Chapter 1: Introduction

1.1 Research Objective

The Department of Defense (DoD) uses simulation models to enhance training and support decision-making. These models help test war plans against adversaries, influence force structure decisions, determine what equipment to acquire, decide the best combination and use of weapons, and explore potential changes in doctrine or tactics [1]. Since there are many factors that can potentially affect military conflicts, most of the traditional community simulations are extremely complex and resource intensive. The scenario generation process for these high-resolution simulations is man-hour intensive and requires detailed knowledge of the simulation models' underlying data and operating assumptions. The time-intensive data collection/scenario generation process, coupled with long run times, often limits analysts to a small set of simulation runs.

Combat simulation systems are used by Armed forces around the world as an important tool to train its personnel and to devise new doctrines and strategies. In a realistic training exercise the cost of involving human players in the battle space is very expensive. Computer Generated Forces (CGF) comes to the rescue here. CGF [2,8] have been used in training as well as tactics development. CGF can potentially replace humans in Combat simulation systems to reduce cost of training exercises.

Over the previous decade, there has been a significant amount of work done on the development of intelligent, Computer Generated Forces (CGF) capable of combat behavior within synthetic battlefields. Irreducible Semi-Autonomous Adaptive Combat (ISAAC) was an attempt to model land combat using agent-based simulation techniques. . The central thesis of ISAAC is that land combat can be thought of as a complex adaptive system - combat forces are composed of large numbers of nonlinearly interacting parts and are organized in a command and control hierarchy .Completed in 1997, the

central thesis of Ilachinski's model [2] is that land combat can be thought of as a complex adaptive system - combat forces are composed of large numbers of nonlinearly interacting parts and are organized in a command and control hierarchy.

Historically computer simulation has been used for the evaluation of acquisitions and of force development options. But modeling and simulation for this purpose is becoming increasingly complex as multi-role, multi-platform and multi-system aspects are taken into consideration. The complexity of this task is further increased by the difficulty in modeling human decision-making with sufficient fidelity using conventional software approaches. Current implementations of Computer Generated Forces, such as ModSAF, have proven to be very useful, but do not model human reasoning and cannot easily model team behavior.

War games are commonly used as a means for exploring the effects of improved equipment or revised operational approaches on force capability. One commonly used war game is CAEN (Close Action Environment), developed by the Defence Evaluation Research Agency in the UK. CAEN allows analysists to model engagements and operations from the level of the individual soldier to the company level, and is used for both rural and urban environments. However, war games, such as CAEN, are currently limited by the need to provide detailed pre-prepared scripts describing the actions to be followed by the simulation entities.Consequently these entities are neither autonomous, nor do they provide the ability to model team behavior.

In conventional simulation modeling environments (such as CAEN [9] or ModSAF), a war game is a tightly scripted scenario in which the activity of each entity is pre-programmed in isolation with respect to the simulation clock. An entity 'moves' and 'acts' in the simulation according to the script, which tells exactly when and how the entity should act throughout the scenario, almost independently from other entities. The conventional simulation

environment offers only a very minimal level of situational awareness, for example, such that an entity may decide to fire or not fire its weapon depending on whether or not another entity is sighted. However, more complex behavioral variations, such as choosing where or whether to cross a road, cannot be expressed within the scripted scenario.

Early applications of intelligent agents in simulations to represent operational military reasoning have proved highly effective. This success comes from the capability of agents to represent individual reasoning and from the architectural advantages of that representation to the user due to the ease of setting up and modifying operational reasoning or tactics for various studies. In addition, the BDI class of agents extends the modeling of reasoning to explicitly model the communications and coordination of joint activities required for team behavior.

Intelligent agents has the potential of representing complex behavior & team modeling capability [3] using agent based simulation [ABS]. ABS[4] represents a shift from the traditional force-on-force attrition calculations (typically containing scripted entities or utilizing humans for decision-making) to considering how high-level properties and behaviors of a system emerge out of low-level rules applied to individual agents. The conceptual focus shifts from finding a mathematical description of an entire system to a low-level rule based specification of the behavior of individual agents making up that system [2].

