is used as part of the team training session. Both choices are likely to result in human to human communications.

A training systems designer can design a training system from a variety of viewpoints regarding human to human communication. Training systems have been built and tested that allowed such communication among trainees, with no consideration of that communication in the underlying agent-based ITT system except in the form of post-session interviews (Hollingshead & McGrath, 1995). In most cases it should be possible to record, external to the Framework, all such communications. However, it may also be desirable for the Framework to be aware of human to human communication. For example, performance assessment modules might want to consider the amount of such communication. Or, if such communication is restricted to a form

that can be understood by machine, it may be desirable to record the communication for both assessment and coaching purposes.

Use of the Framework for building a team training system does not require that the builder do anything regarding human/human communication. That is, human/human communication can either be completely disallowed or allowed and disregarded from the training perspective. On the other hand, if desired, the Framework does provide minimal generic support for handling certain forms of human/human communication in a manner that allows some simple generic performance evaluation (e.g., number of messages) to be performed and also allows domain specific performance assessment modules to make use of the communication (see Section 5.5.1 for assessment support).

How the training system developer plans to handle the communications in regards to the Framework will affect the communication's impact on the virtual team members, Monitor Agent, and assessment and coaching support. Although there are undoubtedly many methods by which a training system developer could incorporate human/human communications on top of the Framework, we outline the general concept we had in mind when developing the Framework.

Support of human/human communication is based on the assumptions that the input can be converted to a bit stream and that individual items of communication can be distinguished if desired. For instance, one might have a push-to-talk type of digital voice recorder that could make bit streams available to the training system at discrete points in time. Everything the Framework does is in terms of this bit stream and assumes no knowledge of the contents of the stream. Essentially, the Framework

provides support for capturing the bit stream, storing it, annotating it with the time at which it was acquired, and limited assessment.

Human-human communication implies a source and a destination. Identifying the destination is domain dependent and might involve complex issues such as speech recognition (though it might be as simple as reading a specific communication channel used). Thus, the Framework takes the perspective of only recording the communication at the source.

For every message a human generates, the Framework has to record this message for the message for use in performance assessment. In order to do the above the training system developer must to the following steps.

1. Implement a domain record method in the extension of ActorDomain
2. Store the recorded bit stream if desired for use by other components of the Framework

The basic storage mechanism available in the Framework is based on the PerformanceResult abstract class (described in Section 5.5.2). The training system developer must extend the PerformanceResult abstract class (which requires giving it a unique name) to provide an object that can store a successive of bit streams corresponding to the segments of human to human communication. The record method that must be added to ActorDomain must then invoke a method of an object of the extended class, passing the recorded bit stream as a parameter.

This storage is done within the Monitor Agent. Therefore, to allow access of the storage of the bit stream by other assessment modules, the training system developer

must add a dynamic module to the Monitor Agent (using the abstract class MonitorModule) that provides an RMI method for accessing the communication bit streams by performance modules (which might be remote – hence the need for an RMI method).. Refer to Section 5.5 for a more in-depth discussion on how to retrieve the stored data and invoke either generic or domain specific assessment modules on the data.

The flow of communications as implemented through the Framework is shown in Figure 21. The capture layer illustrates the level at which the communications can be included into the Framework. While the most obvious form of human to human communication (that is what is shown in the figure) that could be included is voice, other forms such as gestures and gesture recognition are not precluded.

In summary, human to human communications provide both a challenge and an opportunity for the training system developer in which due consideration must be taken of the pedagogy that is desired and the technology that is available to achieve the training objectives. The most complete solution would be to incorporate some type of natural language recognition for use in parsing human messages into a form that is ultimately acceptable to an agent. However, natural language recognition is outside the scope of the Framework.

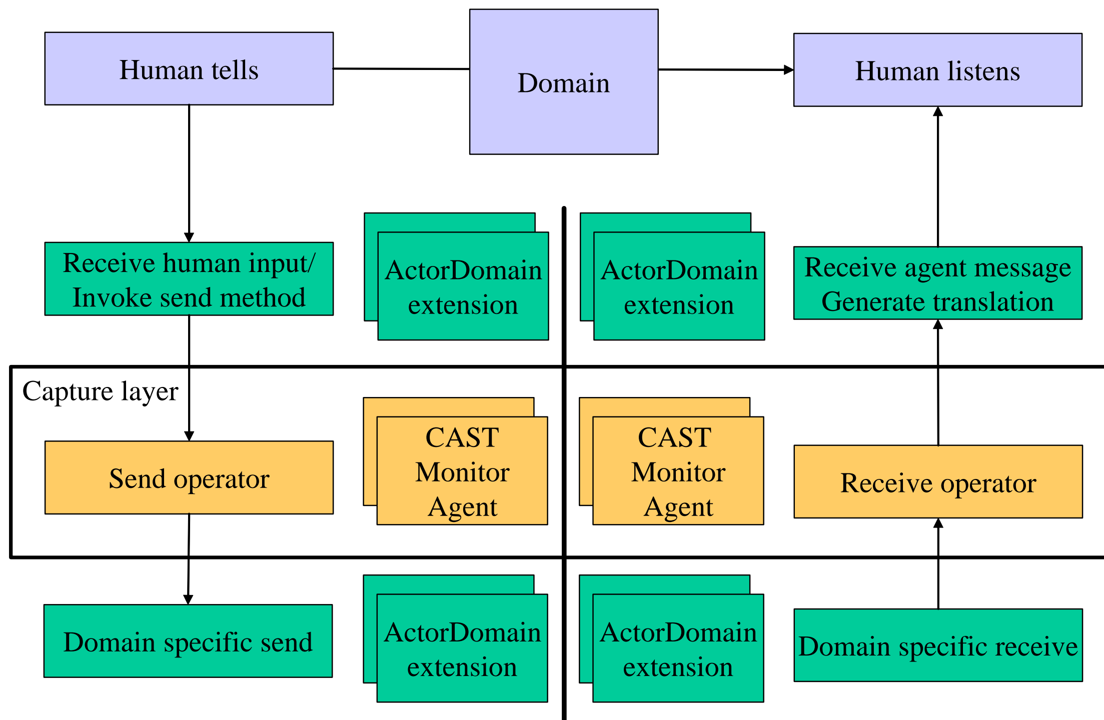

Figure 21: Flow of communications

5.2.5 Observation-based Communications

As opposed to explicit team member communications, a different approach is used for implicit communications. Implicit communications occur through observation instead of through direct message exchanges. An example is a list of incoming tracks tagged to show what track is assigned to which team member. The sending end of the communication would be accomplished by an agent or human making an assignment, while the receiving end is just the observation of this list by the receiving agent or human.

Unlike the explicit send/receive messages in the previous subsections, implicit communications occur as a result of an action by one team member that does not fall into the category of explicit communications (send/receive). Such an action causes a change in the state of knowledge available to other team members. The other team members are able to observe such state changes and add the new information to their knowledge of the domain.

For CAST agents, new information from such actions is treated as a change in the state in the domain (possibly an augmentation of the original simulation domain state), and communications become a query by the agent on its domain knowledge when needing to use the information. This state information is accessed through the sense interface (described in Section 5.1.2). An example of this process is shown below in Figure 22.

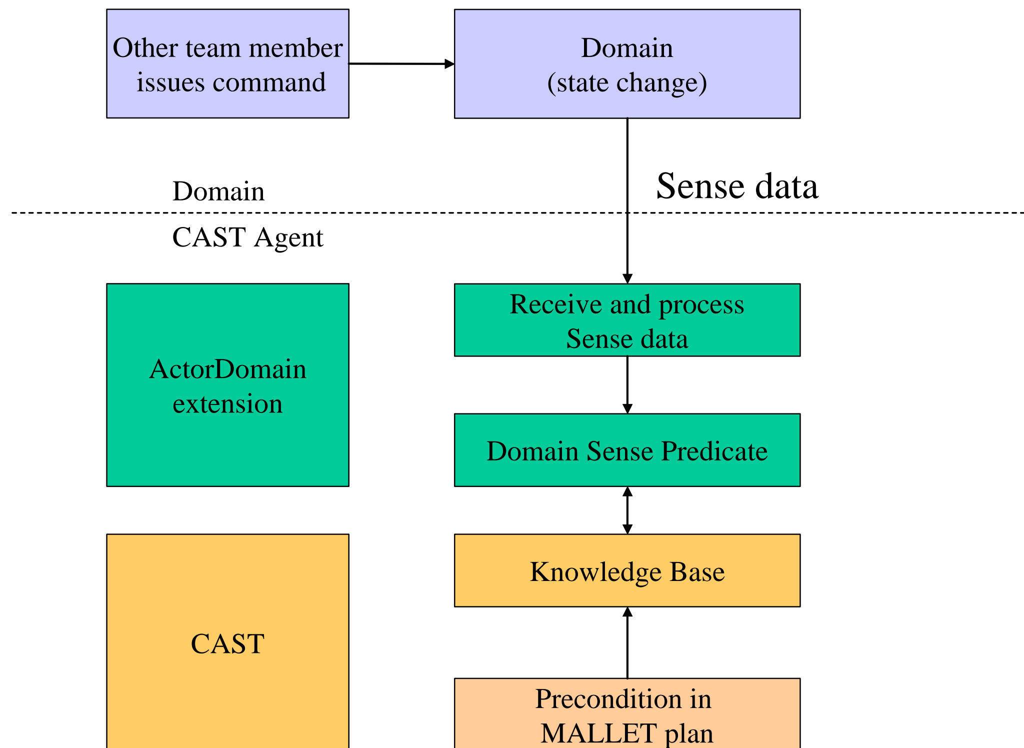

Figure 22: Observation-based communications

Implicit communications generated within the system in effect become domain knowledge that is accessible to the virtual team member. However, such knowledge must still follow the rules of observability in that the virtual team members only observes what a human of that team position could observe. Such rules of observability are determined by the training system developer.

The implications of observation-based communications on the training system developer can be summarized in four points. 1) Using a MALLET operator to change the domain state by a CAST agent. 2) Use of the domain sense mechanism to observe the state change. 3) The human trainee must also be able to generate the state change (i.e.

the MALLET operator is a command in the domain). 4) A domain display must also exist so the human can observer the state change. From the perspective of a training system developer, all this means is that the domain specific extensions for sensing described in Section 5.1.2 must include ensure that all necessary domain states can be observed are met. See the TAP in Section 6.1.2 for an example.

5.3 Integrating a Virtual Team Member with a Simulation Domain

We are training individuals to perform as part of a team. Therefore, the team is modeled as individuals and not as an aggregate. Individual trainees practice as a team member. Other team members that are required for training are represented either by experts or by virtual team members.

The Framework allows such virtual team members to be modeled by the use of intelligent agents. These intelligent agents are referred to as Virtual Team Member Agents (or Virtual Agents). The Virtual Agent is a CAST Agent that replaces those team members which are modeled by the training system developer using MALLET.

CAST Agents acting as team members impose two categories of requirements, the timely behavior by the agents and the specification of a virtual team member, its domain behavior, and its interface to the domain simulation. Timely behavior breaks down into three issues; synchronization of time between Framework (CAST Agent) and domain, timely response by an agent, and performance limited response by an agent.

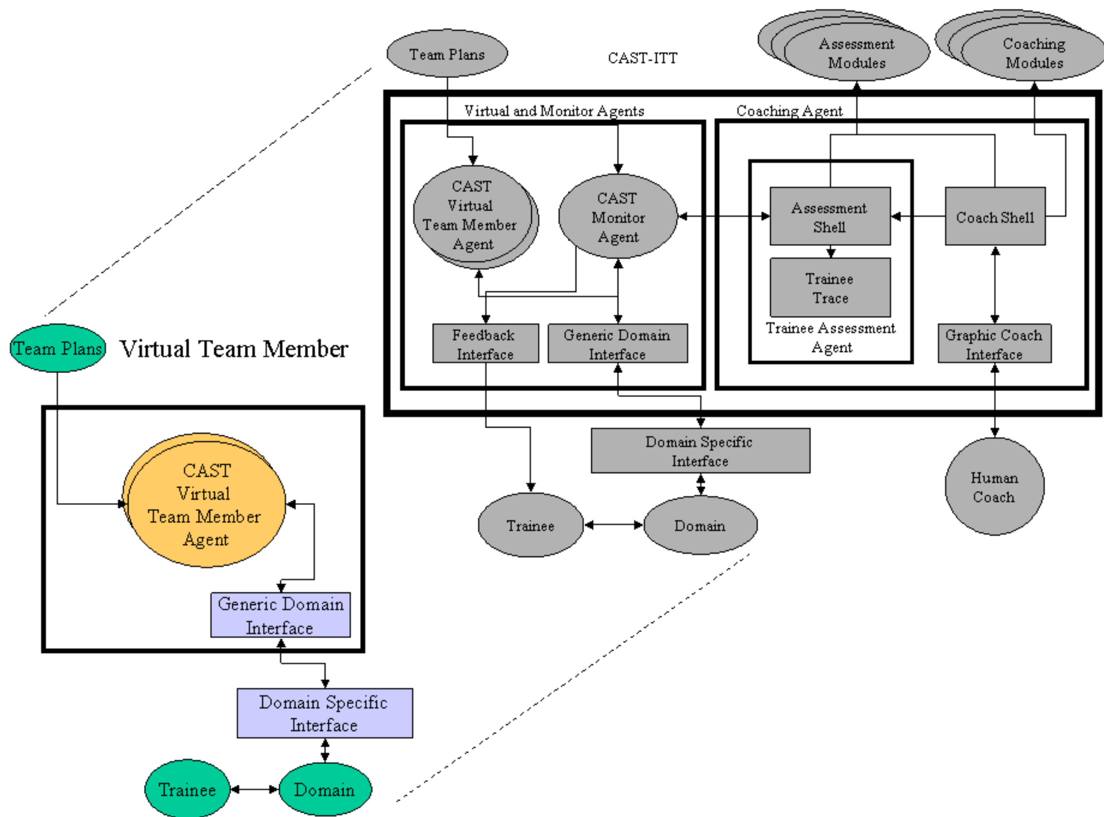

Figure 23: Virtual team member in CAST-ITT

To put the discussions that follow in proper context, in Figure 23 we illustrate the major elements involved. Starting from the top are the domain team plans encoded in MALLET. Below this is the Virtual Agent which is an instance of CAST. The virtual team member acts on a sense, decide, act paradigm using the model of teamwork as described by MALLET/CAST. The Virtual Agent calls upon the generic domain interface (ActorDomain abstract class) previously described. The training system developer will have extended this class to provide their domain specific extension. For

each domain, MALLET plans must be created by the training system developer to engage the MALLET/CAST virtual team members in the training scenarios.

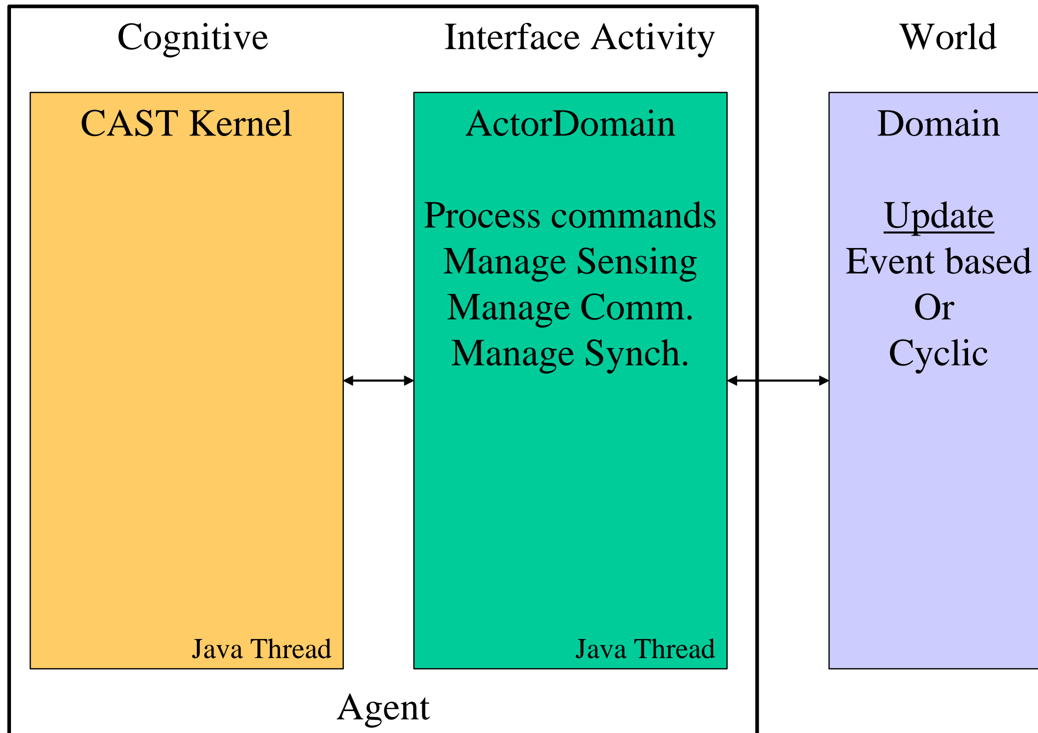

Figure 24: Synchronizing agent to world

We divide the Framework's view of time into three levels; cognitive, interface, and world. Figure 24 illustrates the three levels of timing required by the Framework. For the Framework the first two levels, cognitive and interface, are managed through CAST by the use of two synchronized threads of execution. The cognitive level is handled within a thread for each CAST Agent to support the performance and response

time requirements of the agent. The synchronization (time and data) between the agent and the domain is handled in an interface thread. Each thread (agent and interface) executes and then switches control back to the other thread for each agent decision cycle. The domain is expected to have its own system for managing time.

5.3.1 Maintaining Synchronicity between the Domain Simulation and the Framework

Though the Framework manages time by dividing time management into the three levels identified above, the principal issues are twofold, matching the time performance of the agent with that of a human in the domain, and synchronization of exchanges between the agent and the domain. This section is divided into two parts, corresponding to these two issues.

5.3.1.1 Response Time Issues

The requirement for the Framework is to achieve an adequate match between the time flow of events in the domain simulation and the time flow in the cognitive activities of the agent. In an absolute sense, this cannot be achieved because arbitrarily complex plans can be written and the response time of the reasoning engine made large enough to exceed the needed response time of any given system. Thus, it is not possible to guarantee a timely response on the part of the agent. In a practical sense, however, it is possible to achieve agent response times that match those of humans for the reasonably complex systems we have tested. On the other hand, one must also be concerned that a

computer agent might act significantly faster than a human could, and it should not do so when it is playing the role of a human.

There are three principal aspects of the agent that impact reaction times: 1) time to acquire data needed from the domain for evaluating preconditions, 2) time required by the reasoning engine to actually evaluate preconditions and make a decision, and 3) the time required by CAST to perform its execution cycles (see Section 4.2) and issue a command. The first two limit the domain response times that can be handled without noticeable effect on performance, while the third is a reverse consideration that needs to ensure that the agent does not respond significantly faster than a human.

The time to acquire data from the domain is based on the access time is determined by the domain software and the underlying hardware and network. It is a requirement on the domain that these be negligible with respect to the software response times in the agent.

In regards to the update of data from domain, the significant issue is how updates are done. If event driven then at the next decision cycle the expectation is that the data is from the latest update available. If updated in a cyclic manner, then the agent must be data synchronized with the ActorDomain extension so that such an update is complete before any data is used in a decision cycle, i.e. avoiding the read/write synchronization problem. In order to ensure the agent does not read the data before it has been updated (whether cyclic or event based) the DomainEnv object that is used to transfer the domain data is a synchronized object so that simultaneous writes and reads are not possible. Writes are done in the ActorDomain abstract class which is a thread that must be started

by the training system developer in their extension of the class. Once the thread is running and having met the sensing data requirements in Section 5.1.2, the training system developer has done what is required for synchronizing the use of the sense data for each decision cycle.

For the second issue since one can write arbitrarily complex preconditions and it is thus possible to create preconditions whose evaluation times exceed any specified time, care must be taken in the development of the plans that are used. Experience has shown that with moderately slow computers by today's standard (800 MHz Pentiums in our experiments) human equivalent response times can be readily achieved for moderately complex domains and plans. It is the training system developer's responsibility to verify that the time required for evaluation of preconditions (including any needed sensing) can be accomplished sufficiently rapidly. To aid in this, two mechanisms have been developed to help reduce the time spent evaluating preconditions, one to allow a more efficient implementation of the evaluation, and one to avoid redundant evaluations. Plymale has developed a mechanism that allows a Java implementation of precondition evaluation to be incorporated into the system in place of the normal JARE search methods, thus making precondition evaluation much more efficient (Plymale, 2004).

The second mechanism reduces unnecessary evaluations of preconditions based on the decoupling of the agent's execution from the simulation domain's execution. A variable named TIMESTAMP specifies a timeout period in milliseconds before a query will be reevaluated. The value of TIMESTAMP should be set to the update rate of the

domain data from the simulation domain. For example, if the domain only updates its data once per second, as in DDD, there is no reason to evaluate preconditions on domain data for each agent decision cycle (which may be at a faster rate).

For the third issue, the actual operation of the agent execution is quite complex. Humans can do some things in parallel (e.g., talk and walk), but not others (e.g., input multiple commands via a keyboard simultaneously). Thus, CAST and/or MALLET must provide means for emulating this behavior; this is normally done through use of the **seq** and **par** constructs. To handle the **par** situation, when a command is sent to the domain for the domain simulation to execute, CAST does not wait for the domain to complete this action. Instead, the agent proceeds to consider its next actions immediately after issuing a command. Thus, from the agent's perspective, the actual execution of a command takes almost no time. However, one must then be concerned with the agent responding too rapidly, and thus, it is sometimes necessary to slow the agent down to human scale response times.

Two mechanisms used to prevent a software agent from running faster than a human is to use a timer thread in which the agent kernel executes. In the first, the period of the agent execution cycle (see Section 4.2.1) is regulated by the use of the variable TIMESTEP. The TIMESTEP variable specifies a minimum execution cycle interval in milliseconds. This cycle is intended to help mimic a human response time by slowing down the agent execution. If agent computational time for a cycle exceeds the TIMESTEP, then the next cycle immediately restarts.

In the second, delays may be inserted into MALLET plans to enable the agent's reaction times in performing steps within sub-plans to be paced to average human performance for that training domain. As a reference implementation (used in DDD), a dispatcher plan was created that lets the training system developer insert time delays within the plans themselves (as determined by human response times) that are executed before the sending of commands. The dispatcher plan is covered in more depth in Section 5.3.2.1 as part of the discussion on designing virtual team members.

Both TIMESTAMP and TIMESTEP are set through the use of the agent's start up configuration file (see Appendix B).

5.3.1.2 Synchronization between Agent and Domain

The Framework is designed to work with discrete time inputs rather than continuous inputs. Discrete time-based domains can advance time using two different approaches, event driven or cyclic simulations. Event driven simulations are systems in which events (activities or actions) are posted to the simulation as they occur. Cyclic simulation systems have a system-wide clock that advances at discrete times. At each cycle, changes for that cycle are posted within the simulation.

While the Framework uses a cyclic execution model for managing its agent activities, it supports both event driven and cyclical domain simulations. There are two aspects to time synchronization between the agent and the domain. The first is having a common sense of time for purposes of reasoning about time, and the second is ensuring that the timing of interactions between the agent and the domain occur in a manner consistent with the timing of interactions between a human and the domain.

To address the first issue, all reasoning about time is based either on an abstract view of time or is done on a relative time basis. In the former case, a decision cycle was completed after sense, decide, act sequence in a domain. This approach worked for domains such as Wumpus world with an artificial time step. Alternatively, reasoning about time can be based upon relative times. It is possible to read the current time from the domain. All reasoning about time is based upon multiple readings of time from the domain and computations on the time differences with respect to time differences within the agent. Obviously, there is a time resolution with which such computations can be done. We assume that this resolution is significantly less that the relative times for which decisions must be made. Such has been the case in systems tested to date, e.g., Revised Space Fortress (which increments time in 46 millisecond increments) or DDD (which increments time in one second increments).

To ensure that the timing of interactions between the agent and the domain occur in a manner consistent with the timing of interactions between a human and the domain, it is sufficient to ensure that the agent be able to make decisions and issue commands at least as fast as a human, and force its interactions with the domain to occur with timing consistent with that of a human. The TIMESTAMP, TIMESTEP and dispatcher mechanisms mentioned in the previous section provide a basis for the training systems developer to achieve this condition.

The Framework requires that all interactions between the agent and the domain are based on transmission of coherent blocks of data rather than on explicitly trying to match the sense of time in the domain and the agent. In order for this idea to work, it is

necessary that the response time of the agent be fast enough to keep up with the domain and that it have some mechanism for not running too fast for the domain.

The mechanism used to synchronize data acquisition with the domain is to decouple the acquisition from the basic agent cycle and use an intermediate synchronized object to hold the sensed data. This is, in fact, the role that DomainEnv (see Section 5.1.2), plays. When domain data is needed in an execution cycle, DomainEnv is invoked to obtain it. In the case of the data pull mode (see Section 5.1.2), if the data is not current, DomainEnv will, in turn, pull the data from the domain, and will not return until the data is ready. The pull mechanism for obtaining the data obtains and entire block and uses a synchronized update to DomainEnv. In the push mode, it is the responsibility of the domain to push data in coherent blocks, after which the synchronized update to DomainEnv ensures that the data is synchronized. The training system developer need to nothing more than has already been described (see Section 5.1.2 on handling sensing).

5.3.2 Intelligent Agent as Virtual Team Member

Given a team training environment the training system developer must answer the requirement for additional team member support and interactions as needed to train one or more trainees within the team. The Framework helps to resolve the team training requirement through the use of CAST as the underlying intelligent agent architecture of the virtual team member. CAST is covered in Section 4 as to what it provides in answering the requirements raised in Section 3.3. Specifically CAST brings to the Framework the following capabilities.