Agent-based modeling and simulation [5] is a maturing approach to modeling combat systems comprising of autonomous, interacting battlefield entities which have individual goals as well as overall group goals, that must be balanced to achieve the global objective. These entities, represented as agents, interact with some degree of autonomy and continually make decisions to satisfy a variety of sometimes conflicting objectives. This technique can be used to model battlefield scenarios where multiple entities

sense and stochastically respond to conditions in their local environments, mimicking large-scale combat system behavior which is essentially a non-linear complex system. In such systems, normal linear modeling and simulation techniques do not satisfactorily model or explain the behavior that the system exhibits because processes and actions are not directly proportional to, or related to input. Further, since these complex systems have many components that interact, cause and effect cannot be separated.

ABSs are based on the idea that is possible to represent in computerized form the behavior of entities which are active in the world, and that it is thus possible to represent an emergent collective behavior [4] that results from the interactions of an assembly of autonomous agents [6].

Military combat has many of the key features of complex adaptive systems [2]. Combat forces are composed of large numbers of nonlinearly interacting parts that are organized in a command and control hierarchy

Command & control functions in combat simulations require human decision makers in loop. The need to automate process of C&C arises [7] when either simulation is carried in close loop or human decision makers are not available or cost effective. Further, command decisions are based on static knowledge base of strategic doctrines, tactical situation and experience of the commander. A command decision model based on such static knowledge acts as decision support system for the commanders.

Each battlefield entity in C&C hierarchy has some local objective which it continuously tries to fulfill, in order to satisfy the overall group objective. Each of the individual agents may have partial information (decentralized data) and capability (simple rules) for problem solving and thus a limited view. Each soldier on the battlefield has some degree of autonomy and is continually making decisions to satisfy a variety of sometimes conflicting objectives. For example, a soldier may simultaneously desire to move towards an objective,

remain unobserved by the enemy, obey his commander's orders, stay close to his friends, etc. In addition, each of the soldiers in a unit may value the various objectives differently. Consequently, there may often appear to be disorder at the local level, but long-range order at the global level.

Indeed, using very simple models, Ilachinski [2] has observed "an impressive array of emergent behaviors," such as frontal assaults, retreats, guerrilla-like attacks, flanking maneuvers, encirclements, and many more.

The contemporary trend towards the integration of multi-role forces, together with the high cost of live exercises, has required the development of more realistic training environments.

Using intentional software agents [8,9] in simulation greatly enhances the capability for modeling entity and group behaviors based upon situation awareness. This makes it feasible to express tactics where entity activity is a combination of goal-directed and reactive behaviors dependent on the developing tactical situation.

Intelligent agents allow the Computer Generated Forces in training systems to behave in a more human-like manner, with a much richer set of behaviors including team responses, and dynamic role re-allocation. The result is a more effective training environment with realistic tactical behavior represented, whilst avoiding the expense of having humans involved to provide this.

Ralph Rönnquist [9] in his work, describes the Simulation Agent Infrastructure (SAI), which uses intelligent agents to improve war-gaming with enhanced tactics modeling. SAI offers a modern war-gaming solution with a clear cut separation between the simulation models, the simulation engine, and the simulation scenarios. This makes it easier to use and maintain the software, and it facilitates verification of individual scenarios. Further, it

compartmentalizes behavioral models and models of tactics into distinct agent capabilities, and allows entities in a simulation to be guided not merely by what time it is, but also by the current simulated situation.

The emphasis on timely, accurate information in modern warfare, and the availability of modern communications, have led to the development of more and more complex command and control systems. It is important to understand the behavior of these C3 systems capability [3,7] under a variety of circumstances. However, as they are difficult to analyse manually, advanced modeling and simulation tools for C3 systems development are required. The challenge in C3 systems is to model the reasoning associated with different roles in the command and control hierarchy. Intelligent agents can represent the reasoning and command capabilities associated with their assigned roles in the hierarchy, allowing different command and control strategies to be quickly evaluated under varying circumstances. This power comes from the suitability of the BDI architecture for representing individual and team objectives and roles.