- A model of teamwork as embodied in MALLET and the IARG algorithm
- A decision making cycle acting on a sense, decide, act paradigm
- Interfaces to extend the above capabilities

MALLET is the plan language used to describe the behaviors of the team members in CAST. In the following subsection how the training system developer should proceed in creating a virtual team member will be discussed.

5.3.2.1 Implications for Domain with using MALLET

We have covered in the previous sections how to integrate the Framework into the domain. However, this is just the physical aspect of the Framework acting as a virtual team member. A virtual team member must be able to act within a specific training domain. To do this CAST has an intention structure that executes the virtual team member's individual roles for the provided MALLET team plans. Such domain specific MALLET plans have to be provided by the training system developer.

A more detailed discussion of how to use MALLET in the construction of team plans can be found in the documentation of the MALLET language (Fan et al., 2006). However, the important distinction between that discussion and what is described in this section is the role of the team member. In the original development of CAST the members of a team consisted of all agents. This expectation of an all agent team drives a simple view of the development of such MALLET plans such that the focus is on the behavior of the team members and not how such behavior might compare to a human member playing that same role.

The key point of this section is that in order to emulate human roles the training system developer must have a cognitive task analysis (CTA) of the domain. In a team training environment the interactions of the human trainees and the virtual team members are the central consideration in programming the behavior of the CAST agents. It is important that the agents behave as humans and not exhibit what are intended to be optimal behaviors that are unattainable by or seem counter-intuitive to humans (Cao et al., 2004). The CTA drives the development of the virtual team member. First, the CTA in combination with scenario selection determines what team members need to be emulated. Second, what capabilities the CAST agent must have, based on the CTA, drive the development of the domain specific ActorDomain extension in the choice of the domain sense predicates and domain commands available to the CAST agent. Third, the MALLET plans the training system developer must create, based on the CTA, drive the responses and actions of the CAST agent to match those of its human counterpart.

While the Framework assumes that the training system developer will write MALLET plans in accordance with a CTA, the Framework does provide a set of skeleton MALLET plans (executor and dispatcher) for use by the training system developer. The executor plan is based on the proposition that it will be the top level plan and will be expected to execute several sub plans in parallel. One of those sub plans (called dispatcher) is used to regulate the execution of the agent's domain actions. The dispatcher plan places delays and forces a single thread of execution of domain actions by an agent for actions that cannot be executed in parallel. Other sub plans are used to

model the agent's behavior and responses in reaction to state changes in the domain and internal knowledge base changes enacted by the CAST Agent.

The use of this structure for the MALLET plans allows the training system developer to model the behavior of the CAST agent to match the expected response times of a human team member. Figure 25 illustrates the high level organization of the template plans.

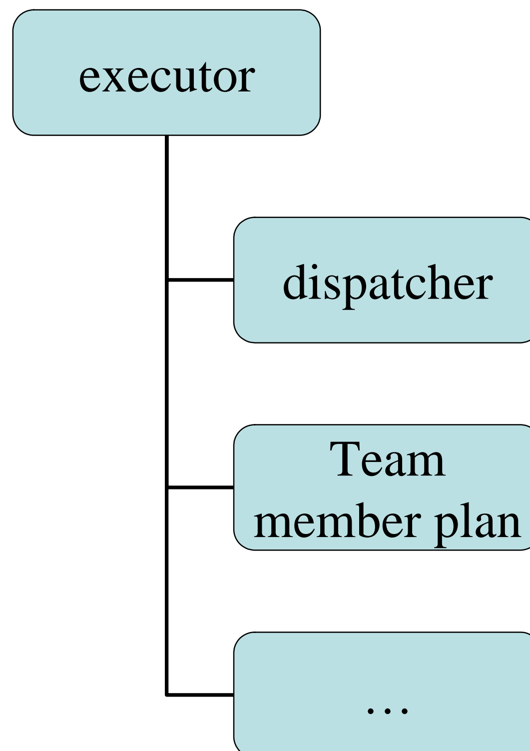

Figure 25: Template for MALLET domain plans for virtual team member

The basic template for an executor plan and dispatcher plan are shown below in Figure 26. In the example, the executor plan executes in parallel (**par** construct) the dispatcher plan and a number of domain specific plans based on the cognitive task analysis. The dispatcher plan uses a set of conditions to enable operators or sub-plans to be fired (executed) with specified delays. Typically such operators or sub-plans are enabled by responses (e.g. other domain specific sub-plans) by the agent. As long as the requirement for a natural response by the virtual team members is met, other approaches using MALLET may be taken by the training system developer to encode the domain plans of the virtual team member.

```
(plan executor ()
     (process
          (par
               (dispatcher)
               (planA)
               (planB)
               (planC)
               (planD)
          ) ; end par
     ) ; end process
) ; end plan executor

(plan dispatcher ()
  (pre-cond (actionRequest ?anyName ?anyDelayTime))
  (process
    (seq
      (foreach (cond (actionRequest ?name ?delayTime))
          (seq
          (retract (actionRequest ?name ?delayTime))

          (if (cond (> ?delayTime 0))
            (delay ?delayTime)
          ) ; end if

          (if (cond (okToAct ?name))
            ; THEN
            (nullIoper)
            ; ELSE
            (seq
              (assert (okToAct ?name))
            )  ; end seq
          ) ; end if

          ) ; end seq
      ) ; end foreach
    ) ; end seq
  ) ; end process
) ; end plan dispatcher
```

Figure 26: Skeleton executor and dispatcher plans

A key aspect in the design of a CAST Agent acting as a virtual team member is the naturalness of that agent's behavior as perceived by the human trainees. As an example in the development of the virtual team member agents for DDD, the development team eventually developed two different sets of DDD MALLET plans for the virtual team members (Srivathsan, 2005). The first version of the DDD MALLET plans was developed based on expert strategies, aiming at gaining optimum scores, but without consideration of the interactions of the virtual team member and the human trainee. The second version was developed taking into consideration the response obtained from human trainees who trained with the first version of the DDD MALLET plans, with a view to making the virtual team member seem more natural to the human trainees. It was observed that the second version that was developed taking into consideration trainee interactions in even a limited form appeared to provide better training results. The human trainees obtained better scores after training with the second version of the DDD MALLET plans.

The time required for developing the MALLET plans and the number of and complexity of the plans for a domain is based in part on the level of activity the training system developer requires of the virtual team member. As an example MALLET plans for a multiplayer Wumpus World (Russell & Norvig, 1995) were initially developed in a few weeks by a single developer and currently consist of 478 lines. The MALLET plans for the DDD test domain were developed over the course of a year by five developers and consisted of 3182 lines of MALLET code. The DDD test domain plans were

developed based on cognitive task analysis done by the researchers for the domain in order to find the expected actions of the team members for encoding as MALLET plans.

In summary, while the Framework provides both an intelligent agent and a template for the development of a virtual team member, the key requirement is that a cognitive task analysis of the positions to be emulated must be available for use by the training system developer.

5.4 Monitoring Trainees

Monitoring a trainee allows an instructor to evaluate a trainee's performance. Team training extends this monitoring requirement to include individual task performance within the context of team performance. Given a team context the Framework represents the trainee domain actions in a form relevant to team performance. In order to record such actions the Framework monitors each trainee and the virtual team members during the execution of the training session. For this purpose, the name, parameters, time, and success or failure of an action are required to be recorded for use by any number of individual and team performance assessment metrics. This monitoring requirement is the subject of this section.

For team training, trainee actions may be subdivided into individual taskwork and team interactions. Individual taskwork are the domain actions a trainee undertakes to accomplish individual and team goals. Team interactions account for the communications and coordination acts that a trainee undertakes to fulfill that trainee's roles and responsibilities within the team. For the Framework, monitoring is done for both taskwork and teamwork of a trainee. As part of monitoring, the recorded data is

stored for use by both assessment and coaching modules as supported through the Framework. In the Framework this recording is referred to as the *student trace* (see Section 2.3.1 in reference to Sherlock). The student trace provides a structured data set to be used by the training system developer to develop assessment and coaching support. Each trainee would have a student trace.

An important point to make is that within the Framework each trainee in the team will have their own Monitor Agent. The Framework represents a team in the training domain as a combination of the Virtual Agents playing the role of virtual team members and each trainee with their associated Monitor Agent.

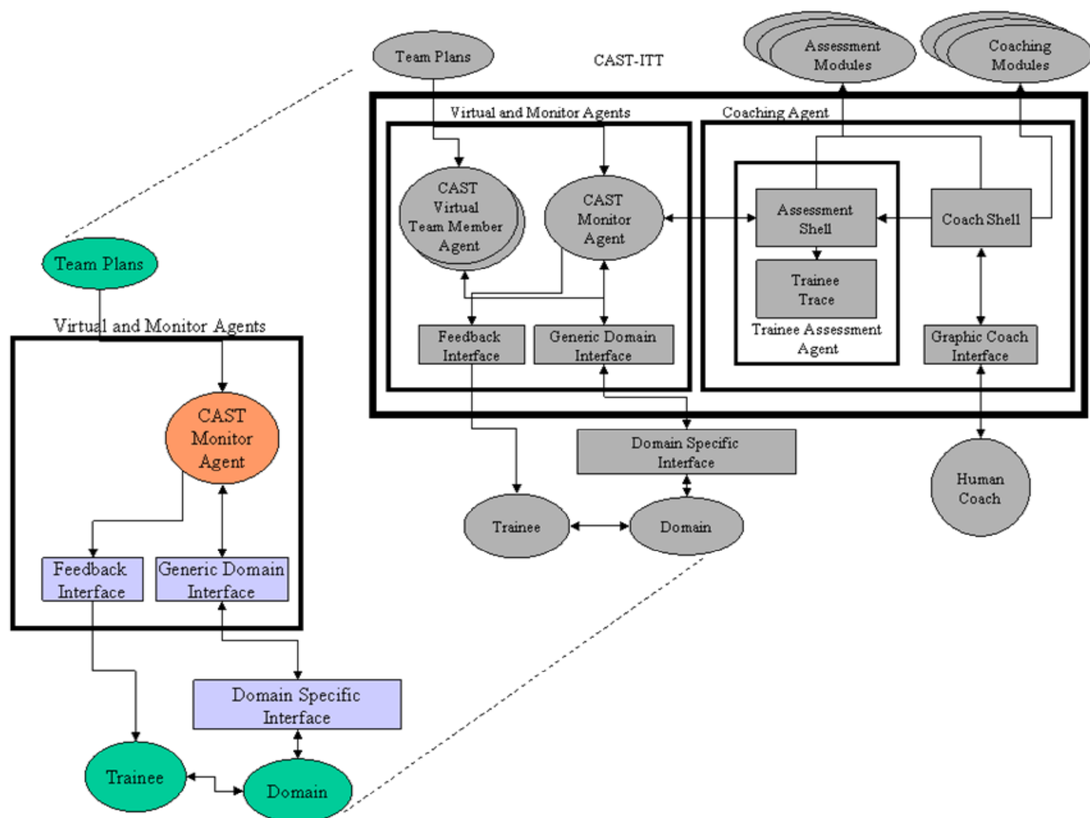

Figure 27: Monitoring agent in CAST-ITT

The CAST Monitor Agent as shown in Figure 27 includes a mechanism for recording monitored trainee domain actions. Each Monitor Agent also collects data of team interactions with a trainee. Thus, the Monitor Agent does more than previously mentioned in Section 5.2 on communications. The Monitor Agent is extended from the CAST Agent and has access to all the same interfaces and data structures of a CAST Agent. The Monitor Agent accomplishes its monitoring functions through use of the same operators and sense predicates that the training systems developer must create for the Domain Specific Interface that is extended from the ActorDomain abstract class.

The Feedback Interface will be discussed in Section 5.6.2 as part of coaching and is only shown in Figure 27 for completeness. In this section the interfaces within the Framework to meet the requirements of a Monitor Agent will be discussed. In order for each Monitor Agent to be able to record trainee activities, requirements are also imposed on the training simulation domain and training system developer. The training systems developer must create code to capture every action by a trainee. Typically, this must be done in two parts, one residing in the domain and one residing in the extension to ActorDomain. For example, the code in the domain must actually capture the issuance of a command and send it via some low level communication mechanism (e.g. socket connection) to a method in the ActorDomain extension. Once received by the ActorDomain extension it is recorded by a Monitor Agent through invocation of the traineeAction method provided by the CASTInterface (detailed description in Section 5.1). In addition to storing the actions, traineeAction also stores related sense knowledge of the domain into the student trace that exists for each trainee.

The use of Virtual Agents executing MALLET plans allows the Framework to provide an additional ability beyond merely monitoring the trainees. Not only are trainee actions monitored but the actions of Virtual Agents are logged. A Virtual Agent does not require the domain specific extensions for capturing commands, as the commands are issued from within the agent. Thus, in this case, the Framework can (and does) automatically log the commands and related sense knowledge, and the training systems developer need do nothing in terms of monitoring the Virtual Agents.

However, the use of Virtual Agents does raise the issue of handling coordination of teamwork between agents and trainees. As a first step, since a Monitor Agent is an extension of the CAST Agent (which is the Virtual Agent), the Monitor Agent is used as a liaison to a trainee in order to automate support for the coordination performatives generated by Virtual Agents engaged in teamwork. How to automate this support within a specific training domain raises the issue of what the training system developer does with the Framework in either revealing these coordination performatives to trainees or masking such performatives from trainees. This depends on training requirements, as discussed earlier in Section 5.2.2 on agent to human communications.

For monitoring team member (trainee and agent) communications, direct communications are captured as a domain action. Other elements of communications such as performatives are embedded within direct communications. They are identified by prepending the string “performative” on the message; that is, a performative message will have the format (performative …). Since the message, once formatted, looks like any other message, the same mechanism can be used for recording all messages,

regardless of type. In addition to the communication and command logging, there is information flow analysis (based on IARG) that can be done by both virtual agents and trainees. For purposes of later assessment or coaching the state of this information is also of interest and is logged.

In the subsequent sections, these issues are discussed in greater detail.

5.4.1 Monitoring Domain Actions

The Framework provides an extension to a CAST Agent in order to make a Monitor Agent. The extension includes a mechanism for handing much of the trainee monitoring requirement in a generic manner. The training systems developer need only create the aspects of monitoring that are domain specific, as described below.

The Framework requires that every domain action that a human trainee can perform be included in the set of domain operations that can be used in MALLET plans. Moreover, every domain operation that a Virtual Agent can execute must be performable by a human trainee. This equivalence is based on the generic nature of the teamwork represented in CAST. Specifically, this equivalence allows the CAST Agents (both Virtual and Monitor) to use the IARG algorithm to examine information needs that exist between team members.

Given this equivalence, domain actions (operators) are expected to be discrete commands to be issued by a trainee and captured by a Monitor Agent. As discussed about MALLET in Section 4.1, operators are discrete domain actions.

Captured commands must have the format of a unique name for each command type and zero or more string values for arguments (see Section 5.1.1). Below in Figure

28 is the code segment example[3] the training system developer should use in their domain specific extension to ActorDomain to record monitored commands. The actual recording of commands is done in the generic traineeAction method provided by the Framework. What is not shown in the example below is how the training system developer implements the receipt of captured commands as they are issued by a trainee. It is expected that the domain provides (or can be modified to provide) the capability to notify the ActorDomain extension when such commands are issued by a trainee. At the time that the command is received by traineeAction, the current local time and the world state as contained in DomainEnv (discussed in Section 5.1.2 on handling sensing) are recorded by traineeAction.

```
import cast3.MALLET2.net.Operator;

String commandName, arg1, arg2, …;

;values for the above strings are acquired in a domain
;specific manner.  Then,

Operator capturedCommand = new Operator(commandName);
capturedCommand.addCallingArgs(arg1);
capturedCommand.addCallingArgs(arg2);
capturedCommand.addCallingArgs(…);
getAgent().traineeAction(capturedCommand);
```

Figure 28: Recording a monitored domain command

[3] The location of this code may vary, depending upon the interaction of the domain and the Framework. In an example to be shown below, it is included with the sensed data.

In the example of DDD, the training system developer inserted a method within the DDD domain code to capture each DDD domain command and its parameters and deliver the resulting string (prepended with a code to indicate a captured command was being sent) to the sense portion of the DDD specific extension of ActorDomain. By parsing the received string, the sensing system determines that a captured command has been received, and the code segment above is executed.

Data that has been collected from monitoring a trainee is actually stored in the Coaching Agent (see Section 5.6). This is done by the Framework when the traineeAction method is called; traineeAction transmits the data via RMI from the Monitor Agent to the Coaching Agent. The Coaching Agent stores the recorded data in a specific Trainee Trace module for each trainee. Given the nature of teams to be studied (C2 teams) the frequency of the commands issued by a trainee is expected to be at most once a second. The second expectation is that the associated sense data will be manageable in size. These two expectations as to frequency and size of the data to be transmitted to the Coaching Agent are dependent on the training domain; therefore the training system developer will need to evaluate how the storage requirement of the Framework impacts on their physical resources. Our experience to date has been that the data volumes do not impose any problems.

The data for each captured command is stored as an object of type Action defined by the Framework. The Action object stores the command and its parameters as a string, the state obtained from DomainEnv, and the simulation timestamp. The Action object is what is sent to the Coaching Agent and stored there; note that there is no local

storage of the data by the Monitor Agent. Further details of the Coaching Agent are found in Section 5.6.

A Virtual Agent automatically handles the monitoring of its own activities by logging all actions, information needs, and communications to the Coaching Agent. The format of that logging is in Appendix C. The training systems developer, in summary, must only do the following to support the monitoring capabilities.

- Ensure that the domain software captures each command issued by a trainee and sends it to the Framework via the ActorDomain extension, writing such code as necessary to achieve this.
- Place code in the extension to ActorDomain to receive, e.g., via the sense capability, the notification of a trainee command and call traineeAction.
- Verify that the size and frequency of command and state data will not overload either the network or the storage capacity of a Trainee Trace module.

5.4.2 Handling Performatives

The handling of performatives, especially coordination performatives, is one of the most complex issues with which the Framework must deal. The difficulties arise because CAST/MALLET was originally designed for dealing with the specification and simulation of agent teamwork. As such, considerable emphasis was placed on implicit handling of coordination among agent team mates. It is important to recognize that with humans there can be both explicit and implicit coordination. Representation of explicit coordination in MALLET would require explicit coordination messages of some kind

among the involved agents, and would be straightforward to handle. The more complex situation is when a part of the training objective is for a trainee to learn when to perform an implicit joint activity. This might involve each team member maintaining a coherent sense of time (start rescue operation at 5:00 a.m.), or be based on observations of the environment (start rescue operation when enemy guards change shift). Such implicit coordination goals are expressed in MALLET by use of the joint constructs without explicit coordination messages in the MALLET plans.

When humans play the roles of team members involved in joint activity, i.e., team operators, joint do's or team sub-plan invocation and execution, one must find a way for the human team member/Monitor Agent pair to deal with the implicit coordination issues. More specifically, when one agent must wait for one or more other agents to complete an activity before it does its next action (e.g., when agent A must complete its action before agent B does the next action), the CAST agent(s) (A in the example) send the appropriate performative to the agent(s) of successor actions (B in the example). If the agent of the precedent agent is a human, the human will not send such a performative, thus potentially blocking an intelligent agent from proceeding.

Similarly, when an agent is ready to perform a joint activity, CAST implicitly sends a performative to the other possible participants in the joint activity. The agent will not proceed until it receives an adequate response from the other participants. When one or more of the other participants is a human, there is no implicit response and the agent can become blocked at this point.

In the case of partially ordered activities (but not joint), the problem is readily handled. CAST automatically sends out **controltell** performatives at the completion of each activity. Thus, we just need to have the Monitor Agent do something that accomplishes the needed actions on behalf of the human participant. Since the Monitor Agent already receives notification (see previous sections) of every operation the human performs, it simply logs operator completions in a special internal coordination class. Then, when some other intelligent agent needs the completion information and doesn't receive a **controltell**, it simply sends a **controlask**, which the Monitor Agent can fulfill on behalf of the human by looking whether or not the necessary predecessor action has been completed, waiting for the action if it has not been completed, and then sending the necessary **controltell**. The more serious issues are with respect to the joint activities.

The two obvious approaches to resolve the problem for joint activity are: 1) convert the implicit coordination messages to be explicit at the human end, or 2) have the Monitor Agent automatically detect the human's activity and handle the implicit coordination performatives on behalf of the human. How these coordination issues are resolved impacts the performative monitoring and may even impact training requirements by necessitating extra activity on the part of trainees.

It can be argued that the first approach violates the intent of teaching a trainee to properly handle implicit coordination. However, with the first approach, the only thing that needs to be made explicit is that a trainee is starting a joint activity. Since a trainee should know that this is the case, it is only a mild variation to require the human trainee to declare that he/she is starting a joint activity by invoking an auxiliary operator.

In order to provide flexibility to the training systems developer, the Framework provides a basic mechanism for dealing with the implicit coordination performatives and allows the training systems developer to add more sophisticated mechanisms if desired. In this section, we discuss the detailed issues involved in each of the approaches mentioned above.

5.4.2.1 Converting Implicit Coordination Performatives to be Explicit

The nature of team operators, joint do's and team sub-plan invocations is that no team member proceeds until a sufficient number of the team members assigned to the coordination activity is ready to do it. CAST agents have built-in algorithms that implicitly send performatives to the other agents involved in the joint activity and wait for suitable replies before continuing. Humans, of course, do not have the built-in implicit capability to respond. The approach of converting the implicit coordination to explicit coordination is relatively straightforward.

For coordination purposes, the key performatives are **controlask**, **controltell**, and **assert**. The first two, **controlask** and **controltell**, are used in the execution of team sub-plans to communicate completion of operators within those team sub-plans. The third, **assert**, is actually a general performative that is also used in generating coordination messages for joint-dos and team operator invocations, and team sub-plan role assignments. For coordination, **Assert** performatives are sent by an agent to tell other agents that the agent is ready to start a joint activity; the other agent asserts the corresponding predicate in its knowledge base so it will have a list of participating agents.

For joint-dos and team operators, the **assert** performative consists of a predicate that begins with the keyword "sync". The other elements of the predicate are the synchronization node within the MALLET plan and the sending team member's name. Once the required set of "sync" predicates is received (one per participating team member), each team member is able to proceed with the joint activity.

For team sub-plan invocation, participating agents send **assert** performatives to each other to acknowledge that execution has reached a sub-plan invocation; this occurs before the **controlask** and **controltell** performatives come into operation in the actual execution of the team sub-plan. In this case, the **assert** performatives uses the keyword "agent-assigned" in the predicate. In the case of a human trainee entering a team sub-plan, a trainee will need to send his/her participation for that sub-plan to other participating agents.

Since we assume in this approach that the human will explicitly indicate when he/she is ready to perform the joint action, the training system developer must provide an interface to the human that allows he/she to make an explicit input that he/she is ready to perform a joint activity. It is interesting to note that the human trainee (or its surrogate Monitor Agent) does not need to initiate the sending of a joint activity to any other team member, since a trainee is being taught **implicit** coordination. Rather, it is sufficient that the Monitor Agent have the coordination information and simply respond when requests come from virtual team mates.

However, it is necessary for the human to be able to specify or find the type of joint activity. Fortunately, the Monitor Agent can help the human trainee with this. The

Predicate Transition Nets generated from the MALLET plans include a special type of Transition called a Joint Transition. The Joint Transitions mark coordination points with a “joint id” for joint-dos and team operators. The training system developer can allow the human to specify the coordination point in the plan as part of the human’s input of readiness to perform a joint activity. The Monitor Agent then determines from that coordination point the joint id to use. For example, the training system developer could display a list of coordination points from which the human can select.

In order to allow a trainee to respond to a Virtual Agent, the training system developer must provide a mechanism to generate and send the performatives on behalf of a human trainee. This is straightforward because the identity of the sending agent(s) is known (and hence the destination of return information) and the identity of the joint activity is known from the sender. When the human inputs that he/she is ready, the Monitor Agent can generate and send the reply to all other team members that need to receive it.

The only complicating factor is if the joint activity that a trainee decides to do differs from the one an agent (or other human) decides to do, e.g., one is ready to jointly pick up the sofa and the other is instead ready to pick up the coffee table. In this case, one of the team members (agent or human) has made an error. It is not the responsibility of the Monitor Agent to recognize and deal with this error. The coach should analyze and deal with anomalies like this. The Monitor Agent would just treat the disparate human input as a new joint activity readiness (see previous paragraph).

Note that this does not violate the earlier assumption of equivalence between human and agent domain commands as it is a special command to support the Framework and is not one of the "real" domain operators. Such a command would not be invoked by Virtual Agents and therefore will not show up in MALLET plans.

5.4.2.2 Automatically Handling Implicit Coordination Performatives

The automatic handling of implicit coordination would have to be handled by the Monitor Agent and is much more complex. To be handled in a completely general way, the Monitor Agent would need to be able to detect when the human was ready to execute a joint activity. If the human were exactly following the team plan, this could be handled by having the Monitor Agent track the progress of the human through the plan. However, as a principal purpose of training is to teach a trainee the plan, one must assume that there will be many instances in which the human does not exactly follow the plan. Thus, one must look for alternative ways to resolve the issue.

First, it is useful to realize that when an agent is ready to perform a joint activity and sends a performative to the other possible participants telling them that is ready to do the activity, it includes information on exactly where it is in the plan and which activity it is ready to do. When the Monitor Agent receives a performative for a trainee, it then knows where in the plan the agent(s) is (are), and it can use this information to help it decide when the human is ready. The Monitor Agent must just recognize when the human trainee has reached the corresponding coordination point. Similarly, if the human trainee reaches the coordination point first, the Monitor Agent must recognize that the human has reached this point. So, in both cases, the principal issue is

recognizing that the human trainee has reached a coordination point and identifying that point.