The team modeling extension also includes the capability of progressing a scenario with different 'team granularity' [9]. For instance, a scenario may involve a number of platoons whose behaviors are detailed at the macroscopic level (as 'indivisible' entities), together with other platoons that are aggregations modeled through the behaviors of the individual soldiers. Such aggregation granularity may also be dynamic and change throughout a simulation run. The team modeling framework [10,11] then provides the concept and language constructs that make it possible to define this kind of simulation model.

1.2 Previous Work done in this Area

Realistic military simulations are needed for analysis, planning and training. Defense organizations are primarily interested in conducting successful and efficient military operations for which intensive analysis and training is required. All Analysis and Training Tools are based on various modeling and simulation techniques. One of the candidate technologies for modeling the decision-making behavior of simulated battlefield entities is the Intelligent Agent Technology. Intelligent Agent Technology is a valuable software concept with the potential to be widely used in military simulation & and command decision modeling. They provide a powerful abstraction mechanism required for designing simulations of complex and dynamic battlefields. . During battlefield simulation these entities generally represent individualistic behavior, taking operational order from higher control and executing relevant plans. Their ability to model the tactical decision-making behavior of battlefield entities gives an edge over many other software techniques because such a problem maps easily into agent based programming. This study demonstrates the strength of this technology in modeling and simulating the battlefields. As a case study [12] the tactical and reactive behavior of lower level battlefield entities such as tanks has been modeled using JACK^{TM} Intelligent Agent Framework.

In this study, the tanks have been modeled as Intelligent Agents that have tactical behavior, plans and capabilities. The Red tank tries to reach its target point by traversing the shortest path (proactive behavior). However, if it comes across an obstacle while moving, its initial plan of reaching the target by the shortest path fails and it is forced to react to the external event: “encounter of obstacle”. It displays reactive behavior by moving around the obstacle. Since its goal is persistent, it still tries to reach the target by the shortest path around the obstacle. Whenever it detects the enemy in the firing range, it neutralizes it.

During battlefield simulation lower level entities (soldiers/ tanks) generally represent individualistic behavior, taking operational order from higher control

and executing relevant plans. A complex battlefield scenario typically involves thousands of entities, their coordinated team behavior should also be considered to make the simulation more realistic. Teamwork requires both coordination and shared goals. The primary contribution of first study is in demonstrating a proof-of-concept model for simulating Armour tanks as agents, as simply as possible. It can be extended to model various kinds of battlefield entities, organized as a collection of agents. This will result in the development of team-oriented behavior, which is a powerful representation of the military command and control hierarchy. In addition to individual beliefs, goals, plans and intentions, a team of agents will also have mutual beliefs, joint goals and combined plans. Our next study[13,14] demonstrates the use of Intelligent Agent based team-behavior modeling concepts in simulating the Armored tanks in a tactical Masking Scenario. In this study, we have considered a scenario in which a Combat Group (CG) of Combat Command (CC) has been assigned the task of capturing an objective. (Fig. 1.1). Combat Group starts from forward assembly area and sends a Recce troop (one section). This troop detects some enemy and informs the Combat Group commander. Combat Group then sends two troops for masking operation so that main armour may move swiftly to the objective. Simultaneously, the masking team keeps on engaging enemy in enemy zone until the main armour moves out of enemy range. This masking team thereafter re-joins the main armour and moves on towards the objective.

In order to model and simulate this scenario using team-oriented concepts, first of all the key abstractions have to be identified. This will enable us to clearly structure the team and define roles and responsibilities of the team members. From the textual narrative stated above, we can directly identify the team controller as the Combat Group Commander, whose top-level goal is to move towards the assigned objective without any enemy interference. It is obvious that three sub teams will be involved, namely: recce team, masking team, and main armour team, each performing its respective role by executing the appropriate plans. For example, Recce team will handle recee events by

having plans for detecting enemy location and informing team controller. Similarly masking team will have plans for engaging the enemy so as to distract him. The masking team will join the main armour, when the enemy detection range between enemy and main armour is beyond reach or when the enemy has suffered more causality then desired threshold limit. The roles, responding events and the corresponding plans for this scenario is given in Table 1.1 below:

Main Team	Sub Team (Role Performer)	Roles	Responding events	Plans (event handlers)
Team Controller	recee_team mask_team armour_team	RECEE_ ROLE MASK_ ROLE ARMOUR_ROLE	recee_event mask_event armour_event	recee_plan mask_plan armour_plan

Table 1.1: Teams and their roles, events and plans

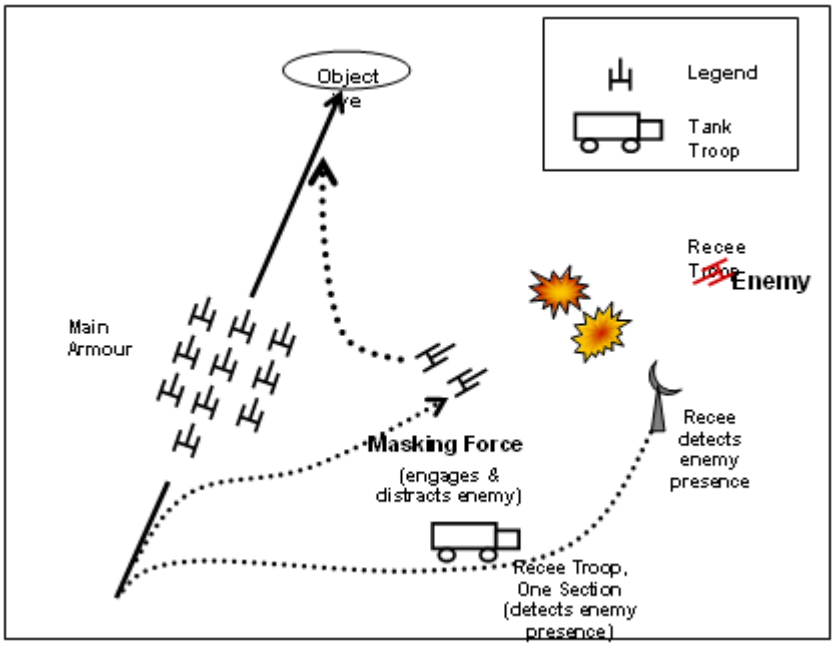


Figure 1.1 Masking Scenario Displaying Agent Oriented Team Behaviour of a Combat Group

Since team behavior is modelled as an extension of agent concepts, we have identified two types of agents in this armour-masking scenario: the tank agents and the team agents that have all the capabilities of agents and also

encapsulate team behaviour. The tank agents have been modeled so as to display tactical behaviour like movement along a route, obstacle avoidance, patrol, firing, etc. A team controller (representing the Combat Group) and three sub-teams are involved. The team controller is also called the “role tenderer”. It is composed of sub-teams that perform roles on its behalf. Role is a behaviour that the “role tenderer” may request the “role performer” to achieve. It represents behaviour of a team member (sub-team) participating in a particular tactical operation. As an example, in the above scenario, consider a troop of tanks. Depending on the battlefield situation, that troop may perform the role of recee, masking, engagement or marching. When a troop performs a given role, it presents a particular view to its battlefield environment at that instance. The other entities that are interacting with it expect certain behaviour at that time, depending on the role that it plays at that time. For example, an instance of the troop in the role of masking would have a different set of capabilities than if that troop was playing the role of main armour heading for assigned objective.

A Infantry ambush scenario [15] have also been modeled taking their team behavior into consideration. In this ambush scenario, when the infantry first platoon enters enemy kill zone, it encounters enemy ambush fire. First platoon immediately informs the team controller (commander) to change the predefined path of other infantry platoons, thus representating coordinated team behavior. These infantry scenario has been successfully modeled using JACK Teams.

1.3 Motivations

Motivation for our work comes from the successful implementation of above mentioned armour and infantry scenarios, exhibiting individual combat entity as well as team coordinated tactical behavior. These study demonstrates the strength of Agent technology in modeling and simulating the battlefields entities individual tactical and coordinated team behavior. In our study we

took a battlefield scenario, where each top level entity refereed to as command agent takes a dynamic reactive response on unforeseen condition based on the updated synthesized belief , derived from the lower level sub ordinate units belief, thus enabling more realism in simulation.