A simplifying assumption can be made which leads to a relatively straightforward solution to the problem. If one assumes that the joint activity is restricted to joint do's and team sub-plan invocations (no team operators), then it is only necessary that the team members be synchronized at their starting points, but the actual order of which team member does their action first is not specified. In addition, for each team sub-plan and joint-do, the Monitor Agent can analyze the corresponding MALLET code and determine the set of possible operations a trainee might perform as the first step in the corresponding joint activity.

Thus, whenever a trainee performs an operation, the associated Monitor Agent can determine a set of joint activities for which this might be the first step. It uses this information to respond to the next or any pending virtual agent coordination requests. Whether the response is actually correct or not is not the responsibility of the Monitor Agent. If the human trainee is performing correctly, the response will be correct. If the trainee is incorrect, the information will be logged and subsequent performance analysis and coaching will deal with the situation.

One must also consider the synchronization of the ending of a joint activity. A human trainee may go beyond the end of a joint activity before the other agents of the joint activity finish. The key point from the perspective of the Framework is the proper unblocking of each agent. However, CAST always handles the end of joint activities by the same mechanism it uses for managing the order of activities. It simply sends a

controltell performative when it completes an action or **controlask** performative if it needs to know when some other team member has completed an action. Since each Monitor Agent already keeps track of action completion, they simply respond to the **controlask** as described earlier. Other conditions, such as not waiting, are errors on the part of a human trainee and just need to be logged so that subsequent assessment and coaching can try to help the trainee. These conditions are relevant to what the training system developer will have to handle.

A more sophisticated approach would be for the Monitor Agent to use knowledge of the coordination point of the agents and the plan to determine what actions a human trainee must take just before he/she is ready to perform a joint action. A Monitor Agent can look for the performance of these precedent actions and trigger the response performative when it detects that the trainee is ready. This could handle team operators as well as **joint do**'s. Unfortunately, there are certain MALLET constructs that make such early detection problematic. Constructs such as **if**, **while**, and **choice** provide multiple paths of execution. It cannot be known what decision a human trainee will make when presented with these branches. It is not the role of the Framework to solve this problem.

5.4.2.3 Framework Mechanisms and Training System Developer Responsibilities

The first choice in handling these self-imposed issues was based on having the training system developer write additional code (e.g. making implicit performatives explicit through TraineeMessages). The second choice was to enhance the Monitor Agent to

support implicitly responding to received performatives. The choice was made to support implicit handling of performatives by the Monitor Agent.

The Framework handles human/agent synchronization needs for joint-do and team sub-plans by detecting team synchronization points as reached by the associated Virtual Agents and trainees. The associated Monitor Agent fulfills Virtual Agent team performative requests automatically. The consequence of this choice is to disallow the use of team operators. This implicit handling of team performatives is controlled by the TEAMUPDATE Boolean flag that can be turned on/off by the training system developer in the agent configuration file (see Appendix A). If set to TRUE, then the implicit team synchronization occurs. If set to FALSE, then the associated Monitor Agent simply logs the received team performatives and responses are not generated.

Agent/agent synchronization needs are handled by the Virtual Agents with no change. Implicit human/human coordination is monitored and logged through the recording of every action (and time thereof) by every human.

Beyond the defaults of the Framework and the decisions taken in designing MALLET plans there are ways for the training system developer to change the behavior of the Monitor Agent in a programmatic manner. One choice is the use of TraineeMessages as detailed in Section 5.2.2 to support the display of the joint activity points in an explicit manner to the trainee (per Section 5.4.2.1). A more complex choice is to actually alter the programmed behaviors of the Monitor Agent. The Monitor Agent provides for additional domain specific support as required through the MonitorModule abstract class. The MonitorModule abstract class is an interface through which one can

add modules that can modify the behavior of the Monitor Agent. The implemented modules are loaded by the Monitor Agent and executed each time the Monitor Agent executes using the Cast Agent decision cycle mechanism. Adding modules at this level is useful as MonitorAgent class extends CastAgent class and therefore provides access to the algorithms and knowledge stored in CAST. For example, if one wanted to capture the system state more frequently than once each performed action, it could be done in this way.

```
public abstract class MonitorModule
{
     private MonitorTrainee monitorTrainee;

     public abstract void execute();
     public abstract Vector getService(String name);
}
```

Figure 29: MonitorModule abstract class

Figure 29 illustrates the major components of the MonitorModule abstract class. The execute method, when instantiated, is executed during each Monitor Agent decision cycle (which overrides the Cast Agent decision cycle). Each module created has a unique name in order to allow its access by other modules based on MonitorModule. This allows modules to build upon one another. Each module also has a getService method which, in conjunction with its string parameter, is used to return the results (as an Object) of the module's computations to other modules (if required).

An example usage of MonitorModule is the MonitorBasicServices utility module which provides a KB query/modification service for the Monitor Agent. This service allows the training system developer to query the KB and assert/retract facts to the KB. The string passed to the getServices method consists of a predicate which is appended to one of the keywords “assert”, “retract”, or “query”. For “query” the Object returned is a Vector with the list of results. This service is also accessible (through RMI) by the Coaching Agent (discussed in Section 5.6).

5.4.3 Monitoring Communication Related Information

Direct communication among team members falls in the category of domain actions and hence is captured by the mechanisms described above. However, there are two categories of communications related activity that should be monitored that do not fall into the class of domain actions. First, the CAST Agents send and receive performatives (see Section 5.2.1) that are necessary for teamwork. Second, each CAST Agent (either Virtual Agent or Monitor Agent) evaluates the production of information needed by others or information it needs as determined from the MALLET plans and that agent’s IARG analysis of information production/needs. While these two categories are related there are differences and they differ from domain actions. Both are important for assessment and coaching, and hence both must be monitored.

5.4.3.1 Monitoring Performatives

Communications are received by both a trainee and the associated Monitor Agent during the execution of the training session. Some of these communications will come from

Virtual Agents and these will be in the form of performatives. The Framework monitors and logs such performatives through the Monitor Agent as discussed in Section 5.2.2. The handling of these performatives has been discussed before for coordination and for communications. In this section we merely note that all performatives are recorded and stored for use by assessment and coaching needs.

The monitoring of performatives described here is done automatically by the Monitor Agents and Virtual Agents. Nothing further need be done by the training system developer.

5.4.3.2 Monitoring IARG Activity

Within the Framework, CAST (being used for both the Monitor and Virtual Agents) uses the IARG algorithm (Yin et al., 2000) to identify information needs for each team member based on the MALLET plans. In the case of the Virtual Agent, identified information needs, both requests and provisions, become part of the log of the performance of the Virtual Agent. Both the predicates needed or provided and the list of potential providers and needers are stored.

For the Monitor Agent the situation is more complex. Trainee actions are known to the Monitor Agent only after the action has been executed. Moreover, a trainee may or may not be properly following his/her role according to the plan. Presently, the Framework only deals with the observed actions and does not try to implement plan tracking; the latter is a future research topic. Each time the Monitor Agent is notified that the trainee has executed a command, it simply looks at the preconditions and effects of the associated operator and invokes the IARG algorithm to identify and record

information that the trainee needed for the action precondition[4] and the information the trainee could have provided after the action, along with the identities of the potential providers and needers.

The monitoring actions described here are done automatically by the Monitor Agent once the training system developer has built the extensions described in the previous section. Nothing additional needs to be done to record the IARG activity.

5.4.4 Summary of Monitored Activities

The following events and activities are recorded for both the trainees (via their respective Monitor Agents) and the Virtual Agents by the Framework.

- Trainee domain actions and related sensed domain state
- Virtual agent actions, plan starts, and plan completions
- Virtual agent role assignments
- Virtual agent joint synchronization events
- Potential information flows as identified by IARG
- Performative receptions
- Performative transmissions (Virtual Agents only)

5.5 Performance Assessment

Beyond supporting a training environment through monitoring and virtual team members, a team training framework should support assessment of the trainees. The Framework meets this support by the provision of interfaces by which the training

[4] This is recorded in case the trainee executed the action without the proper information. This would allow a coach to determine this and provide corrective feedback.

system developer may add performance assessment modules and access the monitored data stored by the Framework and through the provision of certain generic assessment modules.

For assessment support, the Framework utilizes the Monitoring Agent, the Trainee Assessment Agent, and the Coaching Agent. From the Monitor Agent comes the initial monitoring of the trainee state. When the Coaching Agent receives the monitored state, it manages both individual and team assessment. For individual assessment, it passes the monitored state received to a Trainee Assessment Agent (one per trainee) corresponding to the trainee whose monitored data is being processed. The Trainee Assessment Agent stores the captured data. Individual assessment modules may be loaded as part of each Trainee Assessment Agent which is then able to execute the individual trainee assessment modules. The results of each individual assessment are available to the Coaching Agent for final coaching evaluation and generation of appropriate feedback (whether post-session or online). All of these agents exist in a single process that is connected to the Monitor Agents and Virtual Agents through RMI. Shown in Figure 30 is the logical layout of what has just been discussed.

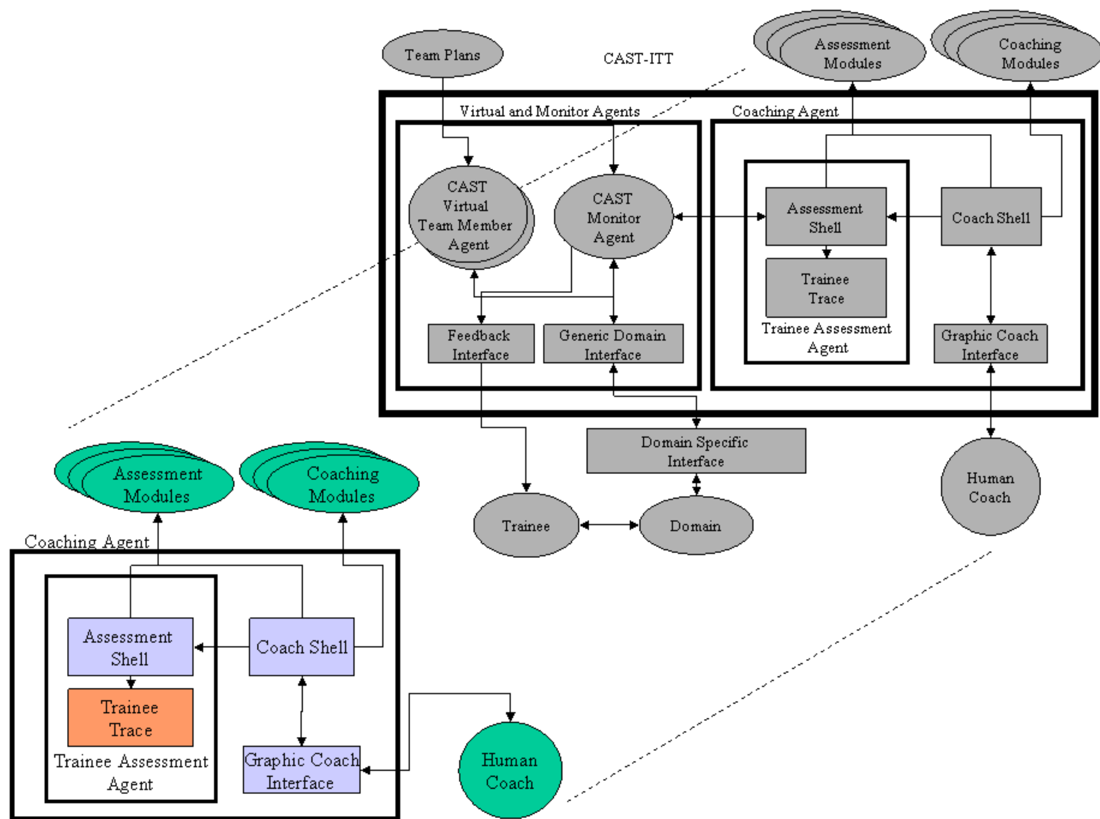

Figure 30: Assessment support in CAST-ITT

The Framework provides two corresponding shells (implemented as the PerformanceModule and TeamModule abstract classes) that the training system developer can extend for each assessment requirement. Each individual assessment module (there may be multiple such per trainee) extends the PerformanceModule abstract class and an instance of each is loaded as part of each Trainee Assessment Agent. Team performance assessment, on the other hand, is done directly within the Coaching Agent. Each team assessment module extends the TeamModule abstract class. The development of specific coaching modules is accomplished by extending the

TeamModule abstract class (discussed in Section 5.6). Each type of module that the training system developer must create is shown in green in the figure. The human coach (if desired) is also shown in green.

Combined with access to the MALLET plans, the student traces and virtual agent logs provide a foundation upon which the training system developer creates domain specific assessment modules. The access to the monitored data and the PerformanceModule abstract class and its use in creating individual assessment modules and the use of the TeamModule abstract class for creating team assessment modules are discussed in the next two sub-sections. The provision by the Framework of a limited amount of generic assessment support is discussed in the last subsection.

Assessment can be divided into in-session and post-session support. Mechanisms for such support will be briefly covered in this section. However, an in-depth discussion on the notion of what comprises a session and how that relates to in-session and post-session assessment and coaching will be part of next Section 5.6 on coaching.

5.5.1 Individual Assessment Support

Individual assessment is supported by the Framework through access to the monitored data about team members (both trainees and virtual team members). The PerformanceModule abstract class provides a structured approach to developing assessment modules that can in turn be accessed by other assessment modules (individual or team) or coaching modules.

The individual assessment modules are normally executed in sequence once per Coaching Agent execution cycle. The delay time between execution cycles can be set by

the training system developer through either the CAST configuration file (see Appendix B) or through the graphical user interface provided as part of the Coaching Agent[5]. Cyclic execution of assessment modules can also be toggled on (the default) or off individually. This allows assessment modules to be event driven (based on receiving data input from each trainee trace) or be executed by other assessment modules.

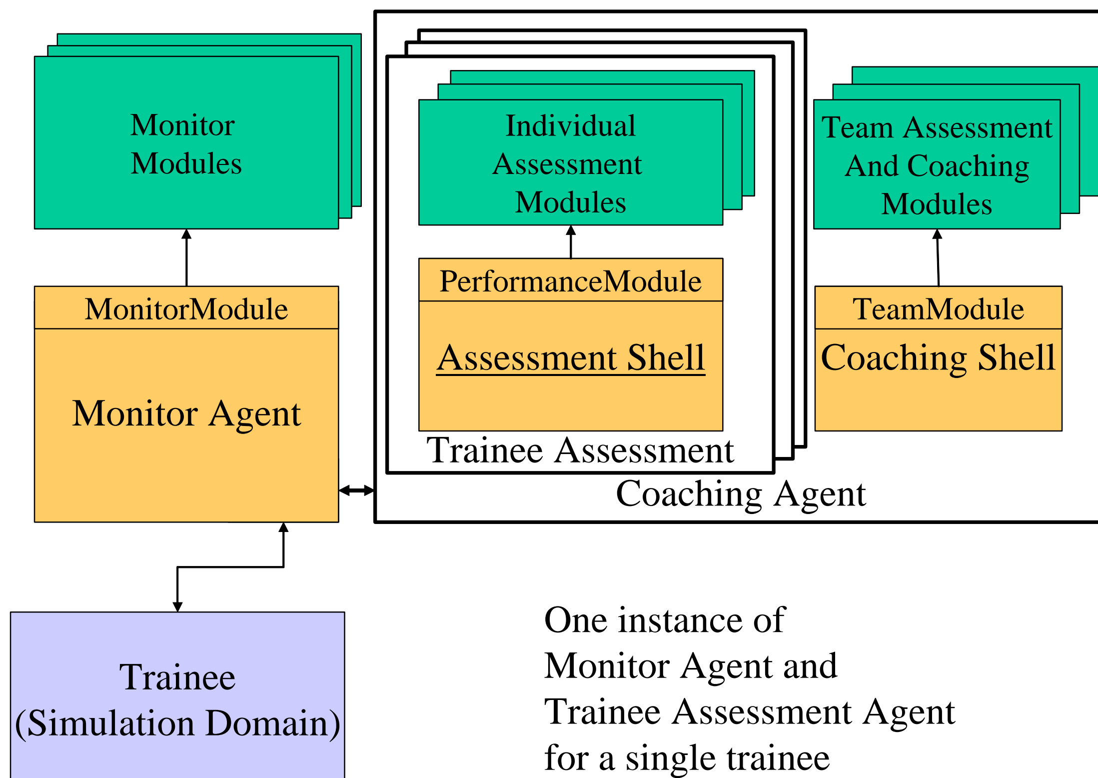

Figure 31: Logical view of individual and team assessment support

[5] The thought was to provide flexibility in the assessment and coaching in deference to varying the amount of data that potentially be collected (e.g. turn off assessment during instructional phases).

In Figure 31 is shown the logical view of the components of the Framework for assessment support. The Framework provides the PerformanceModule abstract class for developing domain specific assessment modules and gives those modules access to the monitored data. The Assessment Shell instantiates and executes the extended assessment modules for each individual trainee represented. In addition, the PerformanceResult interface (not shown in the diagram) extends the Java.util.Properties class to provide a property object (a collection of name value pairs) for storing assessment results.

In the next two subsections will be discussed accessing the monitored data and developing individual assessment modules.

5.5.1.1 Accessing Monitored Data

The capturing and delivery of the monitored data of trainees' actions and Virtual Agents activities has been discussed in Section 5.4. The monitored data is stored in a TraineeTrace object (one per trainee). The Trainee Assessment Agent (an instance of the TraineeAssessment class) provides methods for accessing its own TraineeTrace object. The PerformanceModule abstract class provides a handle to the TraineeAssessment class which the training system developer can utilize to invoke the TraineeTrace object access methods from within the individual assessment modules he/she develops.

The monitored trainee actions within a TraineeTrace object are stored as a LinkedList object. Each element of the linked list is an object of type Event (an abstraction of which is shown in Figure 32). Figure 32 also illustrates the key methods to the TraineeAssessment class and the Event class. The LinkedList containing the stored

Event objects can be accessed by the getActionEvents method. Each action is stored as an Event object on the LinkedList.

```
public class TraineeAssessment {
     public TraineeTrace getTrace();
     public String getLastAction();
     public Vector getLastSense();
     public LinkedList getActionEvents();

     public PerformanceModule
          getPerformanceModule(String name);
}
```

```
public class Event {
     public Vector getSense();
     public Date getTimestamp();
     public String getAction();

     public String getCommand();
     public Vector getArgs();
     public LogEvent getEventType();
}
```

Figure 32: TraineeAssessment and event classes

While the TraineeAssessment class has a handle to the TraineeTrace object, it also has convenience methods (getLastAction, getLastSense, and getActionEvents) to get directly to the last monitored data for a trainee.

The monitored data for an individual action of a single trainee are stored in the Event class. As described in Section 5.4.1, the time, current domain view as embodied in the sense data, and the action and its arguments are captured by the Monitor Agent. The data is stored in memory and written out in a flat file at the end of the session (the format

is in Appendix C). Also stored in an Event object are the other monitored data such as the performatives, the IARG events, and joint activities. The Event class uses the following event types: ACTION, IARG, PERFORMATIVE, PLAN_BEGIN, PLAN_END, JOINT, and ROLE; they are defined in the LogEvent class. PLAN_BEGIN, PLAN_END, and ROLE are used by the Virtual Agent logs.

The Virtual Agents use a different procedure for logging events. CAST uses the Java logging package to record data about the agents. The Java logging facilities hold the recorded logging data in memory until the end of a session, at which time the data is stored in a flat file (the format is in Appendix C). Additionally, the logged data is transmitted via RMI to a central monitoring process, the CastMonitor object. The Coaching Agent instantiates a CastMonitor object as part of its process. Therefore the logs can be accessed through the CastMonitor object stored within the Coaching Agent by the getLog(agent) method within CastMonitor which returns a LinkedList composed of Event objects. The level of detail of the information recorded can be varied by setting the level when the individual log is started in the XML configuration file (detailed in Appendix B). The training system developer may take advantage of these levels to include regular logging services or a more detailed troubleshooting level during development. Individual logs may be started at different levels.

Each log entry for an agent is divided into types; action, IARG, messages received, and team activities (plan begin/end, coordination, and role assignments). These seven types are labeled as ACTION, IARG, PERFORMATIVE, PLAN_BEGIN, PLAN_END, JOINT, and ROLE, as defined in the LogEvent class.

Together with the MALLET files (accessed as discussed in Section 4.1), the logs and student traces present a view of the taskwork and teamwork in the training domain. Based on this view, individual assessment modules can be written by extending the PerformanceModule module.

5.5.1.2 Assessment Interfaces

Domain specific assessment in the Framework is supported through the use of executable modules. Each module is intended as a single assessment view. Therefore each module has its own results for that assessment that are stored as part of that module. Except for the generic support (discussed in Section 5.5.3) all assessments are provided by the training system developer. The Framework provides the interfaces that support the execution of such assessments modules and their access to the monitored data. The key interfaces in developing individual assessment modules are the PerformanceModule abstract class and the PerformanceResult abstract class.

The PerformanceModule abstract class provides the following elements:

- Access to Trainee Assessment Agent
- Access to monitored data
- Access to Monitor Agent
- Generic assessment result format

The training system developer must provide the following elements for their implementation of an individual assessment module, which must extend PerformanceModule.

- Unique assessment module name (class name)
- Execute method for actual assessment (such execution may be event driven, cyclic, or post-session analysis)

The PerformanceResult abstract class provides the following elements:

- Generic assessment result format
- Support for graphically displaying results

The training system developer must provide the following elements for their implementation of an assessment result.

- Storage of results into the generic assessment result format
- Data specific visual display of the results (optional)

The implementation of an assessment module by the training system developer provides the implementation its name. The class name is the module name. The object instance is based on one per trainee. As an example if a TraineeActionCount individual assessment module had been developed then for each trainee there would be an object instance of that class. Each implemented PerformanceModule must have an implementation of the PerformanceResult abstract class. Developed together by the training system developer, both implementations provide a single assessment module and its result.

```
public abstract class PerformanceModule implements ModuleRunner {
      private TraineeAssessment trainee;
      private boolean active = true;      // has getter/setter
      private boolean postSession = false; // has getter/setter

      public abstract void execute(); /** from ModuleRunner */

      /** return an object that can be used as results */
      public abstract PerformanceResult getPerformanceResult();

      public JPanel getTabDisplay() { return null; }
      public String getTabTitle() { return null; }

}
```

Figure 33: PerformanceModule abstract class

In Figure 33 is shown an abstraction of the abstract class PerformanceModule that a training system developer must extend to create an individual assessment module. The variable trainee is the handle to the TraineeAssessment object that each PerformanceModule implementation receives. The Boolean active variable is used to toggle cyclic execution (on if true). The Boolean postSession variable is used to indicate it is now the post session analysis phase. This value is set to TRUE by the Coaching Agent when entering the post-session phase by pushing a button by the trainer after the end of the training scenario and all concluding activities. Also shown is a method for providing a user interface (a JPanel) by the training system developer. The intent of the Framework is that this JPanel would be used to display assessment results during the in-session phase. It is invoked by the Coaching Agent if such in-session display is desired. These graphical elements can be created during the creation of the object for display by the Coaching Agent, in which case the null return should be replaced with a return of the

appropriate handle. A more in-depth discussion of the Coaching Agent display is in Section 5.6.1.1.

The minimum required of the training system developer is to implement the execute method in the PerformanceModule extension.

The PerformanceResult abstract class provides a container and mechanisms for the training system developer to store and access the results of an individual assessment implementation class. The PerformanceResult abstract class includes methods for manipulating name value pairs of assessment results by using the java.util.Properties class as its underlying storage class. This storage format provides a single unified interface for accessing assessment results throughout the Framework. The training system developer may also add to his/her own implementation to fulfill his/her own storage requirements. In this case, the results will not be accessible in a generic manner, but may be accessed by other assessment modules aware of such specialized data storage.

```
public abstract class PerformanceResult extends
java.util.Properties {

      /** screen viewable summary of results */
      public abstract String toString();

      public String getResultName();
      public void setStampedProperty(String key, String data);
      public Date getStampedProperty(String key);
      public String allResults();

      public JPanel getTabDisplay() { return null; }
      public String getTabTitle() { return null; }

}
```

Figure 34: PerformanceResult abstract class

As shown in Figure 34 the PerformanceResult abstract class also provides convenience methods for generating a timestamp while storing data. The allResults method returns a constructed String of all data elements stored using the Property class. The graphical display of post-session assessment results may also be constructed by the training system developer using similar methods as provided in the PerformanceModule abstract class, i.e., creating a suitable JPanel for them. Again, the Coaching Agent would normally invoke the JPanel, but only after the post-session analysis is done.

The key difference in the graphical display is the location and timing of viewing of the displays. The PerformanceModule displays are shown in the Coaching Agent during the session as a tabbed panel in the tab maintained for each trainee. The PerformanceResult displays are displayed after the post-session generation of assessment.

If the storage requirements for the individual assessment results are straightforward the training system developer may instead use the SimpleResult implementation of PerformanceResult. SimpleResult is a default implementation provided by the Framework that supports storing of data in a text form only.

The Framework supports interactions between individual assessment modules by the use of a handle to other assessment modules. Accessed within the TraineeAssessment class is the getPerformanceModule (String) method. Using the implemented class name of the assessment module whose results are desired as the String argument, the training system developer may get an object handle to the

assessment module. From the PerformanceModule handle, PerformanceResult or domain specific access is available. Domain specific access is other methods that the training system developer might place in their PerformanceModule extension that can be accessed by other PerformanceModules the training system developer has developed. Since the calling PerformanceModule extension has to cast the class of the called PerformanceModule extension this is no longer generic access.

Individual assessment modules are only accessible to other individual assessment modules instantiated as part of the same trainee or through the Coaching Agent (used for creating team assessment modules). If access is desired to assessment modules for multiple trainees then such access should be added when extending the TeamModule abstract class described in Section 5.5.2.

5.5.2 Team Assessment Support

A key difference between the Coaching Shell and the Assessment Shell is that there exists only one Coaching Shell for the team in a training domain in contrast to the existence of one Assessment Shell per individual trainee. The Coaching Shell instantiates the desired team assessment modules from implementations of the TeamModule abstract class. The training system developer creates those implementations in order to create assessments of team performance that are able to build upon the individual assessment modules.

The Framework provides the TeamModule abstract class which is identical to the PerformanceModule in its elements except for a single key difference. As shown in Figure 35 the TeamModule abstract class provides a handle to the Coaching Agent.

```
public abstract class TeamModule implements ModuleRunner {
      private CoachAgent coach;
      private boolean active = true;      // has getter/setter
      private boolean postSession = false; // has getter/setter

      public abstract void execute(); /** from ModuleRunner */

      /** return an object that can be used as results */
      public abstract PerformanceResult getPerformanceResult();

      public JPanel getTabDisplay() { return null; }
      public String getTabTitle() { return null; }

}
```

Figure 35: TeamModule abstract class

Otherwise each team assessment module must extend the corresponding elements that had to be extended for an individual assessment module using PerformanceModule. Through the coach handle the training system developer may access the individual Trainee Assessment Agents in order to access individual assessment modules for a trainee. Within the CoachAgent class the method getTrainee (traineeName) is used to access the monitored data and assessments of a specific trainee. The active and postSession flags are discussed in Section 5.6.1.2. The default values will allow the extended module to execute during the in-session phase. The TeamModule abstract class also requires the use of PerformanceResult for storing the results.

5.5.3 Generic Support

To facilitate the development of assessment modules by the training system developer the Framework has an initial set of generic support modules. These generic support modules ease access to the Framework and are usable across multiple training domains.

These generic modules also offer examples to the training system developer in formulating an approach to developing their own assessment modules.

The generic modules fall into two categories. The first category is a collection of default modules to be used in a default configuration by any training domain with minimal work and also to test the Framework. The second category is a collection of convenience modules to improve access to the Framework and provide a minimal set of assessment modules based on the model of teamwork supported by the Framework.

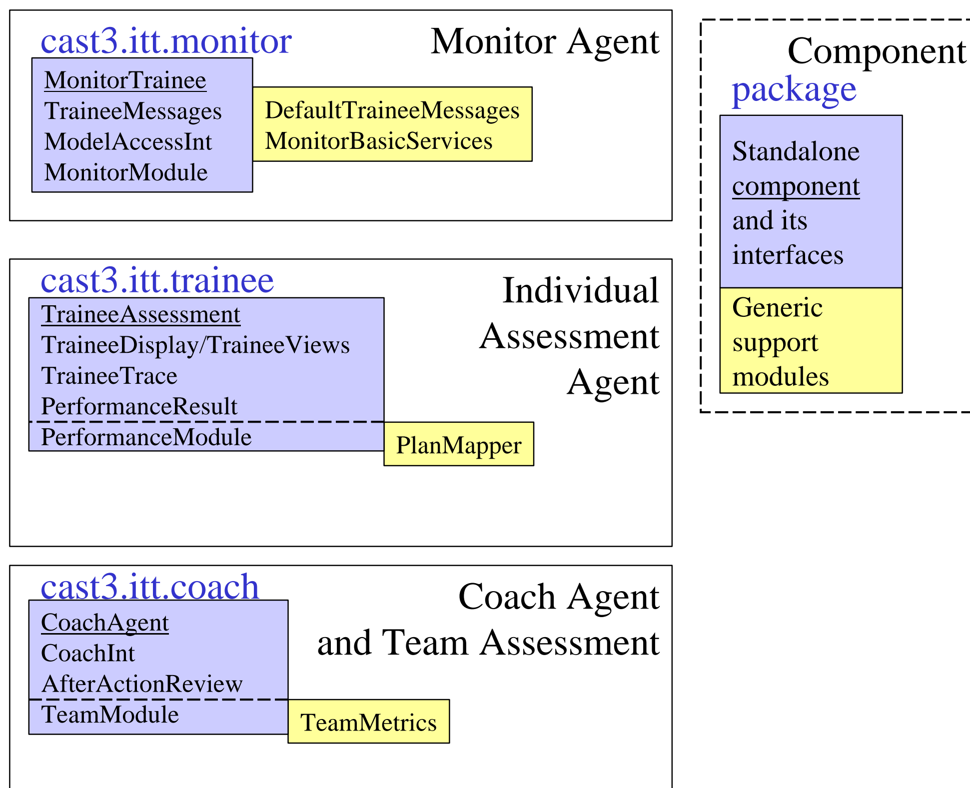

Figure 36: Generic support modules

As shown in Figure 36 there are four generic support modules. DefaultTraineeMessages and MonitorBasicServices fall in the first category of default capabilities in the Framework. PlanMapper and TeamMetrics fall in the second category of generic assessment support.

DefaultTraineeMessages displays performatives to a trainee. It is useful for the initial development and testing of domain specific components of CAST-ITT. This default class displays the performatives without any manipulation through the use of a MessageDialog. If not desired, use of this class can be replaced through the CAST configuration file via the TRAINEEMESSAGE variable using the class name of a domain specific class or it can be set to a null class if not desired.

MonitorBasicServices provides a limited level of access to the Monitor Agent without further work by the training system developer. Using the getService(String) method with either “query (some predicate)” or declare (some predicate)” the training system developer may query or modify the knowledge base of a Monitor Agent.

PlanMapper, shown in Figure 37, provides access to the MALLET files for use by the assessment modules. The MALLET files are parsed and stored in a MALLET object for access. PlanMapper also provides a chronological graphical display of the actions taken by a trainee that is displayed by the Coaching Agent; typically, this is done on a separate processor and monitor for the benefit of the trainer. PlanMapper is loaded automatically for every trainee and sends the display data to the Coaching Agent.

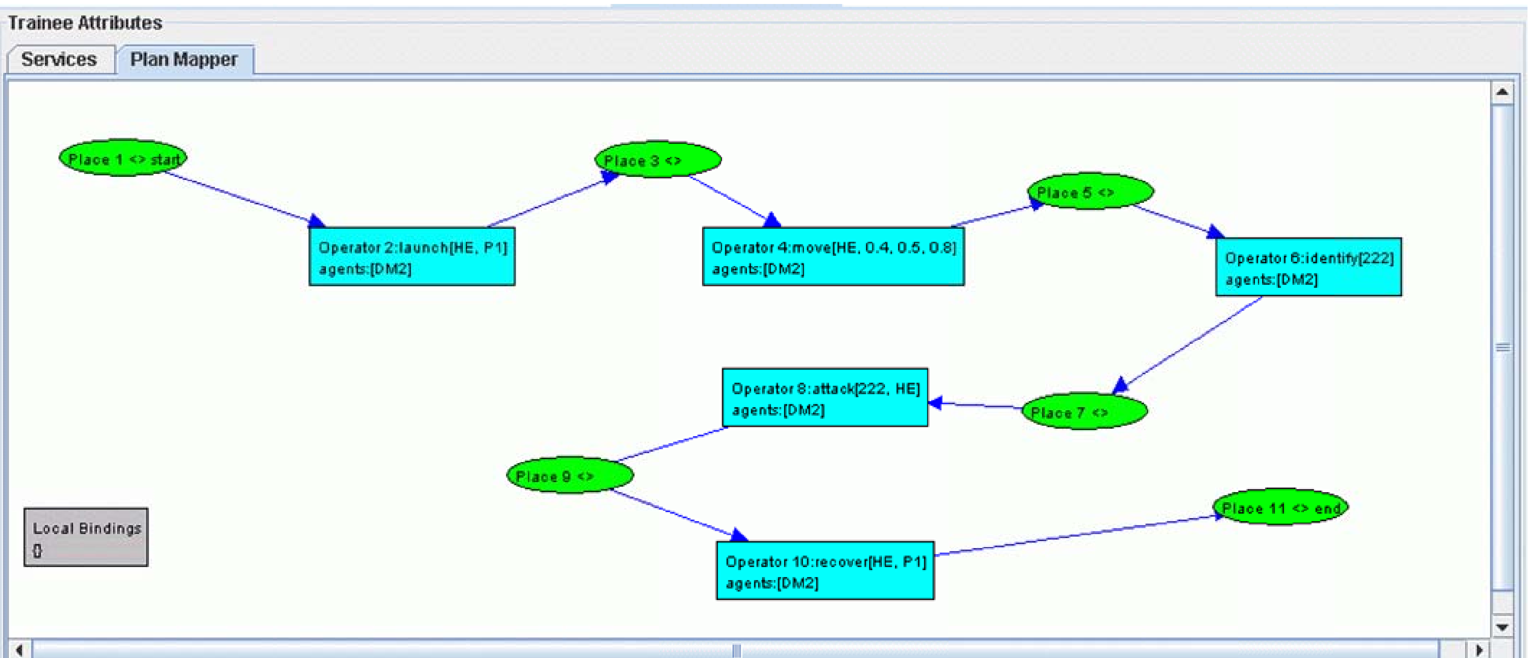

Figure 37: Trainee visual action trace

In Figure 37 is a screen shot of the PlanMapper visual display from a test run. The display presents the list of trainee actions in a Petri Net graph form. This implementation is useful as an example of how to provide such displays for other assessment modules.

It is useful to use the TeamMetrics class as an example of how the elements of the Framework come together to support a team assessment module. TeamMetrics provides a basic count of several elements of monitored data by the trainees and Virtual Agents. When the Coaching Agent invokes TeamMetrics execute method during the post-session phase of the Framework, it does a count of the message performatives sent by member, IARG needs, and synchronization operations provided. It also sums the team member messages sent to provide a total of messages sent within the team. It does these computations by accessing both the Trainee Assessment Agents' trainee traces and the Virtual Agents' logs.

```
public class TeamMetrics extends TeamModule {
      public int getTeamMessageCount();
      public int getMemberMessageCount(String member);
      public int getMemberSync(String member);
      public int getMemberIARGneeds(String member);
}
```

Figure 38: TeamMetrics class

In Figure 38 is shown the TeamMetrics class methods available to the training system developer. The training system developer uses the string "cast3.itt.coach.dynamic.teamMetrics" passed to the getPerformanceModule method in

TraineeAssessement and then casts the resulting object to TeamMetrics in order to access the methods.

5.6 Coaching in Support of Teamwork

The Framework provides support for coaching that can be used either by a human coach (called the trainer) or a software coach to be built by the training systems developer. The Framework uses a Coaching Agent as the underpinning for supporting coaching. The Coaching Agent provides a point of organization in assembling the various components for assessment and feedback. The Coaching Agent works in conjunction with the Monitor Agent and the Virtual Agents to provide assessment and coaching interfaces and monitoring support. The outputs of the built-in generic tools are provided to the trainer with no further development, and can be used by the trainer to provide feedback to the team being trained. In order to provide domain specific assessment and feedback to the trainer or to develop an automated software coach, the training system developer must use these interfaces to build the needed display (see Section 5.6.1.1) or coaching modules. The Coaching Agent as provided by the Framework is more akin to a shell into which functionality can be added by the training system developer.

The training system developer must provide the actual intelligence, display, coaching, and feedback mechanisms within the Coaching Agent. The intelligence may be modeled on whatever approaches the trainer wishes to have built into the training system, and is then expressed in terms of the coaching and feedback mechanisms provided by the Framework. The coaching modules use the assessment modules and

generate feedback. The feedback mechanisms support the actual delivery of feedback to a trainee generated by the coaching modules.

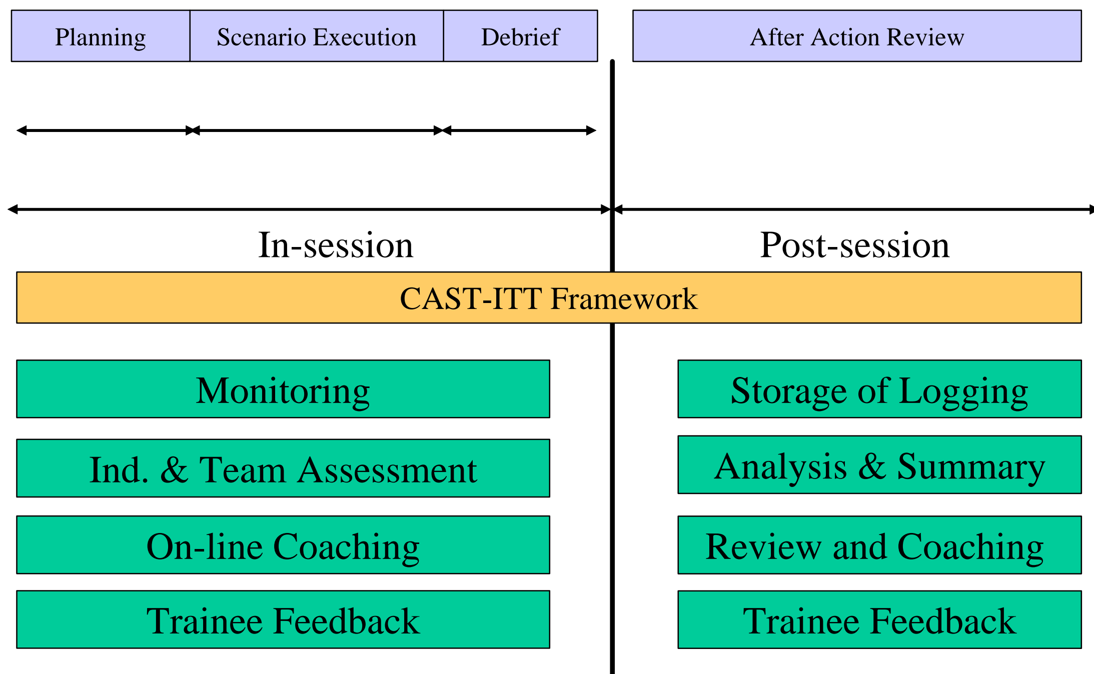

Figure 39: Training session timeline

In Figure 39 is shown a timeline of expected phases that occur in a typical training environment. The Framework divides a training session into an in-session phase and a post-session phase. The in-session phase may include more than the actual training simulation execution. It might also include a planning step and end of session activities such as questionnaires (e.g. for psychology experiments). The post-session is a distinct phase for the Framework that is conducted after all training is complete. In respect to

coaching, the Framework provides support for both in-session and post-session coaching. This distinction between in-session and post-session is also made in the assessment modules. Also shown in the figure is a functional view of the Framework for each phase. The monitoring of the team generates data during the in-session phase; that data is then stored at the beginning of the post-session phase. Analysis and summary of assessments is also performed at the beginning of the post-session phase. In stressful real-time environments only feedback that can be provided between events and that does not overload a trainee may be all that is possible. Therefore such considerations of when to present feedback are left to the training system developer. In depth discussion for review purposes are intended to be reserved for the post-session phase.

The next two subsections discuss the interfaces the Framework provides for supporting coaching modules and providing feedback during in-session and post-session phases.

5.6.1 In-session Coaching

In-session support for coaching feeds from the assessment support interfaces provided by the Framework. These assessment support interfaces (PerformanceModule and PerfomanceResult) are the high level mechanisms, and along with the Monitor and Virtual Agent logging, they feed into the Coaching Agent. The Coaching Agent is the focal point of the coaching and feedback interfaces of the Framework. During the training session the Coaching Agent executes coaching modules which may then generate feedback to a trainee or a human coach during the session. The Coaching Agent also provides a visual display capability that is intended for use in both testing by the

training system developer and during execution by a human trainer. How much real-time oversight and evaluation is provided by the coaching modules and by a human trainer is left to the training system developer to determine.

In the next three subsections, we discuss the displays that the Coaching Agent provides to the trainer, the support for coaching evaluations (i.e., determining what feedback to give to the trainees), and the support for coaching feedback (i.e., the mechanisms for providing the feedback, once determined, to the trainees).

5.6.1.1 Displays for Trainer

It is important that the Framework provide a generic mechanism for display of a wide variety of information. Accordingly, the Framework creates a generic display with a hierarchy of tabs for selecting specific displays. At the top level of the tab hierarchy are the generic display tabs for the agents that run in their own threads, i.e., the CAST display agent described in Section 4, the Coaching Agent, and agents for every team member.

This hierarchy of tabbed panels can be seen in Figure 40. Shown in Figure 40 is a team with four members. Two members are Virtual Agents, Agent DM0 and Agent DM1, and two members are human trainees, Trainee DM2 and Trainee DM3. All of the displays that can be selected by the tabs are intended for use during execution of a training session. In this example, the coaching agent is the top level tab selected. Notice that this choice then displays the subordinate tabs that are available for the agent to select. When a tab in this second level is selected, a third level of tabs for the item selected are displayed. Three levels of these tabs are provided by the Framework. In the

case shown, the main window displays the progress of the selected trainee (at level 2) through the plan.

Hierarchically under the Coaching tab are tabs for each team assessment (only one named “Team Members” in the above figure) and coaching module (named “Generic Coach Tools in this example) as well as one for each Trainee Assessment Agent. At start up, the Coaching Agent loads a unique display window for each of these modules (if provided by the module) and creates a tab for it under the Coaching Agent tab. The specific contents of the displays may be tailored to whatever is needed for the given module. These in-session displays are generated by TeamModule objects via the JPanel displays the training system developer creates. It is worth noting that tabs that refer to individual modules associated with individual team members really connect to (usually) remote processes through RMI interfaces. The individual team member level (the Monitoring Trainee DM2 and Monitoring Trainee DM3 tabs in the example) is where the displays from extensions to the PerformanceModule abstract class, again via JPanels the training system developer creates, are shown.

To be more specific, if the trainer were to select the Monitoring Trainee DM2 tab (as shown in the figure), a subordinate set of tabs would appear, one for each performance module having a display. The displays for each performance module come from the JPanel that the training system developer created when extending the PerformanceModule abstract class (see Section 5.5.1.2).

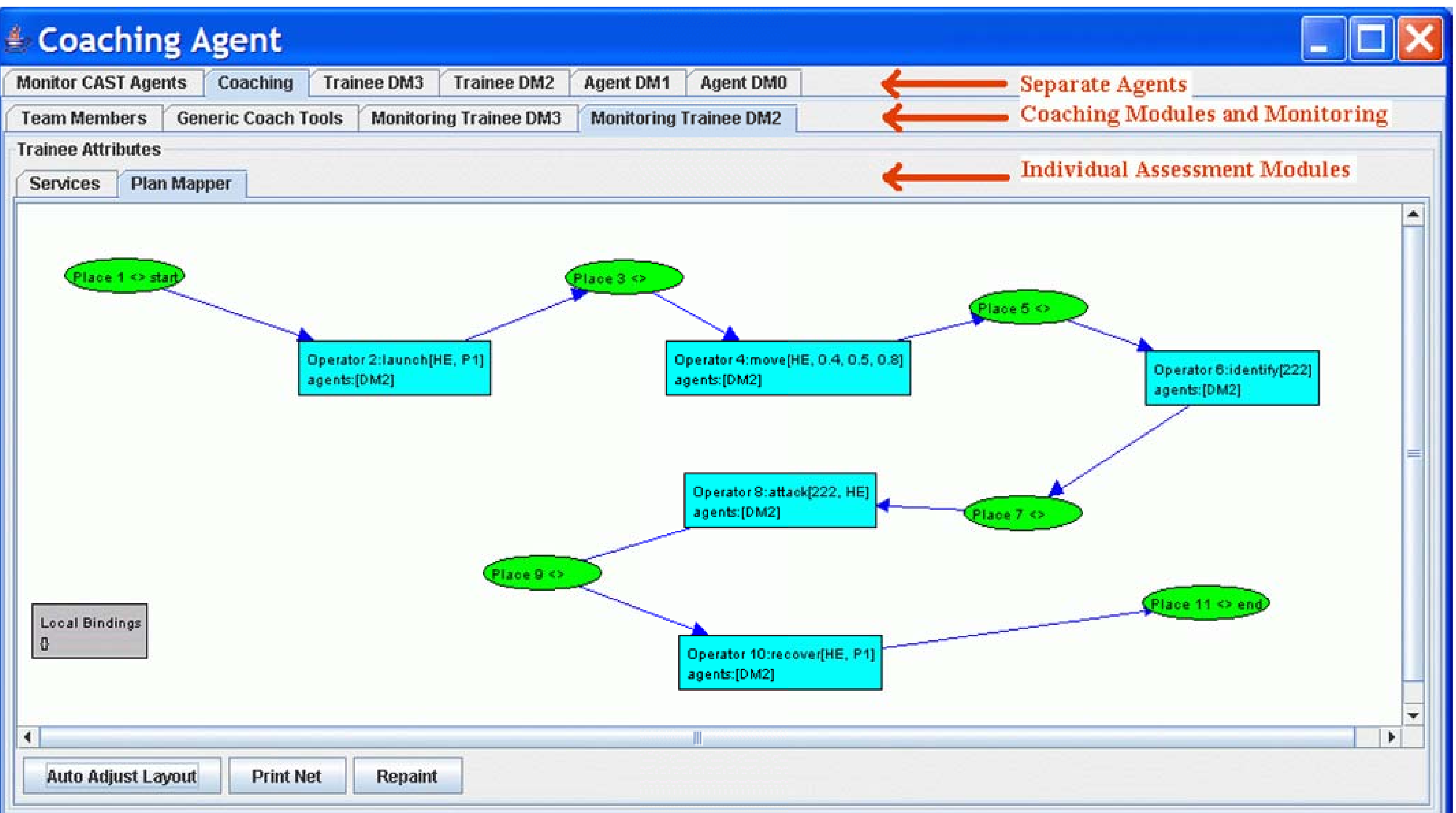

Figure 40: Coaching agent

5.6.1.2 Support for Coaching Analysis during In-Session Execution

Support for the generation (as distinct from actual presentation) of coaching feedback during both in-session and post-session is done generically through modules loaded by the Coaching Agent. Coaching support is placed at the team level (i.e. within the Coaching Agent and not in the Trainee Assessment Agents) in order to support team-oriented coaching. Placing the modules at the team level allows the modules to access data about the state of all team members in generating appropriate coaching responses. However, coaching feedback is oriented towards individual trainees as the team is ultimately comprised of individuals.

At the startup of execution for the in-session phase, the Coaching Agent loads an instance of the Trainee Assessment Agent introduced in Section 5.5 (which in turn loads the individual assessment modules) for each trainee in the team[6]. The Coaching Agent then loads two sets of implementations of the TeamModule abstract class, one set for team assessment and another set for coaching purposes. As shown in Figure 41 the TeamModule abstract class used for developing coaching modules is the same class used for the team assessment modules. The single abstract class TeamModule was used because both assessment and coaching modules need to access the data for the entire team and the coaching agent.

---

[6] There is only one Coaching Agent for the team.

```
public abstract class TeamModule implements ModuleRunner {
      private CoachAgent coach;
      private boolean active = true;      // has getter/setter
      private boolean postSession = false; // has getter/setter

      public abstract void execute(); /** from ModuleRunner */

      /** return an object that can be used as results */
      public abstract PerformanceResult getPerformanceResult();

      public JPanel getTabDisplay() { return null; }
      public String getTabTitle() { return null; }

}
```

Figure 41: TeamModule abstract class

TeamModules may execute in both in-session and post-session or in only one of the phases. This is determined by the active and postSession Boolean flags shown in Figure 41. The active flag is set by the module creator and determines whether or not the module executes during the in-session phase. The active flag may be set externally by other modules (written by the training system developer) in order to allow a hierarchy of modules to execute as desired during portions of the in-session phase (e.g. event-based rather than cyclic). The postSession flag is different in that it is set by the Coaching Agent at the beginning of the post-session phase to indicate the module has entered the post-session phase. Every TeamModule is executed at the beginning of the post session. The training system developer must simply include code to check the postSession flag and not execute the module if it is not needed for the post session.

The Coaching Agent executes each instantiated TeamModule in a sequence based on the order given in the COACHING.XML configuration file (see Appendix A)

used to identify what modules to load[7]. During in-session execution the team assessment and coaching modules are executed at an interval determined by the training system developer. The default interval is every 10 seconds and the interval variable COACHINTERVAL in the CAST XML configuration file may be used to change this value.

The coaching modules have access to other such modules (both coaching and team assessment) and the individual assessment modules. This access was discussed as part of the discussion on the team assessment modules in Section 5.5.2.

As noted above, the TeamModule abstract class has a display interface that is loaded and used during the in-session phase. This display is based on the JPanel class and is loaded under the Coaching tab, as shown in Figure 40. The Generic Coach Tools tab is an example of such a display loaded from the module BasicCoach. All of the displays that are part of the Coaching Agent are intended only for the trainer and not the trainees. These displays give the training system developer the opportunity to add a visual aspect to their implementations of the individual assessment, team assessment, and coaching modules.

5.6.1.3 Support for Coaching Feedback during In-Session Execution

Once assessment and generation of coaching feedback are completed the training system must be able to provide the feedback to a trainee. The Framework provides two mechanisms for generating feedback to a trainee during the in-session phase. The first

[7] Note that there is no distinction in this respect regarding whether the modules are for team assessment or coaching. A training systems developer might have reasons for any ordering.

mechanism provides feedback directly from the coach (human or software agent) to a trainee. The second provides feedback from the coach to a virtual team member, which may in turn take some action to induce a trainee to take the proper action, e.g., asking the trainee to provide some information. The first mechanism is provided by the MonitorModule interface and the second mechanism is through use of the Virtual Agents.

The MonitorModule abstract class has already been introduced in Section 5.4.2.3 for the purpose of domain specific monitoring enhancements. In the context of trainee feedback, the MonitorModule abstract class must be extended by the training system developer to provide feedback information directly to a trainee (e.g. display feedback on the local workstation monitor). In regards to the second mechanism, a Virtual Agent is used to execute a MALLET plan in response to the feedback the coaching module has sent to it. The selected MALLET plan may consist of any valid statements for that domain in order to have the Virtual Agent act as desired. The mechanisms by which these two are accomplished are described below.