1.4 Scope of the work

This paper presents a role-based BDI framework to facilitates representation of military hierarchy, modeling of behavior based on agent current belief, teammate's belief propagation, and cooperation and coordination issues. This BDI framework is extended and based on the commercial agent software development environment known as JACK Teams. This BDI framework builds teams using a simplified, abstract framework called Team-Oriented Programming (TOP) and allows team based tactical operation of military doctrine to be captured in an effective way and be played out in simulation scenario with minimal effort. It also enable handling of dynamically changing combat situation , reasoning on team goal failure at the team level, as well as automatic sharing and aggregation of belief between team and sub teams for accessing of current battlefield situation .

This paper also demonstrates the use of intelligent agent-based team behavior modeling, team belief propagation based situation awareness and generation of expert based appropriate reactive response (past expertise stored in team belief) using a infantry attack scenario exhibiting a infantry company attack against a platoon. The company commander entity (CGF) is modeled as an command agent (CA), which synthesizes the belief derived from its platoons beliefs and generated immediate reactive response to any unforeseen battlefield situation. Similarly the Platoon commander is modeled as an command agent , which synthesizes the belief derived from its sections beliefs and generated immediate reactive response to any unforeseen battlefield situation. The sections in turn synthesizes the belief from its soldiers which are actual combat units & propagates the belief to its platoon commander.

1.5 Organization of the thesis

In this chapter, we have highlighted the usefulness of intelligent agents to model the tactical behavior & team behavior of the combat entity which serves as the motivation for the work reported in this thesis. Furthermore we have also outlined the specific objective of our research and related research work that has occurred in the past. Chapter 2 provides problem description and an overview of the methodology used in this study describing the model infrastructure, command agents, classes of data/information at different level of military hierarchy, Observe ,Orient ,Decide and Act (OODA) Information model. It also briefly introduces the command agent architecture used in the study. Chapter 3 introduces the Team oriented programming approach of modeling Intelligent agent & their team behavior. It describes Team work in detail, which is a central feature of many activities in the modern military. Chapter 4 introduces the Intelligent Agent technology describing in detail the BDI Framework, Agent Team Framework, JACK Team concepts & Implementation. Chapter 5 introduces Tropos: An Agent-Oriented Software Development Methodology in detail. It also identifies the stakeholders & actor diagram in the proposed system. Chapter 6 introduces Agent Unified modeling language (AUML) , an extensions to the UML for designing of agents. Chapter 7 provides an overview of the software detailed design using the JACK™ Development Environment (JDE). It describes the details of plans, events, (messages, percepts), data/knowledge of all agents / teams involved in this system. Chapter 8 gives the implementation details of the system using JACK Tool Kit .Finally, Chapter 9 concludes the thesis and gives some suggestions for future work.

Chapter 2. Methodology

2.1 Concept Demonstration (Scenario)

Concept demonstration of the command agent software

To demonstrate the functionality and capability of the command agent, we have chosen a deliberate attack scenario (see Figure 2.1). The objective is for a mounted infantry company located at Point A to attack an enemy formation occupying a position in the vicinity of Point B. The company commander agent is to produce a plan and courses of action to carry out four phases for the attack, namely 1) preparatory, 2) assault, 3) exploitation and 4) reorganisation5. The company organization consists of three platoons. Each platoon is comprised of three sections, each of which has nine soldiers. To prosecute the attack, the company splits into a fire support platoon and two assaulting platoons. The agent plans the routes, form-up positions and coordination parameters for the attack. The agent will monitor the location and status of its own troops and the enemy and will respond to situations which require changes to the plan. In planning and executing the attack, the agent will apply documented military doctrine and make appropriate use of terrain.

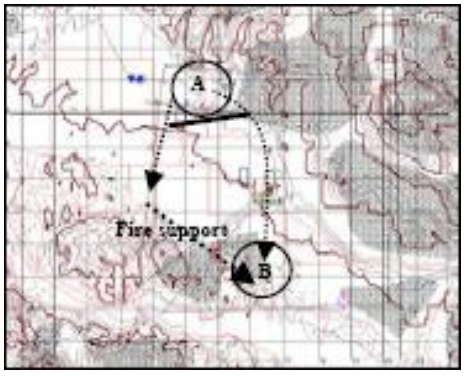


Figure 2.1: The scenario for demonstrating the agent capabilities. The objective is for a mounted infantry company located at Point A to attack an enemy formation occupying a position in the vicinity of Point B.

Objective : An infantry company attacking an enemy area (protected by platoon) using two assault platoon and one support platoon.

2.1.2 Detailed Scenario

If we see a larger picture a Battalion is going to attack a enemy company. There are three company in Battalion (two assault company & one support company). If we seen the military command hierarchy ,each company (Figure 2.2) there are three platoon. Each platoon in turn consisting of three section (two assault section & one support section). In each section there are 9 soldiers. Each section consists of two grenadiers, three gunners & two MMG men & four rifle men.

Mission plan to capture a objective is developed in Battalion HQr. As a part of main mission objective each company has to capture its objective , with in which each platoon of company has to capture its sub objective designated by company commander. If we further zoom our focus to platoon level , each platoon has been assigned the task of capturing one objective protected by enemy section.

When the first platoon starts executing its plan to capture the given objective, it may encounter several obstacles, minefield created by enemy units. Initially the three section of the platoon moves in some given formation , towards the objective. If the first section encounters the minefield, then as part of tactics all other section aligns them in rod formation to allow minimum causalities, showing the coordinated team effort.

There is a firebase group with major weapons (MMG/LMG) situated outside the minefield but near to enemy area. This fire base group fires at enemy, while the three sections are crossing the minefield. When one of the section (assault role) encounters the mine field, it request the platoon commander to give arty fire support from fire base.

While the first section crosses the minefield (Figure 2.3) with slow speed, it may further encounter ambushed enemy firing from the hidden enemy

consisting of three to four soldiers. This enemy action creates some suppression / causality to the assault section. But the section keeps on moving till the suppression of the section goes beyond a particular threshold point. In this case this assault section stops movement & start firing towards the enemy area, while the support section with high morale value gives the support fire to assault section. Meanwhile ,If the enemy ambush section suffers heavy loss , it moves back to safe place through safe route to avoid further damage to its personnel.

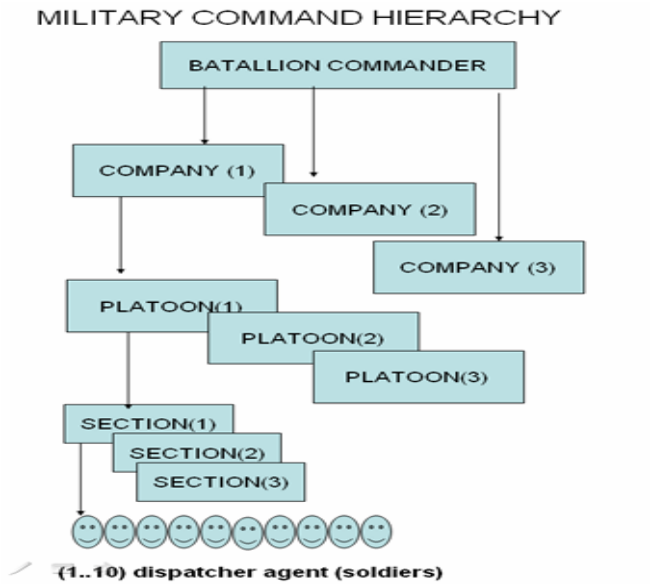


Figure 2.2 : Military command Hierarchy

This support fire action of the support section continues until the enemy is suppresses from further firing or the support section suffers heavy causality. The assault section crosses the minefield, it request platoon commander to stop the arty firing and also issues order to support section to stop engaging the enemy ambush section further. Finally all the three section crosses the mine filed and spread in front of enemy section, covering it from front side so as to isolate the enemy section. While the platoon is spreading in lean on fashion , the enemy arty / mortar group may fire at the three sections, causing their causality. The enemy arty / mortar group fires at three section, only when the arty fire on them is fully stopped.