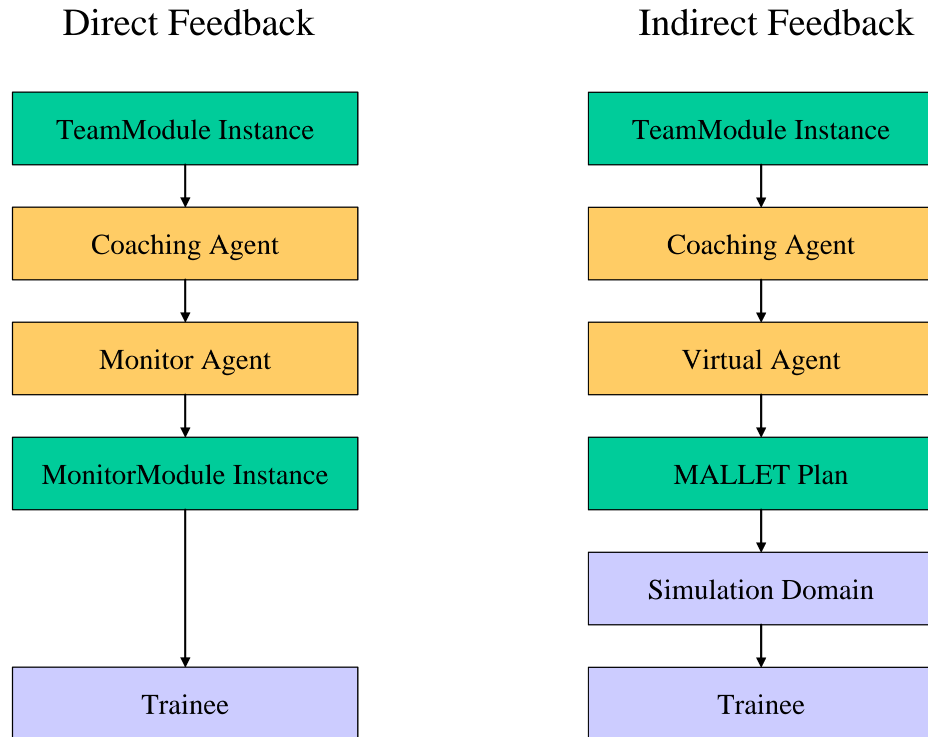

Figure 42: In-session feedback support

In Figure 42 are shown the paths of execution of the Framework for feedback support (direct and indirect). Coaching evaluation is executed within the coaching modules of the Coaching Agent. If trainee feedback is desired then the feedback travels by one of the two mechanisms of either directly to a subordinate module within a Monitor Agent for presentation to a trainee or indirectly by execution of a MALLET plan in a Virtual Agent within the simulation domain. The elements the training system developer must create are shown in green.

Direct feedback to a trainee is intended as intervening coaching response that is executed in-session and provides a direct helping action by the coaching module to a

trainee. Therefore, it is entirely up to the training system developer to determine the nature and mechanism of the direct feedback (if any) through the support of the Framework. Such feedback could be visual displays, generated speech, or any other method that could be invoked by the Framework on the local workstation of a trainee using MonitorModule extension implemented by the training system developer. The training system developer must implement the algorithms as part of one of the extensions to MonitorModule for a trainee to determine the desired feedback and make the necessary calls through the MonitorModule base class to pass the information to a trainee.

Direct feedback through a domain specific MonitorModule implementation is facilitated by the Framework so that the appropriate extension to MonitorModule can be called by a coaching module (TeamModule) from within the Coaching Agent. Through the CoachAgent handle provided by the Coaching Agent, any coaching module can remotely access a Monitor Agent using the getTrainee(name) method. From the Monitor Agent handle the getService(moduleName, service) method (an RMI call) can be invoked by the coaching module. This RMI call connects to the MonitorBasicServices module which resides in the Monitor Agent. The arguments consist of the desired MonitorModule name (matching the return name from getModuleName) and any required arguments concatenated as a string with white space. The desired monitor module will then invoke its own getService(service) method using the passed string. This, in turn, is used by the training system developer to pass the feedback response to the appropriate domain specific MonitorModule implementation.

```
TeamModule: cast3.itt.coach.dynamic.DDDCoach

     getCoach().getTrainee("DM2").getMonitor().
     requestService("cast3.itt.monitor.dynamic.DDDDoma
     in", "Please pay attention to task assignments");
```

```
MonitorModule: cast3.itt.monitor.dynamic.DDDmonitor

     JOptionPane.showMessageDialog(null, message,
     "From coach ", JOptionPane.INFORMATION_MESSAGE);
```

Figure 43: Direct feedback example

In Figure 43 is a direct feedback example showing both sides of the feedback elements that have to be provided by the training system developer. Within the TeamModule extension, the training system developer is able to call upon the desired trainee and MonitorModule with a message for that trainee. Within the Monitor Agent, the MonitorModule extension can then display the string passed to it in a Java message dialog.

Indirect feedback is intended to operate through the mechanism of a fellow team member, in the case of the Framework this fellow team member is a Virtual Agent. The benefit of this mechanism is that it allows the training system developer to provide a coaching response that is natural to the training environment in terms of the presentation to a trainee. This is done through the invocation of a MALLET plan by the chosen Virtual Agent. The key to doing this is that Framework has been designed so that a virtual agent can be directed to add a plan in parallel to the overall plan it is using to

achieve its top level team goal. Effectively, the original top level plan and the "feedback" plan are embedded in a MALLET par structure.

This feedback mechanism is flexible enough to include parameters for the MALLET plan specified by the coaching module. From the CastMonitor handle of the CAST Logger (described in Section 4 and encapsulated in the Coaching Agent), any coaching module can remotely access a Virtual Agent using the getAgent(name) method. From the agent handle the executePlan(planName, args) method (an RMI call) can be invoked. The name of the plan is **planName** and **args** is a Vector of Strings for that plan. The called plan will then generate the behavior desired in the selected Virtual Agent (e.g. execute a helping behavior by the Virtual Agent towards the human trainee).

```
String agentName = "DM0";
String planName = "intervention";
String arg1 = "DM2", arg2 = "assist";
Vector args = new Vector();
args.add(arg1); args.add(arg2);

getCoach().getLogger().getAgent(agentName).
     executePlan(planName,args);
```

Figure 44: Indirect feedback example

In Figure 44 is a simple example of the code required in the desired TeamModule extension to invoke a MALLET plan using Virtual Agent DM0. In this example the plan name is **intervention** and the plan arguments are the trainee to assist, DM2, and an argument **assist**. The **assist** argument in this example simply invokes one path within the **intervention** plan. The training system developer is also required to provide the requisite MALLET plans.

How the training system developer invokes the feedback code (direct or indirect) depends on the path of execution for generating that particular coaching feedback. In the example module, BasicCoach, included with the Framework the executePlan method (indirect) is called by the human trainer pushing a button. The human trainer can select the trainee, plan, and arguments before pushing the button. The training system developer more typically would place the feedback code in their implementation of the execute method of their TeamModule extension as part of the sequence of activities in coaching a trainee by that module.

5.6.2 Post-session Coaching

At the end of the training session, the training environment typically quits but each trainee's tasks are not complete. In many Command and Control environments the trainees must go through a post-session review. The extent of this review may vary from domain to domain, however the Framework simply classifies all such activities as occurring in the post-session phase. At this time the Framework allows the trainer to save the logs of all the agents of the Framework and provides an automated support structure to facilitate the development of automated review tools for the post-session phase.

Post-session coaching and feedback support is initiated by the human trainer (e.g. the person running the session or experiment) or a domain specific coaching module. When the trainer places the Framework into the post-session mode (by the push of the Post Session Review button on the Coaching Agent display under the Generic Coach Tools tab), the Coaching Agent sets the post-session flag to true and invokes all

individual and team assessment modules and all coaching modules. These are all invoked from a single method, postSessionReview, to facilitate the development of an automated coaching module. A domain specific coaching module must provide some mechanism for either detecting when to enter the post session phase or allowing a trainer to initiate such action. Once in the post session phase, an automated domain specific coaching module can invoke postSessionReview within the Coaching Agent to initiate the final run of the assessment modules.

While some of the assessment and coaching module may have been executed during each cycle previously, during this last execution, it is known by all modules that the simulation is over and final results can be calculated. Also, any modules that are to run only during the post-session phase are executed. Each PerformanceModule and TeamModule may (by examining that the postSession flag is true) generate a final summary and/or compilation of results.

These results are intended for direct use of the trainer or automated coaching module only. Separate action (to be discussed below) is required to present ARR information to the trainees. Accordingly, after invocation of all of the assessment and coaching modules, the Coaching Agent instantiates an AfterActionReview object, this displays all PerformanceResults generated by the PerformanceModules and TeamModules on the monitor of the machine running the coaching agent only. In this way, a human trainer may see the results and decide what to do, while an automated coaching module may access the results and decide what to do. Note that even in the case of an automated coaching module, the results are displayed visually (if the

appropriate JPanel has been written) on the machine on which the coaching agent executes. Conceivably, a training system developer could find a way to utilize this.

The textual results and/or visual displays are created during the construction of the AfterActionReview object. This object will automatically call each PerformanceModule and TeamModule and use their PerformanceResult implementation to generate a display. An example of the After Action Review display is shown in Figure 45. As with the in-session display, the basic structure of the display is selection of specific results by tab selection. The textual results are generated automatically by the Framework under a Results Review tab in the display. In addition, each JPanel created by a PerformanceResult has a tab automatically placed in the After Action Review display, which the trainer can use to display the specific results for a trainee. This allows the trainer to view the displays of each result for each individual trainee assessment set of modules and the results of the set of team assessment and coaching modules.

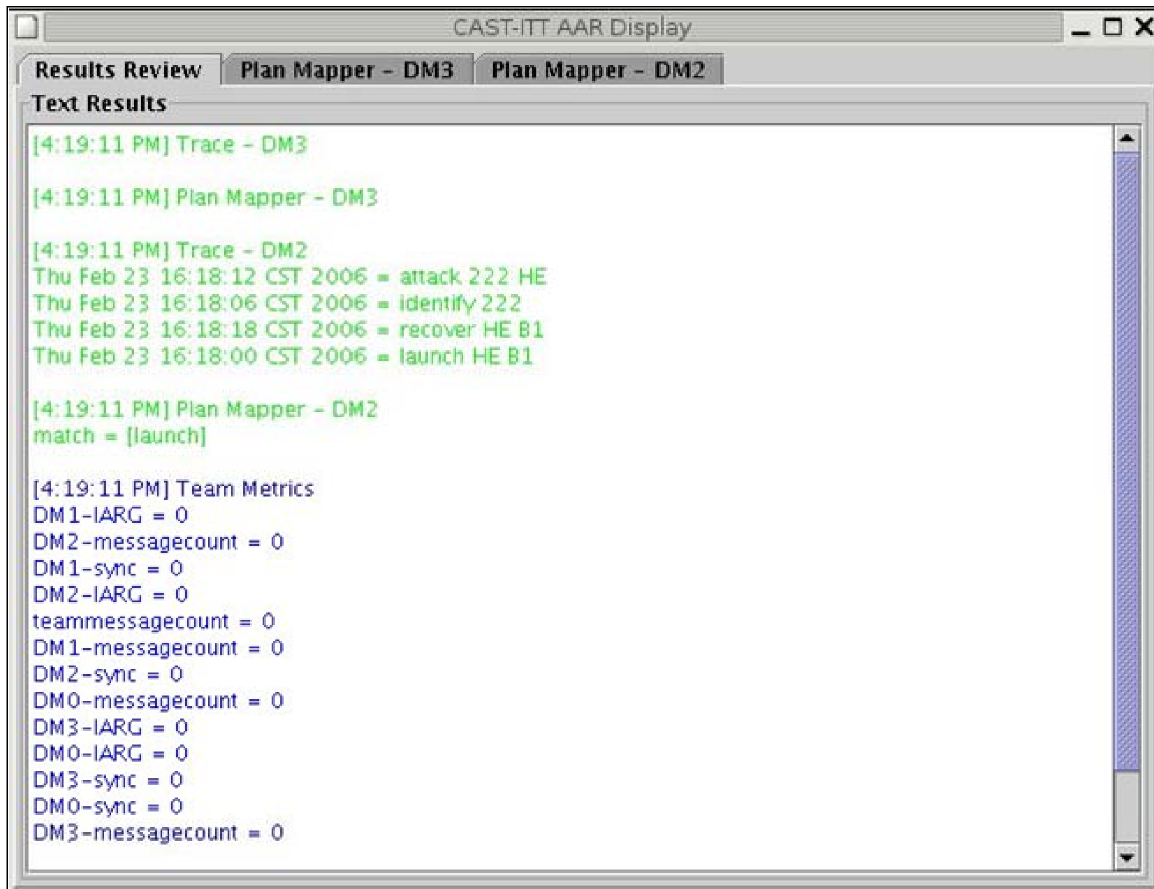

Figure 45: After action review display

Shown in Figure 45 is a sample execution of the AfterActionReview object with two individual assessment results for trainee DM2 (TraineeTrace and PlanMapper) and a team assessment result (TeamMetrics). In this example a four member team is given with DM2 and DM3 as trainees, and DM0 and DM1 as the virtual team members. The textual data is read directly from the associated PerformanceResult object. In the example above the trace is a set of name value pairs in the TraineeTrace result using the date and event (or action taken by a trainee. PlanMapper has no textual component in its associated PerformanceResult object. Instead PlanMapper has generated a display which

is shown by selecting the PlanMapper–DM2 tab (see Figure 40 for an example of such a display). The PlanMapper–DM2 tab provides the visual action trace of trainee DM2 from that individual assessment module. There would be one such tab for each trainee in the scenario being used. The TeamMetrics data is a set of value descriptors and their associated count during the execution of the training session.

The training system developer can produce other forms of displays through his/her design of the JPanels for each PerformanceResult module implemented. The visual component to the post-session execution of the AfterActionReview object is intended for the trainer but other possibilities exist (e.g. for automated environments without a human trainer). The primary purpose of the AfterActionReview object is to automate the post-session execution of the individual and team assessment modules and coaching modules as part of the post-session phase.

5.6.3 Summary of Coaching Support in CAST-ITT

The following interfaces and mechanisms are provided by the Framework.

- Division of coaching into in-session and post-session
- Executable modules to support domain specific coaching
- Support for direct feedback mechanisms through the Monitor Agents
- Support for indirect feedback mechanisms through the Virtual Agents
- Support for visual display of in-session assessment/coaching
- Support for visual display of post-session assessment/coaching results

6. DOMAIN: DISTRIBUTED DYNAMIC DECISION MAKING

For joint human and agent testing, a suitable team training simulation was required. However, we needed more than just a typical training simulation. We also required a research tool that would allow us to manipulate the experimental setup in order to compare the Framework to other learning approaches.

Requirements for the team domain simulator were:

- A software-based Command and Control simulation for use by humans
- An interface for connecting the agents into the simulation
- A suitable cognitive science backing to the simulation to support a real world evaluation

While the first two requirements are from Section 3, the last requirement was desired in order to test the Framework in a real world training environment.

The DDD simulation domain previously discussed in Section 2.2.2 was our choice for testing the Framework. The original DDD code provided no agent interface or external connection for interacting with other tools. It was a self-contained simulation environment. Therefore changes were made to DDD to improve its usefulness as a research tool. In addition, changes were also made to better support the training protocols in pursuit of the research goals. In particular, the cognitive science research goal was to study the training of helping behaviors. These changes were done through performance support tools added to support mission planning and team cooperation during execution.

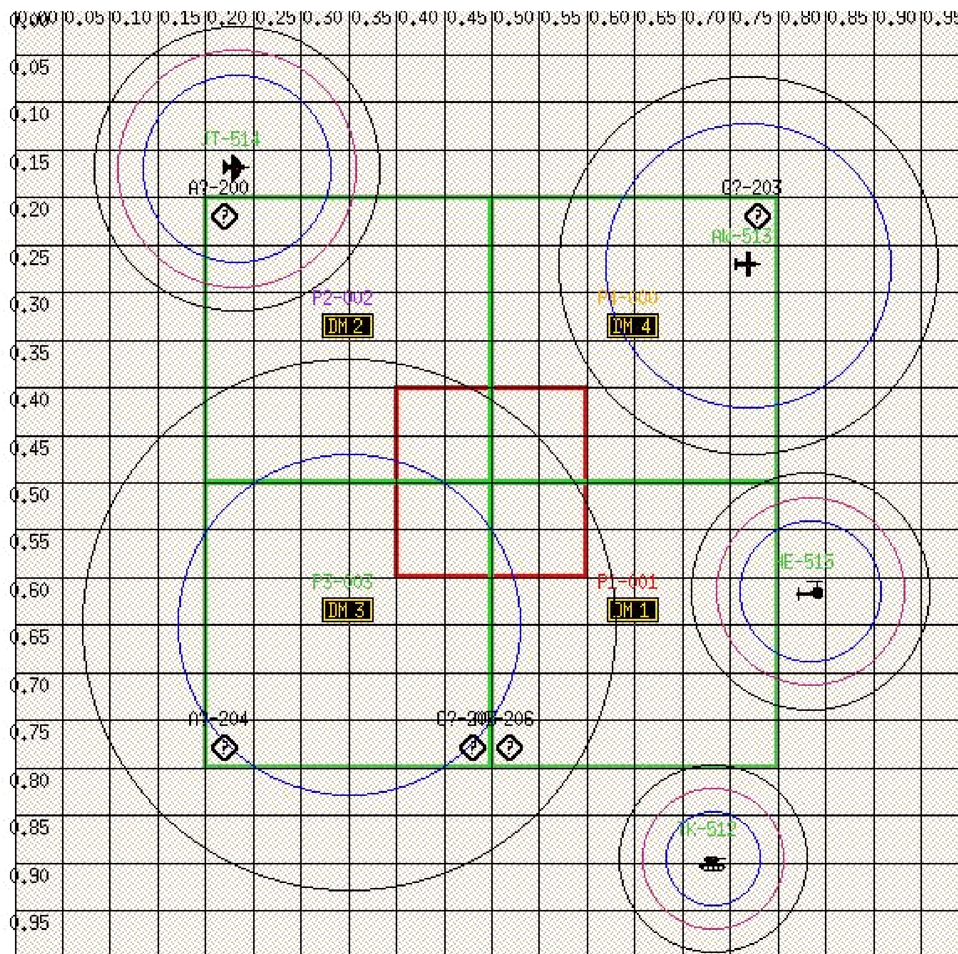

Figure 46: Decision maker screen in DDD

Figure 46 shows the training environment. In this case, the decision maker (DM) has a view of the simulation world and control of a number of vehicles (assets). In the training scenario used in this research, the DM has four assets: an AWACS (AW), a tank (TK), a helicopter (HE), and a jet (JT). These labels identify the various assets; assets have varying capabilities. The AWACS can only detect enemy tracks. The tank is slow but can destroy any track. The helicopter has moderate speed and power. The jet is fast but weak in power. Each DM has a zone, which that DM is responsible for but DMs are

encouraged to cooperate and assist each as required. Each zone has a base from which the assets are rearmed and refueled. A list of the assets is provided in Table 1 below.

Table 1: DM assets

VEHICLE TYPE	FUEL CAPACITY (IN SECONDS)	STRENGTH	SPEED (Simulation units per second)
Tank	480	5	0.0030
Helicopter	240	3	0.0090
Jet	120	1	0.0160
AWACS	360	0	0.0160

During execution of a specific scenario a number of hostile and friendly vehicles will enter the environment. These vehicles are referred to as tracks or tasks. Incoming tracks need to be identified in order to distinguish between hostile tracks and friendly tracks. Hostile tracks need to be destroyed only if they enter the Green Zone. The Red Zone is a high priority zone, which also requires the destruction of enemy tracks. DMs are able to transfer identifications done by themselves to other DMs for identified enemy tracks. In Figure 46 above, the game is configured as a four-player game with each DM given an identical set of vehicles (assets) and a zone to defend. Other configurations are possible with the substitution of a single scenario file. Scenarios may have different tasks, assets, or goals to fulfill the requirements of the needs of the experiments as required by the researcher.

6.1 DDD Performance Support

For the experimental protocols used with the DDD task, two additional performance support tools were added. The addition of the performance support tools was done to provide a closer match between the design of the DDD task and the actual work activities of U.S. Air Force personnel in AWACS aircraft. The first tool supports the planning and execution of the DDD mission. The second tool provides a nonverbal means of exchanging assignment and prioritization information between DMs during mission execution. The tools have the added bonus of providing additional knowledge to the CAST agents as to the intentions of the human team members.

6.1.1 Intel Report and Planning Tool

The intel report and planning tool[8] is a two-step tool that provides a preliminary indication of the number and strength of the enemy tracks expected to appear in each zone across time intervals during the execution of the mission. The first step is done before mission starts as a planning step. The second step is use of the created plan during execution of the mission.

Before the start of the mission, the trainees are provided the opportunity to examine an intelligence report of the expected enemy activities and plan their defense strategies accordingly. The intelligence report will also provide an opportunity for the trainees to recognize times and locations of overload among individual team members and allow them to plan team behaviors to support their fellow team members during those periods.

[8] The concept and development of this tool was done at Wright State University.

Figure 47: Intel report and planning tool

As shown in Figure 47 each DM is able to allocate their assets in response to the intelligence report. Asset allocation can be done for each DM's own zone or within the zone of their fellow DMs. In the current mission sessions, four planning intervals are provided per session for allocation of assets. In the above figure, the intelligence report shows the expected arrival time and strength of the enemy tasks; three planning intervals are visible. The DM in Figure 47 has not started the planning process but can place assets during each time planning interval for each zone.

During execution of the mission, the planning information and intelligence report is provided in a compact form as a memory and communication aide. A condensed visual summary of the above plan allows the DM to see both the DDD window and the plan window during the execution of the mission. The DM still has to manage the launch times and precise locations of each asset during the execution intervals.

The planning tool is used in a team session, allowing team members to plan and record how they expect to help one another during execution.

CAST agents were not involved in this planning process. However, CAST agents may access the planning state as it is encoded as predicates for use in their individual knowledge base.

6.1.2 Task Assignment Panel

The Task Assignment Panel (TAP) has been designed to provide an execution aid to DDD to allow the players to coordinate their activities with minimal voice communication. Minimizing voice communication has the added benefit of allowing agents and humans to cooperate through TAP communication without the use of complex voice recognition and generation tools. In particular, a mechanism was added to allow one team member to assign a track (target) to another team member (or itself), with a visual indication of the assignment on the screen of each trainee.

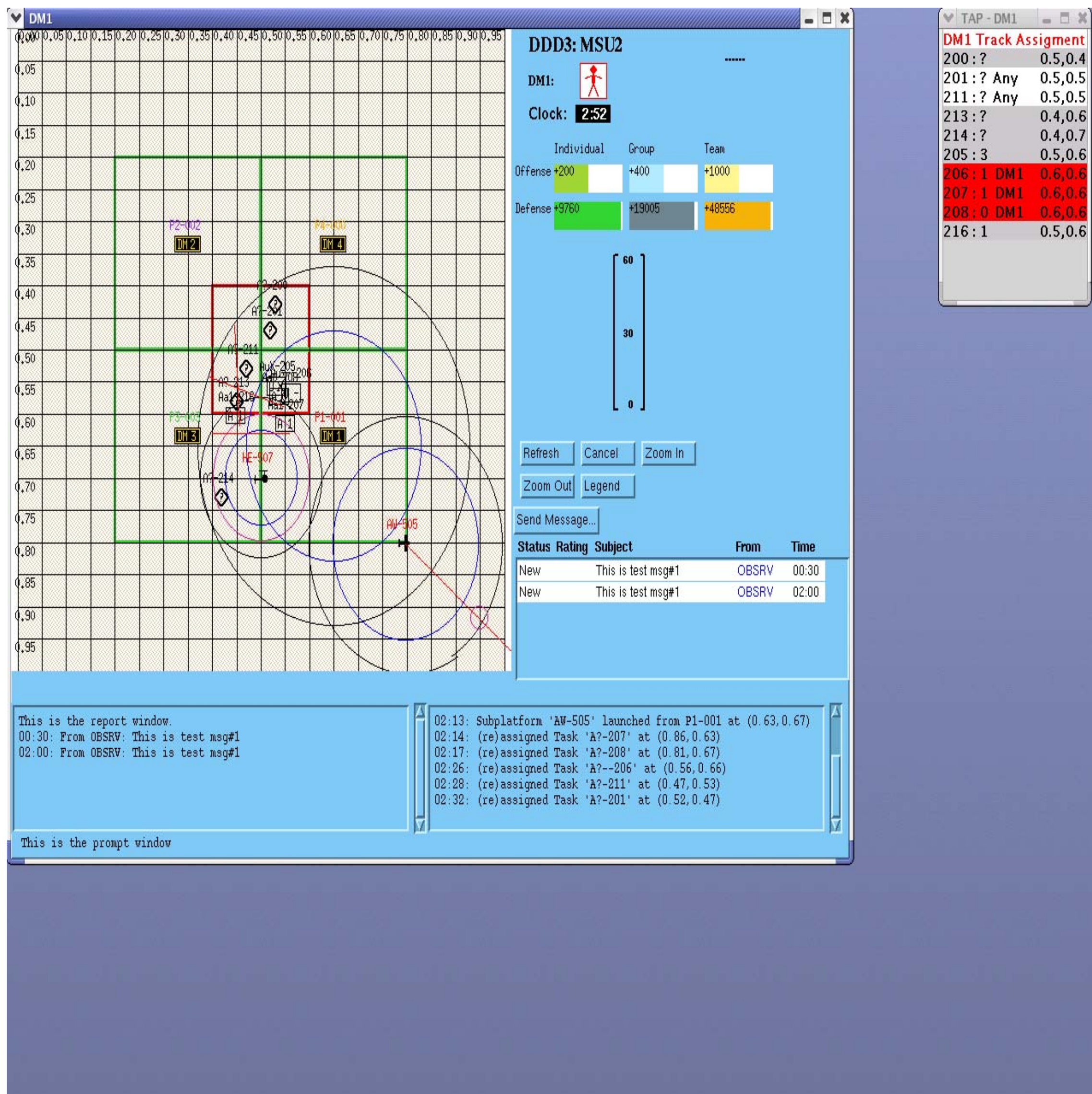

Figure 48: Task assignment panel and DDD

As can be seen in Figure 48 the DM in the DDD simulation has multiple assets and views to manage and multiple tracks that must be attacked. As helping behavior is the subject of the cognitive research, the scenario allows team mates to help in these attacks. However, making attack assignments and displaying who has responsibility for an incoming task is not easily handled in DDD. Therefore, trainees in prior experiments

simply used voice communication and their individual memory to keep awareness of who was attacking what.