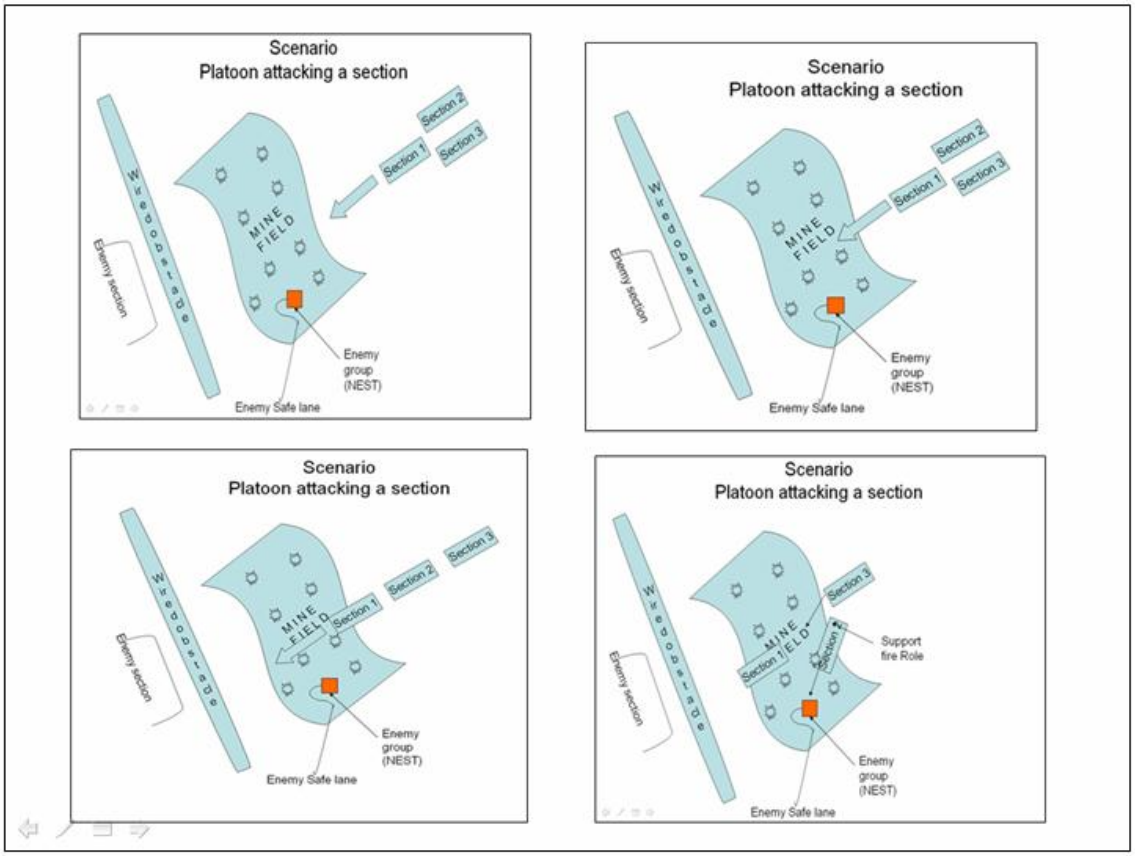


Figure 2.3 Platoon crossing a enemy minefield

The demonstration scenario for this Model is a company attack in plain terrain. The scenario chosen was such that a company which has three platoons will perform an attack on an enemy force whose positions and intent were assumed to be known. The command agent's (CA) role is to deploy forces and tactics according to military doctrine. The CA needs to generate a plan for maneuvering the troops taking into account the enemy's LOS and their weapon fire ranges. Other resources issues to be considered are: fuel, ammunition, food, health, morale, etc. In a company attack, the commander is impounded with information from different sources. It is therefore necessary to fuse the information on which knowledge is based.