Therefore, the TAP was added to provide two additional functions to the DM:

1. Assign responsibility for the destruction of an enemy track
2. Display assignments as a memory aide

The TAP window with auxiliary information and functionality is shown in the upper right hand corner of Figure 48. With the TAP extension, the assignment of a track to a DM is given a color cue that matches the DM's color, both for the enemy track in the DDD display and the TAP window. In addition, the TAP shows text stating the DM to track assignments. DDD has been modified to support the TAP window and allow a trainee to use either the TAP or DDD's primitive mechanisms to do the assignment. The assignment information has become part of the simulation state.

The assignment information is also provided to the CAST agents through the ActorDomain interface (DDDEntity is the domain specific implementation) to DDD. This interface allows agents to assign tracks to human trainees or receive assignments from them.

6.2 CAST and DDD

DDD was designed with the understanding that all team participants would be human. Therefore, the user interface was designed exclusively for human use. In order to integrate CAST-ITT to DDD, the DDD code had to be changed. The DDD code was altered to provide a TCP/IP socket connection to allow external tools to both execute commands in DDD and extract environment knowledge from DDD. Each DDD client

(one for each team member) has such a socket connection. This allowed the DDD software to fulfill the requirements from Section 3.1.

Once the socket interface existed, it was a simple matter to plug DDD into the Framework. A Virtual Agent can act as a DM by connecting to the appropriate socket connection associated with that DM position. The Monitor Agent associated with a trainee uses the appropriate socket connection for the DM position of the trainee being monitored. Each CAST agent (Virtual or Monitor) has access to the same knowledge and commands that the human DM does.

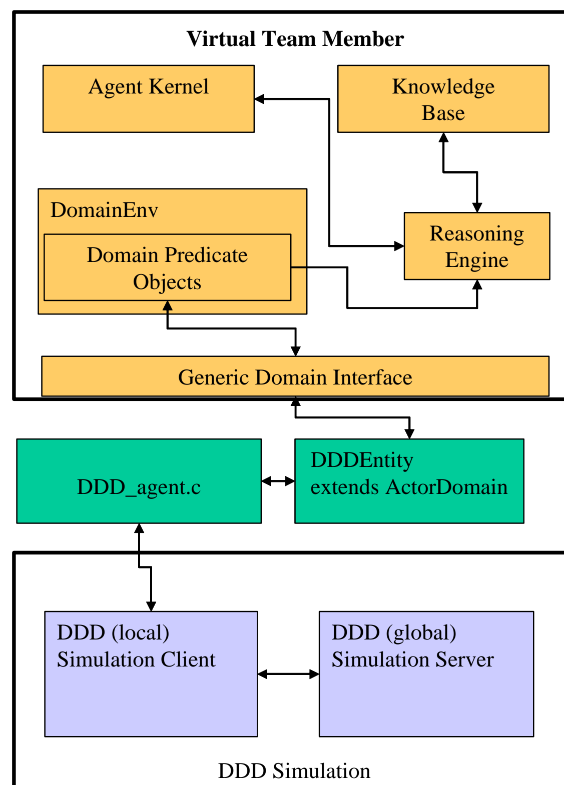

Figure 49: TWP-DDD integration

Figure 49 is a diagram of the integration of CAST with DDD. DDDEntity is the domain specific implementation of ActorDomain in the DDD simulation domain. Each DDD command must be a java method in DDDEntity. Each command, when invoked, is sent to the DDD client as a text string through its TCP/IP socket channel. The command is turned into a C method call that updates the DDD server, which in turn updates the other DDD clients.

Listed below are the commands used in DDD. Where the label asset#, task#, base#, or DM# is used, the label refers to a numerical id for that element. Commands available to the CAST agent in DDD:

- startgame
- pursue task# asset#
- fusion task DM#(n, ALL)
- assign task# DM#
- recover asset# base#
- move asset# x y throttle
- waypoint asset# x y
- stop asset#
- launch asset# base#
- attack task# asset#
- dualattack task# asset# asset#
- transfer asset# DM#
- refuelmove asset# tanker#
- tankrefuel asset# tanker#
- identify task#
- message rating receivers subject body

Since there are no return values, the failure or success of a command is not immediately available to an agent. If such information is needed, the need will be expressed as a condition in some MALLET statement that is dependent upon an environment state (e.g., whether or not an asset has been deployed after a launch

command). As described in Section 5.1.2, conditions involving a state in the environment are transformed into a query to the environmental knowledge in order to find the result of a command or the current simulation state. Such knowledge is stored as domain predicates. For our DDD domain, these domain predicates have the following names and are listed below in the form used to query them:

- asset ?ID ?name ?type ?owner ?x ?y ?vx ?vy ?str ?status ?deploy
- task ?ID ?name ?x ?y ?vx ?vy ?str ?visible
- zone ?ID ?own ?type ?x ?y ?w ?h
- simtime ?time
- score ?DM ?ioff ?idef ?goff ?gdef ?toff ?tdef ?timeRed ?atkOut ?atkWrong ?outOfFuel

The above predicates allow Virtual Agents acting as DMs to manage assets, identify and attack hostile tasks, be aware of their own and other's defense zones, monitor the passage of time, and monitor their individual score and the team score. MALLET plans must use these predicates within conditional checks in order for the agents to act within the DDD domain. For example it takes 10 seconds for an asset to be deployed after a launch command. Before issuing a move command to an asset, the agent's plan must call for a delay until the asset has been deployed. This is done by using the "asset" predicate as part of a precondition on the move sub-plan. Via the mechanisms described in Section 5.1.2, this precondition is queried and the current state of the simulation is found through the "asset" predicates that unify.

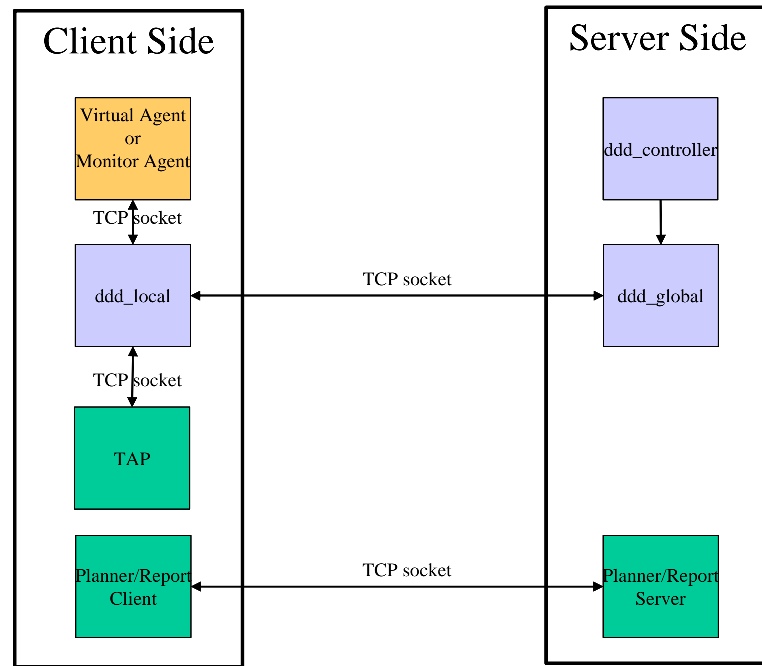

Figure 50: DDD with CAST-ITT

In Figure 50 above is shown the separation of client side (one instance for each player) and the server side processes. The entire TWP-DDD training system is distributed in that each player or agent has his or her own workstation on which to run his or her local DDD client. However, all clients connect to a central server called the DDD global server. The DDD global server executes the simulation. The DDD controller is a visual display for starting and stopping the simulation. DDD local is the visual display for the player (DM).

The processes (client and server) include the two performance support tools added to the DDD task to assist the trainees. The Planner/Report processes are run

before the DDD training session is begun in a planning mode. The planning information derived from the planning step is made available to the human DMs in the form of a Planner/Report client screen that is read only and is available during the training session. This planning information is provided to the agents as predicates. As stated before the TAP client has both a visual display and provides the agents a list of predicates to represent its state.

6.3 MALLET and DDD

The general objective in developing the MALLET plans is to allow CAST agents to act in a number of similar scenarios without changes to the CAST agent or MALLET plans. The behavior of agents acting as DMs are specified in the MALLET plans. The MALLET plans for a CAST agent are intended to be flexible enough to be reused for similar scenarios that differ only in the timing and paths of the incoming tasks. In order to plan for and manage overloaded situations that appear in these scenarios expert human DMs devised a strategy of using a lead DM to assist in managing assignments in support of an overloaded DM. The position of lead DM rotated depending on which DM had the least load during a wave. This same strategy is supported by the MALLET plans devised for the Virtual Agents acting as partner DMs.

The basic structure of the MALLET plans developed for use as partner agents is shown in Figure 51 below. Each Virtual Agent acting as a DM in DDD starts its own copy of the *execute* plan and continues this plan until the training scenario is completed. Inside the *execute* plan is a **Par** construct which starts parallel sub-plans for the management of each asset controlled by that DM. There are also several sub-plans for

management of individual tasks for each agent such as launching assets, recovering assets low on fuel, identifying unknown tracks, and updating the DDD database for other DMS as to that identification. The *dispatcher* sub-plan manages the issuance of commands by the Virtual Agent so as to match human performance timing in issuing commands.

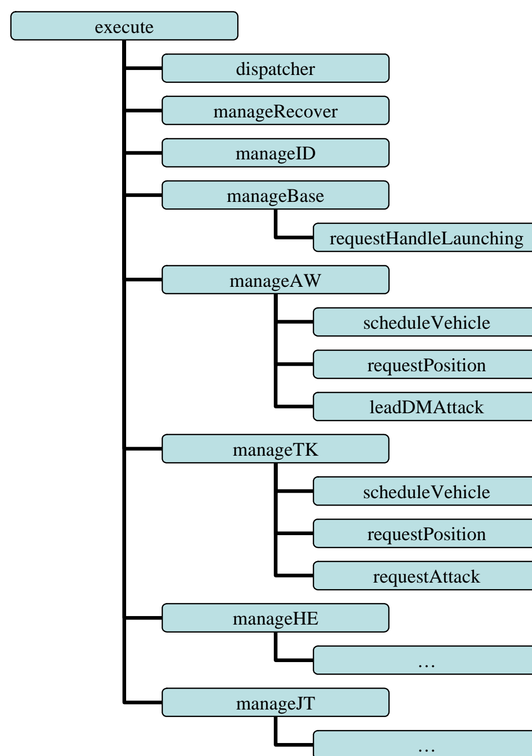

Figure 51: MALLET plans in DDD

Each *manageXX* (where XX equals an asset) plan invokes several sub-plans for controlling that asset. Each asset has limited resources and fuel and has to be coordinated

in its launching, attacks, and rearming in conjunction with other assets managed by that DM and assets managed by other DMs.

One of the benefits of the DDD Planning Tool was to provide assignments to the Virtual Agents. Virtual Agents did not participate in the planning process for the allocation of the DMs' assets in support of other DMs. Nevertheless, each Virtual Agent acted in the role of a DM during the scenario execution. Therefore, the Virtual Agent had to be aware of the plans made by its human partners in the team. MALLET operators were created by the training system developer to allow the Virtual Agents to parse a planning file generated from the Planning Tool. The MALLET plans created for the Virtual Agents then used this information to schedule asset launches and determine locations to which assets should be sent.

The Task Assignment Panel (TAP, the second performance support tool) was used to overcome communication difficulties in managing assignments of actual attacks on hostile tracks. The *requestAttack* sub-plans were used to monitor the TAP in assignments of hostile tracks. The Virtual Agent could then execute an attack if requested.

6.4 Agent-Human Communications in DDD

While CAST implements communications between agents, the communications of agents to/from humans required an integration of domain specific communication channels through CAST-ITT. In the DDD domain, the primary type of communications during execution was the assignment of hostile tracks between team members. To reduce the need for voice communications, the performance support tool TAP was introduced to

the DDD task. The TAP allowed assignments between team members to be transmitted and received electronically.

From the MALLET perspective of receiving an attack assignment, an assignment predicate is created that lists the task id, location, and the decision maker (DM) to whom that task was assigned. The Virtual Agents can include the assignment predicate in conditions to determine if any tracks are assigned to them. The Virtual Agents can also issue an assignment command that places the assignment in the TAP tool (which, in turn, places the assignment predicate in the knowledge base of all agents) and color-codes the track on the DDD window for use by the human team members.

The other primary form of team communications in DDD was the creation and use of a placement plan. The Virtual Agents do not participate in the planning phase, so the planning phase activities and communications were not implemented for the Virtual Agents. However, an implicit form of communications occurs between the trainees and the Virtual Agents during execution. The Virtual Agents are capable of reading the placement plan and incorporating the plan into the agents' actions during the course of a training session.

Taken together, these two means of communications eliminated the need for natural language recognition as part of the current human/agent communications in the DDD domain.

6.5 Summary

The Framework was successfully integrated with the DDD simulation and a series of experiments was run by the researchers at Wright State University (Volz et al., 2005). In

the process of using the Framework to build a training system for the experimentation described above, the two principal issues had to be addressed. These issues were the development of the MALLET plans and the modification of the underlying CAST architecture for performance reasons. The development of the MALLET plans was an ongoing process in response to performance and feedback from the cognitive psychology experimenters. The MALLET developers had to ensure the naturalness of the behavior of the agents in order to elicit a better learning experience from the experiments (Srivathsan, 2005). For the CAST architecture, the code for connecting DDD and CAST was completed very early in process and did not require major changes later on. However, the older nature of the computing hardware used required that we get the best possible performance from the CAST architecture and so work was done in parallel with the MALLET developments to improve the performance of CAST.

In summary, the use of the Framework proceeded as expected with no major difficulties and led to a successful experimental use.

7. VALIDATION OF THE FRAMEWORK

The focus of this dissertation has been on creating a team training framework that is generic and flexible enough to facilitate the creation of a wide variety of team training systems with various coaching technologies that can operate in a number of training domains. Being able to test across a large set of training systems and training domains is not feasible due to the resources and time required. Instead, the Framework has primarily been validated against a single training domain, DDD, and a team training protocol used in an actual experiment of sufficient complexity to exercise key portions of the Framework. In addition, a follow on experiment using DDD has been run to test features of the Framework not captured by the cognitive psychology experiment, and an additional training system for coaching based on the Framework is being developed by a researcher at another university.

Validation is the process of ensuring that the design of the system solves the desired problem. Unlike verification, validation is used to determine whether the designed system is useful to the target user. Validity is also used to decide whether the system provides answers that "make sense" or are useful to any available human experts. In the case of the Framework, validation involves addressing the question of whether the Framework is a useful solution for creating a team training environment. In our case, we consider two interpretations of "useful." One, obviously, is whether or not the use of the training system developed achieved its training goals. The second is the usability of the Framework for building the specific training system and the ease with which the training system may be constructed. For validation, the following objectives are addressed.

- Build and demonstrate use of a system for joint human/agent teamwork built using the Framework in a training environment
- Demonstrate proactive information exchange between humans and agents
- Demonstrate the use of the Framework in monitoring, assessment, and coaching

These objectives provide a basic validation of the Framework and lay a foundation for more extensive validation in the future.

In order to exercise the capabilities of the Framework the validation has been done in stages. The first objective was covered as part of the TWP-DDD experiments done by the MURI group from Texas A&M University, Wright State University, and Pennsylvania State University. To complete validation of the Framework a follow on experiment has been run. This experiment addressed the validation of the generic components of the Framework by using the existing test domain DDD. In addition, a set of assessment and coaching modules devised specifically for helping behaviors in DDD via an event based coaching system is being created by Cong Chen at Pennsylvania State University, and will be described briefly.

7.1 Validating the Framework in a Team Training System: TWP-DDD

For the cognitive psychology DDD experiments on teamwork, the MURI group desired to make a comparison between an all-human team and a mixed human/agent team in training helping behaviors. In contrast, the objectives of this validation study were to demonstrate the use of the Framework to build a suitable training system and the operation of the system built for such training purposes. In particular, in addition to the

successful use of the training system for the cognitive psychology experiments, we are interested in the ease with which the training system could be built and any problems that arose during its construction. The issue of interest for validation includes such things as the integration of a training domain into the Framework and the creation of a training system that employs Virtual Agents acting as team members and able to execute MALLET plans for the training experiment with sufficient efficiency.

We consider the creation of MALLET plans for Virtual Agents so that they acted in cooperation with human trainees also acting as team members as distinct from the creation of a training system using the Framework. In general, one could write many different MALLET plans for different experiments using the same training system. Thus, the creation of the plans is not itself a subject of validation, though the plans are necessary to perform the validation. The creation of the MALLET plans is part of the domain and experiment specific work that must be done to utilize a training system built with the Framework. The only intersection between the MALLET plan development and the Framework validation is in the area of inclusion of domain specific utility operators and tools; the ease with which such operators can be included is of concern. In order to check the efficiency of the training system built using the Framework, we ran experiments on the capabilities and performance of the CAST Agents acting as an all agent team.

Following is a list of the specific characteristics of the Framework that were tested via the cognitive psychology DDD experiment:

- Integration of the DDD domain into a system built on the Framework.

    - Command execution by Virtual Agents
    - Sensing by Virtual Agents
    - Integration of experiment and domain specific operators and tools
- Communications between DDD team members (agents and/or human)
- Execution of MALLET plans by Virtual Agents in the DDD domain
- Limited monitoring of the DDD domain (Virtual Agents only)

These features of the Framework were described in the first three sections of section 5. What was not covered in the cognitive science DDD experiments was the full use of the Monitor Agents and Coaching Agent and the interfaces provided through these agents by the Framework. The reason these were not covered is that they were not needed for the cognitive psychology DDD teamwork experiment.

7.1.1 Description of DDD Helping Behavior Scenario and Protocol

The objective of the cognitive psychology experiments was to compare the performance of human trainees trained using agent partners versus those trained using human partners. Helping behavior among decision makers (team members) in the DDD team was based on coordination of the matching of asset resources versus hostile tracks across neighboring zones of control. From the perspective of the Framework these experiments also provided an opportunity to demonstrate the underlying model of teamwork (CAST) used in order to support information exchange between human trainees and Virtual Agents acting as team members (see Section 6 for details).

The experiment was based on the use of the AIM (Active Interlocked Modeling) protocol (Shebilske et al., 1992). In the AIM protocol, trainees work together with each

trainee performing a specific subtask and rotating between subtasks. Through rotation each trainee is able to learn and execute all subtasks. The experiment consisted of three different trial conditions. The first condition was the control condition, called Individual, which consisted of four human trainees acting as a team without any further training protocol. The Partner Agent and Human Partner conditions split the four original roles into eight roles (four pairs, one pair per zone) in order to reduce the work load and support a focus on teamwork. The second and third conditions consisted of a set of training sessions with the eight team members, followed by a set of testing sessions with four member teams, as in the control condition (Individual). The difference between the Partner Agent and Human Partner conditions was that for the Partner Agent, one member of each pair during training was an intelligent agent, whereas in the Human Agent condition, all team members were human trainees.

There were four kinds of trials, Baseline (B), Practice (P), Assessment (A), and Transfer (T). Each trial had a 5-min. Planning session, a 15-min. Mission, and a 5-min. Debrief session. The sequence of events was: instructions (2.5 hrs.), B, P, A, P, A, P, A, T, T (4.5 hrs). The dependent variables were measures of teamwork, attack assists, and communications of task IDs, as well as measures of team mission performance, defensive scores and offensive scores. Scores are points which are added or subtracted based on specific events in the simulation such as destruction of hostile tasks, incorrect destruction of friendly tasks, time of hostile task in the DM zones, and other metrics that matched important mission objectives.

7.1.2 Results of Using Training System Developed in Experiment

From the perspective of the psychology experiments the training system developed utilizing the Framework worked well in that the performance of trainees with agent training partners matched that of trainees with human training partners. The experiments were able to be completed by using the training system built using the Framework to provide integration with DDD and the Virtual Agents acting as partners for the AIM component of the experiments.

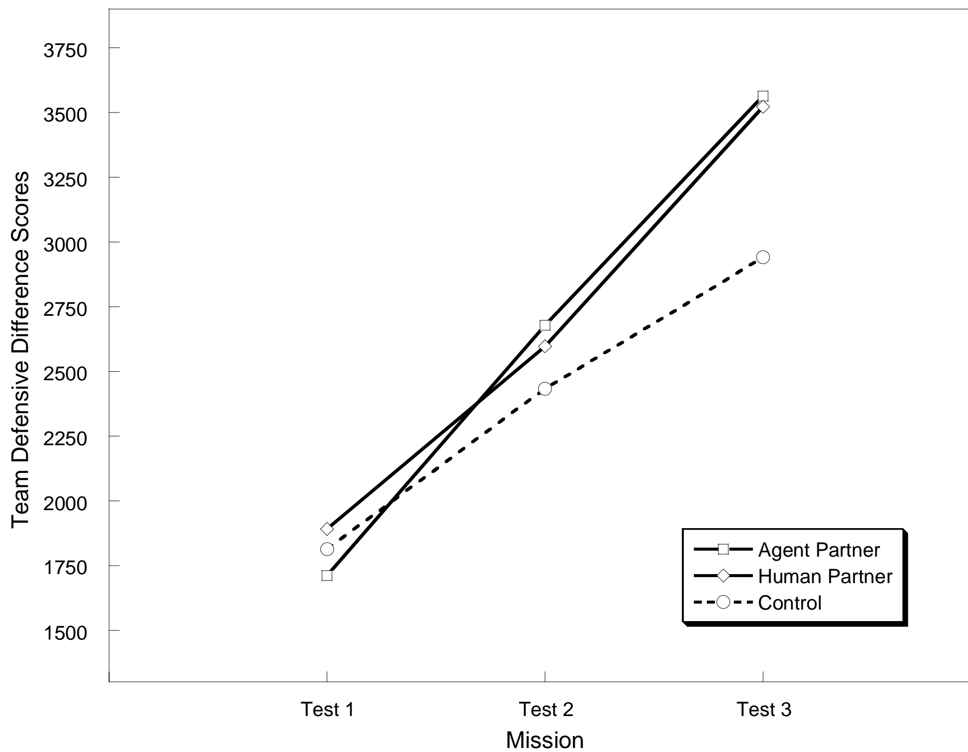

Figure 52: Performance of partners versus no partner[9]

[9] This figure comes from the final project report.

Figure 52 illustrates the performance variations of the three trial conditions. In the figure is shown the Team Defensive scores versus the three sessions run for each condition. As shown, the Partner Agent trials scored better than the Individual (Control) trials and matched the performance of the Human Partner trials. This performance evaluation was the intended objective of these experiments as published in the final project report (Volz et al., 2005).

7.1.3 Validation of Framework Use in the Above Experiment

From the perspective of validating the Framework, the experiments demonstrate that the Framework can be used to build a working training system as intended. The Framework was instrumental in developing the experiment in the amount of time available. In this subsection, we discuss the use of the Framework with respect to each of the characteristics mentioned in the introduction to the section.

The implementation of the first two features for DDD, command execution and sensing by Virtual Agents, was discussed in Section 6.2 under CAST and DDD integration. For validation of command execution and sensing, the first question to ask is if the agents were able to perform all the commands that a human trainee could in the DDD simulation. The answer is yes. Aside from the validation from use in the MURI experiment, each command was tested individually. More importantly, the design of the experiment dictated that capabilities beyond the standard DDD simulation were needed (the intelligence report, the generation of the placement plan and the TAP). It was important to evaluate the ease with which these could be integrated (as distinct from the tool creation) into a training system built with the Framework. In addition, it was found

that a number of cognitive tasks that humans would do during execution of a scenario, e.g., find the track closest to an asset, were more readily handled through creation of experiment and domain specific MALLET operators than direct expression in MALLET. The integration of such operators into the system built with the Framework was also of interest. The integration of these features in DDD was trivial in comparison to the effort to build the tools and operators themselves. While quantitative measurement of the effort is not available, we do know that once the procedural attachments and efficiency improvements were made to the Framework, the focus of almost all of the discussion and work was on the tools themselves, with their integration into the training system via Framework provided interfaces being treated almost as a plug in.

As examples of the MALLET operators added during creation of the training system, we consider Virtual Agent/human communications. Specific needs occurred both as a result of the placement plan generation (performed only by the human trainees) and during execution of the simulation. At the start of a scenario simulation execution, the agents needed to become aware of the asset assignments expressed in the placement plan so that they could act in accordance with the placement plan. For the agents, a MALLET operator was written that converted the placement plan into predicates that could be used in MALLET plan execution. The MALLET plan for the agents, then needed to invoke this operator as their execution started. In the case of the placement plan, the operator named partitiontaskwindows reads an ASCII file created by the trainees using the planning software. This operator then creates a list of predicates of the form (planIntNumVeh ?timePeriod ?anyDestZone ?anyAW ?anyJT ?anyHE ?anyTK)

and inserts them into the agent's knowledge base for use in planning what assets to launch and when to launch them.

The primary need for communication during the scenario execution was the need to inform team members of specific attack assignments generated real time by the lead DM (introduced in Section 6.3) for other DMs. As specific attack assignments were generated by trainees acting as lead DM during execution, they also became predicates for use by the agents. In the reverse (lead DM agent to human DM), the color coding of the assignment of attack tasks and the use of the TAP provided the communication from agent to human. A more in-depth discussion of the TAP is in Section 6.4. In order to integrate the TAP into the Framework two interfaces were used. First, the **assign** command was created by the training system developer for use by the Virtual Agents and for monitoring by the Monitor Agent. Second, the results of the **assign** command were made visible to the agents by adding a domain sense predicate.

More generally, the MALLET[10] plans were the most complex aspect of the use of the training system built using the Framework and consumed the bulk of the time involved in creating the software needed for the experiment. However, the construction of these MALLET plans was not part of the development of the training system, per se. They were part of the preparation for use of the training system for a specific training objective with a specific training scenario. In other words, the bulk of the work was on using the training system developed with the Framework, not on building the training system itself.

---

[10] See Section 6 for a detailed description of the design of the MALLET plans.

From the perspective of the Framework, the key issue to note is that the needs for additional tools and operators were derived from the cognitive task analysis performed by the psychologists and readily integrated into a training system built using the Framework either through writing a MALLET operator which needed minimal knowledge of the Framework (only how to insert a predicate into the knowledge base), or the use of domain commands. This demonstrated both the flexibility of domain specific features that could be added to a training system built using the Framework and the minimal knowledge of the Framework and ease with which such features could be integrated.

While a few bug fixes and performance enhancements were made during the construction of the training system, no need to change any of the interfaces the Framework provides were encountered. The Framework seemed robust in its capability to support the integration of the simulation domain and the creation of a specific training system.

The use of the IARG was not a consideration during the development of the agents, as it was not the focus of the cognitive psychology experiment. Further the Monitor Agents were not used in the DDD experiments. The additional monitoring information that could be provided by the Monitor Agents was not utilized as the experimenters focused on the use of the in game scoring metrics of DDD. Instead limited monitoring of the DDD domain was provided by the Virtual Agents as they recorded their own actions and scores for use by the experimenters in evaluating the performance of the trainees and the performance of the Virtual Agent team members.

7.2 Validating Additional Features of the Framework

In this section we cover those components of the Framework not tested in the original DDD experiments. We do this by enabling those components of the Framework not originally used and by extending the MALLET plans used in the previous DDD experiment to exercise the proactive information exchange component of CAST not exploited previously.

In this validation testing we focus on all three classes of agents that are available to a training system developer through the Framework. These agents in turn utilize the various interfaces that the Framework provides. The agents in question are the Virtual Agents, Monitor Agents, and Coaching Agent. The proactive information exchange component of CAST exists within the Virtual Agents but was not utilized in the previous experiments. The Monitor Agents and the Coaching Agent were not used in the original DDD helping behavior experiments.

Therefore, a follow on experiment was created to validate these components of the Framework. In this section we discuss this experiment and how validation of the Framework occurred. First, we describe the changes made for the validation experiments. Second, we go through what was tested for validation purposes. Third, we describe what the results of those tests were.

7.2.1 Modified Experiment

For the modified DDD experiment, we used a similar setup to the control setup in the DDD experiments which was a four player team. However we changed the team of four human players to a team of a single human trainee and the trainee's associated Monitor

Agent, three Virtual Agents, and the Coaching Agent. We also modified the MALLET plans to support information exchange. The normal four player game runs for a full session of 15 minutes. However for these trials we wished to match the length of the partner trials which were a 7.5 minute session each time.

For running the validation experiments, the human trainee did two sets of tests. In the first set, the trainee (DM1) acts as supporting DM in the first half of the session and as an overloaded DM in the second half. In the second set the trainee (DM4) takes on the role of lead DM for the first half and acts as supporting DM in the second half of the session and. In a 7.5 minute session there are two waves of hostile tracks, the first wave is against DM3 and the second wave is against DM1. Having the trainee act as both the lead DM and as the overloaded DM allowed for the trials to test both the sending and receiving of the proactive information from the perspective of a human trainee.

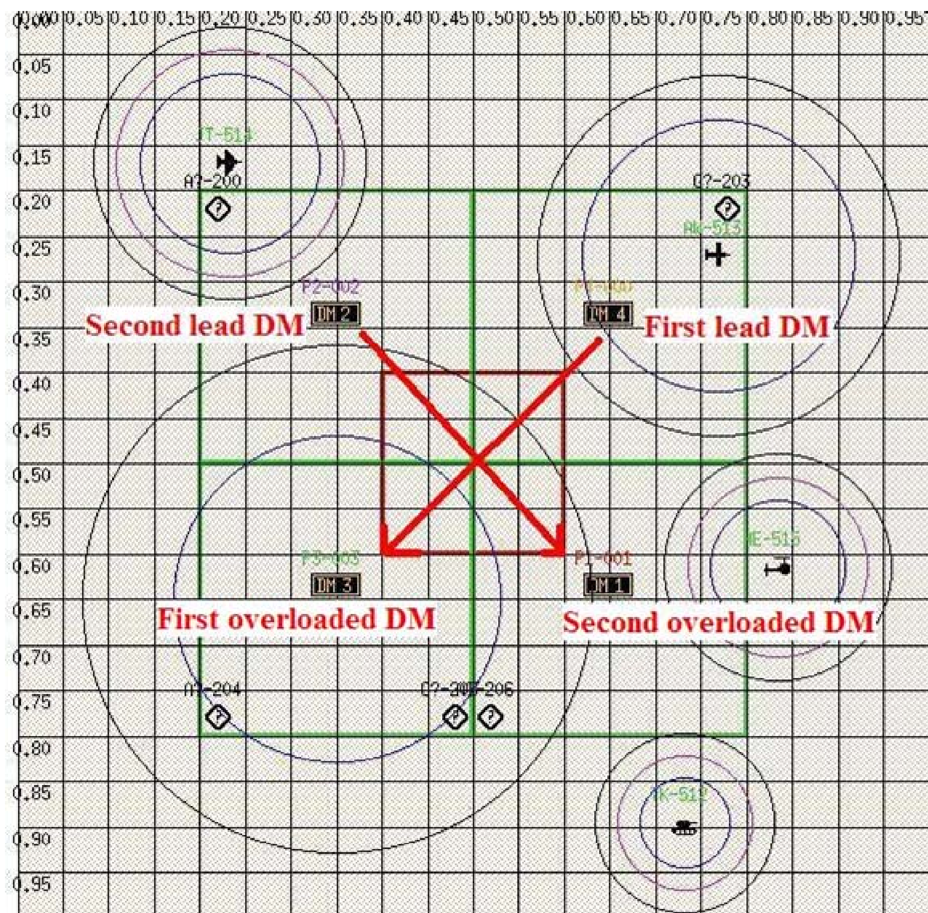

Figure 53: Lead DMs in session

In Figure 53 is shown the geographical location of those DMs. Also shown are the roles of the DMs in the two waves of the experiment. In the first wave DM4 provides assignment assistance for the overloaded DM3. In the second wave DM2 provides assignment assistance for the overloaded DM1.

The experiments were run on a single laptop PC in which the single trainee was able to use his local DDD display. In the rest of this section we will discuss the changes made to the MALLET plans and the experimental setup of the training system.

7.2.1.1 Proactive Information Exchange

Both the original DDD helping behavior experiments and the validation experiments were based on the passing of detection information revealed about identified hostile tracks from the detector to the assigned attacker. This is done by the lead DM by identifying and assigning hostile tracks to other DMs that had assets in that zone. In proactive information exchange terms, the lead DM is the provider of information that another team member (the needer) requires in order to achieve their part in the team plan. MALLET was designed to support the automation of the passing of this kind of information.

Therefore to further explore the use of proactive information exchange in agent/human communications and teamwork we redesigned the MALLET plans by adding preconditions and effects to the MALLET operators that handled the use of the TAP (described in Section 6.1.2) in assigning hostile tasks to a DM which would then attack the assigned hostile track. In Figure 54 are shown the specifications of the modified operators. The original operators did not have the effects or preconditions. This simple change now allows the IARG algorithm in CAST to detect and generate information exchange concerning who generates assignments and who requires such assignments.

```
 (ioper assign (?trackID ?DMNumber)
     (effects (assigned ?trackID ?DMNumber))
)

(ioper attack (?trackID ?vehID)
     (pre-cond (assigned ?trackID ?DMNumber))
)
```

Figure 54: Modified operators in DDD

The other change made to the MALLET plans was to ensure that all DMs were identified as being in a single team and that the team was responsible for all the plans instead of just a single agent executing its own instantiation of the plans. These changes are shown below in Figure 55.

Old start of an agent's top level plan

```
(start DM4 (execute DM4))
```

New start of an agent's top level plan as member of a team

```
(team ddd (DM1 DM2 DM3 DM4))
(start ddd (root))

(plan root ()
  (pre-cond (self ?self))
  (process
      (execute ?self)
))
```

Figure 55: Starting a plan as a team

From the perspective of a human trainee, what these changes illustrate is the requirement to inform other team members of which team member has been assigned to attack a hostile track. The original DDD software provided very limited support for passing this information (i.e. a cumbersome dialog interface and no visual cues). The TWP DDD variant added color cues to the assignment information and an easier to use interface. However, these do not interface well with the kinds of proactive information exchange supported by CAST. Recall that the capture command capability that is required as part of interfacing a domain to a system built with the Framework informs the Monitor Agent of every domain command that a human executes. Also, every domain command is also represented as a MALLET operator so that agents can execute them. Therefore to support agent/human communication, the Monitor Agent identifies the trainee's execution of operators (such as the **assign** command) with effects attached to them and generates the necessary information as required to support seamless interaction between the Virtual Agents executing MALLET plans and the human trainee. The reverse is true in that the Monitor Agent can detect that the assignment information was provided to the human trainee by a provider (e.g. the lead DM) before a trainee executes the attack by using the precondition attached to the **attack** command.

7.2.2 Validation of Generic Monitoring, Assessment and Coaching

For the additional experiment, the configuration file was changed to enable the Monitor Agent (a trivial change in the XML file), and to start the Coaching Agent instead of the CAST Logger (a diagnostic and management tool).

For monitoring, the objective was to validate that the Monitor Agent captured and transferred trainee actions to the Coach Agent. The Monitor Agent also needed to detect and record identified information exchanges based on the **attack** and **assign** commands using the IARG algorithm. In particular we refer to the support of the passing of **assert** predicates from agent to human as part of the generation of proactive information exchange. Such exchanges manifest themselves in the form of an **assert** predicate sent from one agent to another agent (or in this case a human trainee). In the default mode of operation of the Framework (detailed in Section 5.2.2) such predicates are displayed in a Java dialog to be seen by the human trainee. For human to agent proactive information exchanges, if the human trainee did the assignment, the assignment effects were forwarded to the relevant team members by the Monitor Agent to allow the Virtual Agents to continue functioning as if a human trainee were another CAST agent. For validation of the proactive information exchange we looked at the predicates and IARG messages generated by the team during the execution of the scenario. Each attack of a track by a team member should have the appropriate assignment of that team member to the track provided through proactive information exchange.

The objective for validating the assessment components of the Framework focused on the generic modules already incorporated into the Framework. These are the PlanMapper and TeamMetrics modules. As a reminder these modules respectively provide a visual and recorded trace of each trainee's actions and a count of the individual

communications (a simple message count) and identified information exchanges (through IARG) within the team.

The objective for validating the coaching components of the Framework focused on the generic components of the Coaching Agent and their execution during both the in-session and post-session phases. These generic components are the BasicCoach module and the AfterActionReview display. The BasicCoach module provides access to a trainee through simple dialogs and has the button for a trainer to start the post-session review. The AfterActionReview display executes the post-session mode and shows the results of the post-session review. Additionally, validation included the use of the graphical interfaces in supporting a human coach. In particular, graphical displays are available for each Virtual Agent, each Monitor Agent, and the Coaching Agent.

For validation of the overall framework we ensured that all components (agents and interfaces) were exercised and functioned as described in Section 5. Even though we did not have domain specific coaching modules the generic assessment and coaching modules do use the Framework interfaces that the domain specific modules would use, and hence allowed us to do this.

7.2.3 Validation Experiments and Results

In these tests we were looking at how the Framework performed versus the psychological experiments which looked at the performance of the trainee. Therefore, the trainee used was an expert in playing the scenario and was instructed to follow the same procedures and protocols that the Virtual Agents were following.

The Framework, itself, generated six logs in executing this experiment. These were the three Virtual Agent logs, the Monitor Agent log, the Trainee Assessment log, and the After Action Review log. These logs recorded the information generated by the generic assessment and coaching components of the Framework.

Domain specific information can be part of these logs but since no domain specific modules existed for DDD in the Framework no such information was recorded. However, the domain specific scores were recorded as the scores were available to the CAST agents as a domain predicate. Additionally the **assign** predicates generated as part of the proactive information flow were automatically recorded.

The Framework was able to monitor the trainee's receiving and sending (through the Monitor Agent) of assignment information and the generation of the information flows by the Virtual Agents for the assignment information as expected. The agents of the Framework also performed as expected in monitoring and recording the generated data from DDD.

Table 2: Attack assignments with trainee as DM1

Overloaded	First wave					Second wave				
Trials	1	2	3	4	5	1	2	3	4	5
DM1	0	0	0	0	0	4	4	5	3	2
DM2	0	0	0	0	0	0	1	0	0	0
DM3	0	0	0	0	0	0	0	0	0	0
DM4	5	5	5	4	5	0	0	0	0	0

In Table 2, extracted from the four virtual and monitor agent logs, are the totals for the attack assignments sent by the assigning DMs to other DMs in the trials. These

assignments indicate that the virtual team members sent messages generated by the IARG algorithm. Also shown are the attack assignments made by the human trainee (highlighted in grey). For both tables the total of ten trials is shown divided into the two waves that occur in each trial. In Table 2, the trainee is the overloaded DM for the second wave. In Table 3, the trainee is the lead DM for the first wave. An observation to be made about the first five trials with the trainee as DM1 was that the trainee took over the lead DM position for the second wave from the DM2 agent that was supposed to be acting as lead DM. For the second set of trials with the trainee as DM4 followed the protocols more closely except for the fourth trial in which the trainee did not assign as much.

Table 3: Attack assignments with trainee as DM4

Lead	First wave					Second wave				
Trials	1	2	3	4	5	1	2	3	4	5
DM1	0	0	0	0	0	0	0	0	0	0
DM2	0	0	0	0	0	2	2	2	2	2
DM3	0	0	0	0	0	0	0	0	0	0
DM4	5	5	5	1	3	1	0	1	1	0

As shown in the above two tables the information exchanges are a crucial part of the functioning of the DDD team. The domain specific performance of the team depends upon the timely identification and assignment for attack of enemy targets. It is important to recognize that the virtual and monitor agent logs demonstrate the capturing and storage of this domain specific information exchange in a generic manner.

```
8246    1141920273578  FINE   TraineeViews   ActionPerformed  launch 506 1
8495    1141920286425  FINE   TraineeViews   ActionPerformed  move 506 0.4464 0.5431 1.00
8570    1141920290181  FINE   TraineeViews   ActionPerformed  launch 505 1
8689    1141920296407  FINE   TraineeViews   ActionPerformed  attack 217 507
8844    1141920304435  FINE   TraineeViews   ActionPerformed  attack 216 506
```

Figure 56: Sample monitor agent log

Another example of the Monitor Agent log of a trainee's sequence of action is shown in Figure 56. In this sequence of action covering less than a minute of real time the trainee manages three different assets (505, 506, and 507) and attacks two hostile tracks (217 and 216). The other logs follow the same file format.

For assessment support there are two generic modules, PlanMapper (one per trainee), and TeamMetrics (one per team). The PlanMapper module generated the visual trace of the trainee (shown directly on the trainers display) and the log of the trace (placed in the trainee assessment log) as required. The TeamMetrics module generated the count of the individual communications (a simple message count placed in the After Action Review Log) and recorded information exchanges (through IARG) of the individual team members for the post-session review. As can be seen in Figure 57 the total message count for the team was 63. All of these messages were generated by the IARG algorithm for the Virtual Agents as part of their attack assignments. The important point to be made is that the message count and information exchanges are generated through generic mechanisms but are based on domain specific MALLET plans.

For validation of the coaching support in the Framework we looked at both the Coaching Agent's functionality and the generic module, BasicCoach. The Coaching Agent loaded and executed the assessment and coaching modules as expected. The

Coaching Agent also generated the visual displays to watch the training session execute in real time. The BasicCoach module was used to start the post-session phase. As part of the post-session phase, the generic Coaching Agent saved the required logs and the AfterActionReview generated the post-session results and review display.

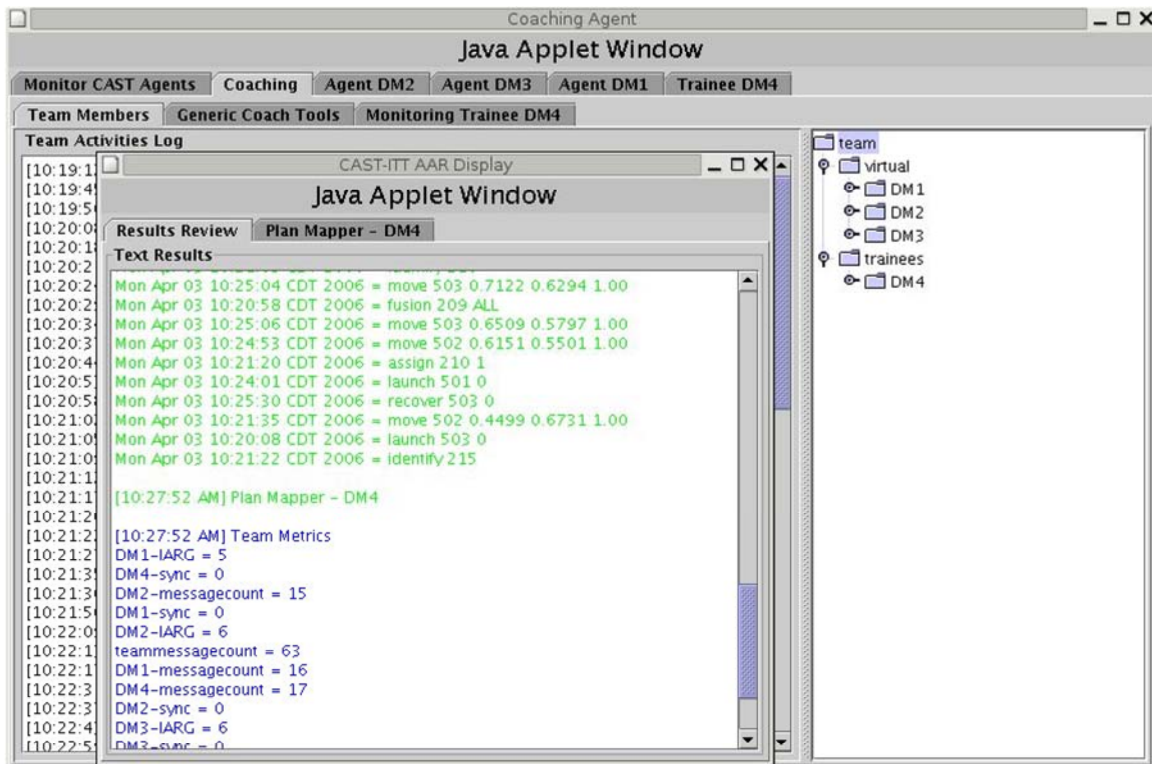

Figure 57: Generic coaching agent

Figure 57 shows an example of the execution of the Coaching Agent after a completed session with the AfterActionReview window also open. As demonstrated in

these validation trials the Framework provides useful capabilities even without domain specific assessment and coaching.

7.3 Domain Specific Assessment and Coaching for DDD

In addition to the generic coaching support described in the previous section, the Framework provides mechanisms, as described in Section 5, by which domain specific assessment and coaching can be included in a training system. We have not tested those capabilities here, but work being conducted at Penn State University has been using these features for over a year to develop a training system for further investigations of training helping behaviors.

Chen Cong is a PhD student at Pennsylvania State University involved in this research. Her research is focused on developing a model of event-based coaching to identify and evaluate supporting helping behavior among team members. Already for DDD as part of the MURI effort, Cong has constructed a number of coaching aides as part of her research (Xu et al., 2003). In her work, she has built upon the implementation of the Framework to provide an event-based assessment of each individual trainee in the DDD domain in regards to their domain performance. The basic idea for event-based assessment is to use process-oriented performance metrics linked to the training objectives through scenario events. This breaks down into three parts:

1. Identify and model the performance metrics
2. Monitor and identify the key events associated to those metrics
3. Provide feedback as appropriate based on the metrics

For the Framework there is no additional work to be done. Instead, the training system developer (in this case Cong) extends the Coaching Agent through the interfaces provided to enable her model of coaching to function. Cong is using all of the key components of the Framework. The only component not being used is the real time feedback support in the Framework. This is dictated by the training objectives in her experiments. The completion of her use of the Framework is planned for the summer of 2006.

8. FUTURE WORK AND CONCLUSIONS

A team training framework for constructing training systems both speeds the development of team training systems and allows the training system developer to focus on solving their domain specific problems by providing a basic set of capabilities. The additional complexities of building team training systems can be reduced by using an agent-based approach that provides a natural view of the team being trained to a training system developer.

The Framework produced for this dissertation allows a training system developer to build team training systems with the following capabilities: 1) support for flexible training strategy selection, 2) support for easily incorporating a variety of simulation training domains, 3) an agent-based understructure that speeds the development of such systems, 4) mechanisms for building and utilizing joint human/agent teams, 5) support for the building of coaches implementing a variety of approaches, 6) automation of the support for providing monitoring capabilities, 7) monitored data access for, and interoperation of, individual and team assessments, 8) provision of interfaces for both team evaluations by coaches (human or automated) and specific individual feedback, and 9) displays for human coaches (which can be tailored to specific domains).

Future work discussed in this section focuses on both what can be done in the short term with the Framework to improve the ease of development and provide more sophisticated support for other aspects of teamwork, and, in the long term, adding more intelligence to agents and better mechanisms for achieving human-like behavior.

8.1 Significance of Research

This research demonstrates an approach towards constructing agent-based team training systems through the use of a team-oriented training framework. Such a team training framework provides a basis for building training systems in which virtual team members can inter-operate with human trainees, team-based training scenarios can be executed, and either human or automated coaches can provide feedback to the trainees. In this research, we have made several contributions towards outlining the issues involved and a framework that support these capabilities.

A primary contribution is a framework that supports construction of training systems. The Framework does this by providing well defined interfaces that facilitate the development of specialized team training systems. The Framework also incorporates significant parts of a very general training system to reduce the work required of a training system developer. Furthermore, the Framework supports the easy incorporation of different simulation domains representing C2 environments by making a distinction between a training environment and the team that trains in that environment.

The distinction between the training environment and the team allows the Framework to address connecting to a training simulation as a separate issue from providing the virtual team members as part of a team for purposes of training. This distinction allows the training system developer to manage separately the development of the physical training system and the development of the protocols of the team training in regards to the training objectives.

Communications for teamwork is handled by the Framework through the use of intelligent agents. One class of agents acts as virtual team members for interoperating with human trainees. Another class of agents acts as monitors for individual trainees. Furthermore, the monitor agents play a supporting role in enabling agent/human and human/agent communications. Underneath both classes of agents exists an architecture that supports teamwork and team oriented communications, CAST. This combination of agents allows for agent/human and human/agent interactions that can be extended to almost any level of communications that the training systems developer can build.

The most important use of intelligent agents by the Framework is in regards to virtual team members. The agents of the Framework share the commonality of the underlying model of teamwork expressed in MALLET to support interactions between human trainees and virtual team members. MALLET does this by supporting a notion of explicit plan execution based on implicit information exchanges and role assignments at the team member level. The use of MALLET provides a team-oriented view for modeling the team and such a view is used by all of the agents incorporated within the Framework.

Monitoring of individual trainees in the context of a team model gives us a powerful way to look at both the individual team members and the overall team. The Framework supports both the individual level and the team level by monitoring individual trainees and collating such monitored data to a team level, allowing both individual and team assessment and coaching. Monitoring and assessment are separated in order to make the development and incorporation of domain specific capabilities

simpler. Moreover, certain generic monitoring tools are provided that may be used with any domain. To enable the use of domain specific monitoring tools, we provide interfaces that can be extended and implemented to include whatever tools the training systems developer can create.

The Framework also supports incorporation of both generic and domain specific assessment and coaching. In particular, the Framework provides certain basic generic assessment and coaching capabilities and interfaces that can be extended and implemented to provide either additional generic assessment and coaching or domain specific assessment and coaching. These capabilities and interfaces allow the assessment of both taskwork and teamwork to be included. For coaching, the Framework additionally has support for adding coaching via either humans or intelligent agents for both on-line and during the post training session.

In demonstrating the Framework though the TWP-DDD training system we have shown that a training system can be built that has good performance and can be used for achieving real world training objectives. In addition, very significant extensions and improvements have been made in the TWP-DDD over the capabilities of the initial version of CAST.

8.2 Future Work

There are both short term and long term directions that have been identified in the course of this research for future work. In the short term are the goals of working towards making the interfaces of the Framework even easier to use by the training system developer, improving the system performance of the agents within the Framework, and

using the Framework for other training systems. In the long term there are three directions for future research that we have identified. These three directions are improving the model of teamwork used in the Framework; adding advanced capabilities such as more inherent intelligence in the agents of the Framework, and better support in the Virtual Agents for more human-like behavior as has been identified in the psychological research.

MALLET/CAST is a model that is undergoing constant improvement. Improving the model of teamwork used or adding support for alternate models of teamwork improves the capability of the Framework. Expansions on the model of teamwork such as observability of other team members, and adding stronger intentions support for analyzing other team members' behaviors have been identified during this research as subjects for future work. Observability can be used to reduce required team communications (Ioerger, 2004). This is done by adding beliefs on observations of other team members to a variant of CAST/MALLET (Zhang et al., 2002). From the perspective of the Framework, incorporating observability into the model would involve changes to CAST such as using observability in the IARG analysis or extending the capabilities of the monitor agent and display components. From the training systems developers' perspective, the expert model created might well become more sophisticated and the coaches could have additional information on which to base evaluations and feedback. Better observability leads to better support for analyzing intentions by both virtual team members and Monitor Agents. Another approach to adding stronger intentions support is through smarter agents.

The intelligence built into the agents is crucial to the Framework. The more capable the agents are, the more that can be done with them by the training system developer. As a start, by improving awareness of others' behaviors by the Virtual Agents, we can improve understanding of their decision making. By adding extra consideration for choosing when and how to act, e.g. inclusion of other models of decision making such as Klein's Recognition-Primed Decision framework (RPD), an agent can better support team collaborations (Fan et al., 2005). We can therefore improve the Virtual and Monitor Agents' ability to react to changes in the behaviors of human trainees who are following a naturalistic decision making process, a consideration in training humans in C2 teams. This leverages a training system built with the Framework since other components such as the Coaching Agent use the Virtual and Monitor Agents.

As discussed previously in regards to appropriate behavior for training purposes, it has been found that human-like behavior in agents can be beneficial (Ioerger et al., 2003). Therefore providing better methods and/or tools for facilitating the development of such behaviors in agents is beneficial toward reducing the development time required by the training system developer. For example, adding a sense of time and operator timing requirements to MALLET/CAST could enable better emulation of human behavior. In addition, it would be useful to create a few high level templates for team activity, e.g., such as the basic parallel structure used in the DDD training scenarios. These kinds of additions could help the training systems developer more easily create the desired human-like behaviors in agents.

8.3 Conclusions

In this dissertation we have defined a team training framework for constructing team training systems for team training domains. The validation and real world use of the prototype developed for the Framework demonstrates the leveraging provided to the training system developer in reduced development time and enhanced training support given by the Framework. We thus believe that it is feasible to use a general framework for building training systems and that by doing so, the effort required to incorporate different simulation training domains can be reduced and the developers allowed to focus more on the creation of new training protocols, new uses for intelligent agents (such as for coaches) and more advanced team behaviors for which training is desired.

While the integration of a simulation domain and construction of a training system are facilitated by use of the Framework, the key issues of determining desired expert behavior remain. From our experience, it will require significant cognitive task analysis before much can be done with building the training system, and this is likely to take much longer than the actual construction of the training system. We have also learned that the use of intelligent agent-based training systems requires a bit of extended work on the cognitive task analysis. There is a tendency for the agent builders to try to optimize the agent behavior, given a high level task analysis. The result can be that the agent team members perform the tasks ascribed to them by the cognitive task analysis, but do so in a manner that does not seem “natural” to human trainees. In a sense, this adds a more detailed aspect to the needed cognitive task analysis, that of capturing what feels natural to a team member, not just what tasks the team member does.

Another potentially limiting consideration is the nature of agent/human and human/human communication. As natural language understanding is not yet available for computer systems, training situations that depend upon that will be difficult to handle. Fortunately, for many important domains, reasonably narrow prescribed verbal communication protocols are used and may often be moderately easily incorporated into computer systems. However, rather than try to incorporate a specific general communication mechanism, the Framework provides a very general mechanism for incorporating whatever communication mechanisms a developer can create for a given application.

In summary, we believe that the use of a team training framework, in general, and the one developed here, in specific, is a useful way in which to experiment with agent-based training systems. Nevertheless, extensions to include the kinds of enhancements identified above would make such a framework even more useful.

REFERENCES

Allen, J., Hendler, J., & Tate, A. (1990). *Readings in Planning*. Morgan Kaufmann, San Mateo, CA.

Ashcroft, M. H. (1994). *Human Memory and Cognition*. HarperCollins College Publishers, New York, NY.

Barbuceanu, M., & Fox, M. S. (1995). Cool: A language for describing coordination in multi-agent systems. In *Proceedings of the First International Conference on Multi-Agent Systems*, pp. 17-24. San Francisco, CA. MIT Press.

Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, *13*, 4-16.

Bratman, M. E. (1987). *Intention, Plans, and Practical Reasons*. Harvard University Press, Cambridge, MA.

Builder, C. H., Bankes, S. C., & Nordin, R. (1999). *Command Concepts: A Theory Derived from the Practice of Command and Control*. RAND, Santa Monica, CA.

Cannon-Bowers, J. A., & Salas, E. (1998). *Making Decisions under Stress: Implications for Individual and Team Training*. American Psychological Association, Washington, DC.

Cannon-Bowers, J. A., Tannenbaum, S. I., Salas, E., & Volpe, C. E. (1995). Defining competencies and establishing team training requirements. In Guzzo, R. A., & Salas, E. (Eds.), *Team Effectiveness and Decision Making in Organizations*, pp. 333-380. San Francisco, CA. Jossey-Bass Publishers.

Cao, S. (2005). *Role-based and Agent-oriented Teamwork Modeling*, Ph.D. dissertation, Texas A&M University, College Station, TX.

Cao, S., Volz, R. A., Johnson, J., Whetzel, J., Xu, D., et al. (2004). Development of a distributed multi-player computer game for scientific experimentation of team training protocols. *The Electronic Library*, *22*, 43-54.

Carbonell, J. R. (1970). AI in CAI: An artificial intelligence approach to computer-assisted instruction. *IEEE Transactions on Man-Machine Systems*, *11*, 190-202.

Carr, B. P., & Goldstein, I. P. (1977). Overlays: A Theory of Modeling for Computer-aided Instruction. Tech. rep. AI Lab Memo 406, MIT, Cambridge, MA.

Cohen, P. R., & Levesque, H. J. (1991). Teamwork. *Nous*, *25*, 487-512.

Coovert, M. D., & McNelis, K. (1992). Team decision making and performance: A review and proposed modeling approach employing petri nets. In Swezey, R. W., & Salas, E. (Eds.), *Teams: Their Training and Performance*, pp. 247-280. Ablex Pub Corp. Norwood, NJ.

Corbett, A. T., Koedinger, K. R., & Anderson, J. R. (1997). Intelligent tutoring systems. In Helander, M. G., Landauer, T. K., & Prabhu, P. V. (Eds.), *The Handbook of Human-Computer Interaction*, pp. 849-874. Elsevier Science. New York, NY.

Donchin, E., Fabiani, M., & Saunders, A. (1989). The learning strategies program: An examination of the strategies in skill acquisition. *Acta Psychologica*, *71*, 16-35.

Fan, X., Sun, S., McNeese, M., & Yen, J. (2005). Extending the recognition primed decision model to support human agent collaboration. In *Proceedings of The Fourth International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS 2005)*, pp. 945-952. Utrecht, The Netherlands. Springer-Verlag.

Fan, X., Yen, J., Miller, M. S., Ioerger, T. R., & Volz, R. A. (2006). MALLET – A multi-agent logic language for encoding teamwork. *IEEE Transactions on Knowledge and Data Engineering*, *18*, 123-138.

Fan, X., Yen, J., Wang, R., Sun, S., & Volz, R. A. (2004). Context-centric proactive information delivery. In *International Conference on Intelligent Agent Technology*, pp. 31-37. Beijing, China. IEEE Computer Society Press.

Finin, T., Labrou, Y., & Mayfield, J. (1997). KQML as an communication language. In Bradshaw, J. M. (Ed.), *Software Agents*, pp. 291-316. AAAI Press. Menlo Park, CA.

Genesereth, M. R., & Fikes, R. E. (1992). Knowledge Interchange Format, Version 3.0, Reference Manual. Tech. rep. Logic-92-1, Stanford University, Palo Alto, CA.

Georgeff, M. P., & Lansky, A. L. (1987). Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pp. 677-682. Seattle, WA. AAAI Press.

Goettl, B. P., Halff, H. M., Redfield, C. L., & Shute, V. J. (1998). Intelligent tutoring systems. In *Intelligent Tutoring Systems '98*, pp. 1-30. San Antonio, TX. Springer-Verlag.

Grosz, B., & Kraus, S. (1996). Collaborative plans for complex group action. *Artificial Intelligence*, *86*, 269-357.

Hollingshead, A. B., & McGrath, J. E. (1995). Computer-assisted groups: A critical review of the empirical research. In Guzzo, R. A., & Salas, E. (Eds.), *Team Effectiveness and Decision Making in Organizations*, pp. 46-78. Jossey-Bass Publishers. San Francisco, CA.

Hutchins, E. (1995). *Cognition in the Wild*. MIT Press, Cambridge, MA.

IEEE. (1997). IEEE 1278, IEEE Recommended Practice for Distributed Interactive Simulation. 2005, from http://standards.ieee.org/catalog/olis/.

IEEE. (2000). IEEE 1516-2000, IEEE Standard for Modeling and Simulation (M&S) High Level Architecture. 2005, from http://standards.ieee.org/catalog/olis/.

Ioerger, T. R. (2003). Java Automated Reasoning Engine. 2005, from http://jare.sourceforge.net/.

Ioerger, T. R. (2004). Reasoning about beliefs, observability, and information exchange in teamwork. In *7th International Conference of the Florida Artificial Intelligence Research Society (FLAIRS'04)*, pp. 23-31. Miami Beach, FL. AAAI Press.

Ioerger, T. R., Sims, J., Volz, R. A., Workman, J., & Shebilske, W. L. (2003). On the use of intelligent agents as partners in training systems for complex tasks. In *Twenty-Fifth Annual Meeting of the Cognitive Science Society, CogSci 2003*, pp. 51-56. Boston, MA. Lawrence Erlbaum Associates.

Jennings, N. R. (1993). Commitments and conventions: The foundation of coordination in multi-agent systems. *The Knowledge Engineering Review*, *8*, 223-250.

Jennings, N. R., Sycara, K., & Wooldridge, M. (1998). A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, *1*, 275-306.

Jensen, K., & Rozenberg, G. (1991). *High-level Petri Nets*. Springer-Verlag, New York, NY.

Johnson-Laird, P. N. (1993). *Human and Machine Thinking*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Johnston, J. H., Smith-Jentsch, K. A., & Cannon-Bowers, J. A. (1997). Performance measurement tools for enhancing team decision making. In Brannick, M. T., Salas, E., & Prince, C. (Eds.), *Team Performance Assessment and Measurement:*

Theory, Research, and Applications, pp. 311-330. Lawrence Erlbaum Associates. Hillsdale, NJ.

Kass, R. (1989). Student modeling in intelligent tutoring systems - Implications for user modeling. In Wahlster, A. K. a. W. (Ed.), *User Models in Dialog Systems*, pp. 386-410. Springer-Verlag. New York, NY.

Katz, S., Lesgold, A., Eggan, G., & Gordin, M. (1994). Modeling the student in sherlock II. In Greer, J. E., & McCalla, G. I. (Eds.), *Student Modeling: The Key to Individualized Knowledge-Based Instruction*, pp. 99-126. Springer-Verlag. Berlin, Germany.

Klein, G. A., Calderwood, R., & Clinton-Cirocco, A. (1986). Rapid decision making on the fireground. In *30th Annual Meeting of the Human Factors and Ergonomics Society*, pp. 576-80. Dayton, OH. Human Factors Society.

Kleinman, D. L., Young, P. W., & Higgins, G. (1996). The DDD-III: A tool for empirical research in adaptive organizations. In *Proceedings of the 1996 Command and Control Research and Technology Symposium*, pp. 197-209. Monterey, CA. CCRP Publication Series.

Kupper, D., & Kobsa, A. (1999). User-tailored plan generation. In *Seventh International Conference on User Modeling*, pp. 45-54. Banff, Canada. SpringerWienNewYork.

Laird, J. E., Pearson, D. J., Jones, R. M., & Wray III, R. E. (1996). Dynamic knowledge integration during plan execution. In *AAAI-96 Fall Symposium on Plan Execution*, pp. 92-98. Cambridge, MA. AAAI Press.

Lajoie, S. P., & Lesgold, A. (1992). Apprenticeship training in the workplace: Computer-coached practice environment as a new form of apprenticeship. In Farr, M. J., & Psotka, J. (Eds.), *Intelligent Instruction By Computer: Theory and Practice*, pp. 15-36. Taylor & Francis New York, Inc. Washington, DC.

Lawson, J. S. (1981). Command and control as a process. *IEEE Control Systems Magazine*, *1*, 5-11.

Lee, P., Phillips, C., Otani, E., & Badler, N. I. (1989). The JACK interactive human model. In *First Annual Symposium on Mechanical Design in a Concurrent Engineering Environment*, pp. 179-198. Iowa City, IA. ASME Books.

Lesh, N., Rich, C., & Sidner, C. L. (1999). Using plan recognition in human-computer collaboration. In *Seventh International Conference on User Modeling*, pp. 23-32. Banff, Canada. SpringerWienNewYork.

McIntyre, R. M., & Salas, E. (1995). Measuring and managing for team performance: Lessons from complex environments. In Guzzo, R. A., & Salas, E. (Eds.), *Team Effectiveness and Decision Making in Organizations*, pp. 9-45. Jossey-Bass Publishers. San Francisco, CA.

Miller, M. S., Yin, J., Ioerger, T. R., Yen, J., & Volz, R. A. (2000). Training teams with collaborative agents. In *Proceedings of the Fifth International Conference on Intelligent Tutoring Systems*, pp. 63-72. Montreal, Canada. Springer.

Moldt, D., & Wienberg, F. (1997). Multi agent systems based on colored petri nets. In *18th International Conference ICATPN'97*, pp. 82-101. Toulouse, France. Springer-Verlag.

Morgan Jr., B. B., & Bowers, C. A. (1995). Teamwork stress: Implications for team decision making. In Guzzo, R. A., & Salas, E. (Eds.), *Team Effectiveness and Decision Making in Organizations*, pp. 262-290. Jossey-Bass Publishers. San Francisco, CA.

Morgan Jr., B. B., Glickman, A. S., Woodard, A. E., & Blaiwes, A. (1986). Measurement of Team Behaviors in a Navy Environment. Tech. rep. 86-014, Naval Training Systems Center, Orlando, FL.

NASA. (1998). Space Shuttle News Reference Manual. 2005, from http://www.globalsecurity.org/space/library/report/1988/stsref-toc.html.

Nolan, M. S. (2003). *Fundamentals of Air Traffic Control*. Thomson Brooks/Cole, Belmont, CA.

Orasanu, J., & Connolly, T. (1993). The reinvention of decision making. In Klein, G. A., Orasanu, J., Calderwood, R., & Zsambok, C. E. (Eds.), *Decision Making in Action: Model and Methods*, pp. 3-20. Ablex Pub Corp. Norwood, NJ.

Pew, R. W., & Mavor, A. S., Eds. (1998). *Modeling Human and Organizational Behavior: Application to Military Simulation*. National Academy Press. Washington, DC.

Plymale, J. (2004). Extensions to the Jare Inference Engine. Tech. rep. 2004-12-X, Texas A&M University, College Station, TX.

Rao, A. S., & Georgeff, M. P. (1991). Modeling rational agents within a BDI architecture. In *Second International Conference on Principles of Knowledge Representation and Reasoning*, pp. 473-484. Cambridge, MA. Morgan Kaufmann.

Rao, A. S., & Georgeff, M. P. (1995). BDI agents: From theory to practice. In *First International Conference on Multi-agent Systems*, pp. 312-319. San Francisco, CA. AAAI Press.

Rickel, J., Lesh, N., Rich, C., Sidner, C., & Gertner, A. (2001). Building a bridge between intelligent tutoring and collaborative dialogue systems. In *Proceedings of Tenth International Conference on AI in Education*, pp. 592-594. San Antonio, TX. IOS Press.

Russell, S., & Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ.

Self, J. (1990). Bypassing the intractable problem of student modelling. In Frasson, C., & Gauthier, G. (Eds.), *Intelligent Tutoring Systems: At the Crossroads of Artificial Intelligence and Education*, pp. 107-123. Ablex Pub Corp. Norwood, NJ.

Shebilske, W., Goettl, B., & Regian, J. W. (1999). Executive control and automatic processes as complex skills develop in laboratory and applied settings. In Gopher, D., & Koriat, A. (Eds.), *Attention and Performance XVII*, pp. 401-432. MIT Press. Cambridge, MA.

Shebilske, W. L., Jordan, J. A., & Arthur Jr., W. (1993). Combining a multiple emphasis on components protocol with small group protocols for training complex skills. In *37th Annual Meeting of the Human Factors and Ergonomics Society*, pp. 1216-1220. Seattle, WA. Lawrence Erlbaum Associates.

Shebilske, W. L., Regian, J. W., Arthur Jr., W., & Jordan, J. A. (1992). A dyadic protocol for training complex skills. *Human Factors*, *43*, 369-374.

Singhal, S., & Zyda, M. (1999). *Networked Virtual Environments*. ACM Press, New York, NY.

Sowa, J. F. (2000). *Knowledge Representation: Logical, Philosophical, and Computational Foundation*. Brooks/Cole, Pacific Grove, CA.

Srivathsan, B. (2005). Impact of Agent Design on Training. Tech. rep. 2005-5-X, Texas A&M University, College Station, TX.

Tambe, M. (1997). Towards flexible teamwork. *Journal of Artificial Intelligence Research*, *7*, 83-124.

Tambe, M., Adibi, J., Al-Onaizon, Y., Erdem, A., Kaminka, G. A., et al. (1999). Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence*, *110*, 215-240.

TRADOC (1997). *FM 101-5 Staff Organization and Operations*. Department of the Army, Fort Leavenworth, KS.

TRADOC (2001). *FM 100-5 Operations*. Department of the Army, Fort Leavenworth, KS.

TRADOC (2003). *FM 7-1 Battle Focused Training*. Department of the Army, Fort Leavenworth, KS.

Volz, R. A., Ioerger, T. R., Shebilske, W. L., & Yen, J. (2005). MURI Annual and Final Report: Intelligent Distributed Group and Team Training Systems. Tech. rep. F49620-00-1-0326, Texas A&M University, College Station, TX.

Wahlster, W., & Kobsa, A. (1989). User models in dialog systems. In Wahlster, W., & Kobsa, A. (Eds.), *User Models in Dialog Systems*, pp. 4-34. Springer-Verlag. New York, NY.

Wenger, E. (1987). *Artificial Intelligence and Tutoring Systems*. Morgan Kaufmann, Los Altos, CA.

Xu, D., Miller, M. S., Volz, R. A., & Ioerger, T. R. (2003). Collaborative agents for C2 teamwork simulation. In *International Conference on Artificial Intelligence (IC-AI'2003)*, pp. 723-729. Las Vegas, NV. CSREA Press.

Yen, J., Yin, J., Ioerger, T. R., Miller, M. S., Xu, D., et al. (2001). CAST: Collaborative agents for simulating teamwork. In *Seventeenth International Joint Conference on Artificial Intelligence*, pp. 1135-1144. Seattle, WA. Morgan Kaufmann.

Yin, J. (2001). *A Multi-Agent Framework for Simulating Proactive Teamwork*, Ph.D. dissertation, Texas A&M University, College Station, TX.

Yin, J., Miller, M. S., Ioerger, T. R., Yen, J., & Volz, R. A. (2000). A knowledge-based approach for designing intelligent team training systems. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pp. 427-434. Barcelona, Spain. ACM Press.

Zhang, Y., Volz, R. A., Ioerger, T. R., Cao, S., & Yen, J. (2002). Proactive information exchange during team cooperation. In *International Conference on Artificial Intelligence (IC-AI'2002)*, pp. 341-346. Las Vegas, NV. CSREA Press.

Zsambok, C. E. (1997). Naturalistic decision making: Where are we now. In Zsambok, C. E., & Klein, G. (Eds.), *Naturalistic Decision Making*, pp. 3-16. Lawrence Erlbaum Associates. Mahwah, NJ.

APPENDIX A

A BNF representation compiled from a parser built to test the elements of a MALLET file. A MALLET file may also load other MALLET files using the Load command.

NumberLiteral ::= <INTEGER> | <FLOAT>
Identifier ::= NumberLiteral | <NAME> | <PATH>
Variable ::= <VARIABLEPREFIX> Identifier
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 | <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)+ ")" ")"
RoleDef ::= "(" <ROLE> RoleName (Invocation | "(" (Invocation)+ ")") ")"
PlaysRole ::= "(" <PLAYSROLE> AgentName (RoleName | "(" (RoleName)+ ")") ")"
FulfilledBy ::= "(" <FULFILLEDBY> RoleName (AgentName | "(" (AgentName)+ ")") ")"

PreConditionList ::= "(" <PRECOND> (Cond)+ (<IFFALSE> (<FAIL> | <WAIT> (<INTEGER> | Variable)? | <ACHIEVE>))? ")"
EffectsList ::= "(" <EFFECTS> (Cond)+ ")"
TermConditionsList ::= "(" <TERMCOND> (<SUCCESS> | <FAILURE>)? (Cond)+ ")"

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

IOperDef ::= "(" <IOPER> OperName VariableListOpt (PreConditionList)* (EffectsList)? ")"
TOperDef ::= "(" <TOPER> OperName VariableListOpt (PreConditionList)* (EffectsList)? (NumSpec)? ")"
PlanDef ::= "(" <PLAN> PlanName VariableListOpt (PlanOption)* "(" <PROCESS> MalletProcess ")" ")"
PlanOption ::= PreConditionList | EffectsList | TermConditionsList
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 ")")+ | <DO> ByWhomSpec MalletProcess | <AGENTBIND> VariableListReq "(" <CONSTRAINTS> (Cond)+ ")" | PlanOrOperName (Identifier | Variable)*) ")" | AssertDef | RetractDef)

APPENDIX B

The CAST configuration files use XML to describe the format of the various configuration parameters for starting up agents in CAST. Each CAST agent will be listed by name, domain, the top level MALLET file, and the location of the CAST Monitor (a monitoring tool provided as part of CAST). Optional arguments are to set the agent as a monitor of a trainee (TRAINEE="true"), and a number of configuration arguments for displaying of the agent status during execution. The entire CAST system can be started in a paused state and the time step for all agents to follow is specified using TIMESTEP="time step in ms". The training domain developer may also specify unique arguments for their domain using the NAME VALUE pair data element. The CAST Document Type Definition details the CAST configuration file format.

CAST.DTD

```
<!ELEMENT CAST ((AGENT)+,CASTMONITOR?)>

<!ELEMENT AGENT EMPTY>
<!ATTLIST AGENT NAME CDATA #REQUIRED>
<!ATTLIST AGENT DOMAIN CDATA #REQUIRED>
<!ATTLIST AGENT MALLET CDATA #REQUIRED>
<!ATTLIST AGENT MONITOR CDATA #REQUIRED>
<!ATTLIST AGENT TRAINEE CDATA #IMPLIED>
<!ATTLIST AGENT DISPLAY CDATA #IMPLIED>
<!ATTLIST AGENT FOLLOW CDATA #IMPLIED>
<!ATTLIST AGENT LOGGER CDATA #IMPLIED>
<!ATTLIST AGENT IARG CDATA #IMPLIED>

<!ELEMENT CASTMONITOR ((DATA)*)>
<!ATTLIST CASTMONITOR PAUSED CDATA #IMPLIED>
<!ATTLIST CASTMONITOR TIMESTEP CDATA #IMPLIED>

<!ELEMENT DATA EMPTY>
<!ATTLIST DATA NAME CDATA #REQUIRED>
```

```
<!ATTLIST DATA VALUE CDATA #REQUIRED>
```

Example.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CAST SYSTEM "cast.dtd">
<CAST>
        <AGENT NAME="DM0" DOMAIN="cast3.dynamic.demo.Default"
                MALLET="teamplans/demo/example.mlt" FOLLOW="true"
                MONITOR="localhost"/>
        <AGENT NAME="DM1" DOMAIN="cast3.dynamic.demo.Default"
                MALLET="teamplans/demo/example.mlt" FOLLOW="true"
                MONITOR="localhost"/>
        <CASTMONITOR PAUSED="true" TIMESTEP="500"/>
</CAST>
```

The file name of the CAST configuration file is passed to the CAST-ITT system at startup and can therefore be named to match the configuration as appropriate.

Additionally there are four more configuration files that are used by the CAST-ITT system. These configuration files are KERNEL.XML, TRAINEEMONITOR.XML, TRAINEASSESS.XML, and COACH.XML. All four of these configuration files use the Module Document Type Definition to detail their configuration file format.

MODULE.DTD

```
<!ELEMENT MODULES ((MODULE)+)>
<!ELEMENT MODULE EMPTY>
<!ATTLIST MODULE NAME CDATA #REQUIRED>
<!ATTLIST MODULE ACTIVE CDATA #REQUIRED>
```

CAST-ITT uses these four configuration files to load modules for the appropriate associated agent at start up.

- Virtual Agent KERNEL.XML
- Monitor Agent TRAINEEMONITOR.XML
- Trainee Assessment Agent TRAINEASSESS.XML
- Coaching Agent COACH.XML

APPENDIX C

CAST logging services depend on two types of logs. These logs are the Java logging API and a RMI logging service. The Java logging service has been extended with the ExcelFormatter for storing the log records in a flat tab delimited file. This file format is also used for all logs that are created by the Framework. The tab delimited file has the following columns:

- Sequence number
- Milliseconds since start
- Level of log entry
- Class name of calling method
- Calling method
- Log text

VITA

Name: Michael Scott Miller

Address: Department of Electrical Engineering and Computer Science

U.S. Military Academy

West Point, N.Y. 10996

Email Address: msmiller@acm.org

Education: B.S., Computer Science, Texas A&M University, 1990

M.S., Computer Science, University of Houston-Clear Lake, 1997

Ph.D., Computer Science, Texas A&M University, 2006