Having the knowledge, one is then able to acquire an adequate understanding of:

1) Belief about health, morale, leadership factor, suppression factor, casualty & location of its own units.
2) Resource level of its own units such as fuel, ammunition, food,
3) Enemy's intent based on their locations, movements and weapons deployed;
4) The availability of friendly forces and resources

2.2 Model Infrastructure & Implementation

In an attempt to model the behaviors of a company commander, one must consider the following:

1) Sensor data consisting of lower level actual entities (soldiers),section unit data,
2) Platoon reports, data, and situation awareness.
3) Information compilation from raw data, reports to commander belief data
4) Information retrieval from Commander Belief data set
5) Integration of propagated belief from lower level entities to upper level
6) Effective plan generation derived from commander Belief /knowledge
7) Decide and select the most viable plan and
7) Act upon those COA and control and monitor the plans.
8) Handle uncertainties / enemy intent
9) Modify current plan, if required

Model is implemented using 'JACK Teams', which is based on multi-agent framework and BDI (Belief, Desire, Intention) reasoning. JACK Teams [6,7,8] was developed for operations analysis in the military environment. It provides mechanisms for modeling key aspects of team operations deployed in land operations such as:

1) A hierarchical command structure,
2) Team oriented activities,

3) Team intentions,
4) Extensive reasoning over plan failure,
5) Team reformation and re-organization,
6) Connection of beliefs between teams up and down the command hierarchy; and
7) Autonomy at each command hierarchy level.

2.2.1 Command Agents

Within the Command Agent (CA), we identify three key capabilities (Figure 2.4): planning for action, control of action and reporting on action. These three capabilities operate on a shared belief structure that contains the CA current beliefs regarding the WG simulation.

Planning for action: A number of possible scenarios are examined and evaluated in the processes of planning and re - planning. Re-planning is important in that it allows the CA to deal with wide variety of circumstances. Hence increasing the robustness of the CA.

Controlling of act: The CA is to manage the teams in a timely manner whilst keeping track of the planned activities. This involves observing and monitoring the planned activities throughout the execution phases of the plans.

Reporting on action: In a multi-agent framework representing Military command & control has a hierarchical structure, reporting mechanism must make provision for collaboration between agent-and-agent and agent-and human.

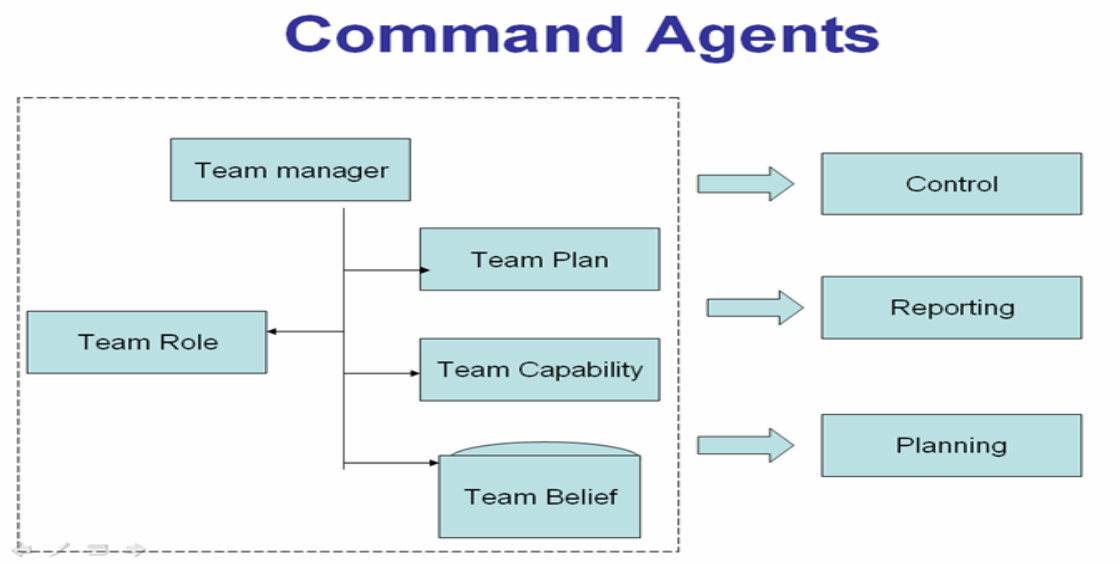


Figure 2.4 Command agent Capability

2.2.2 Classes of data / information at different levels

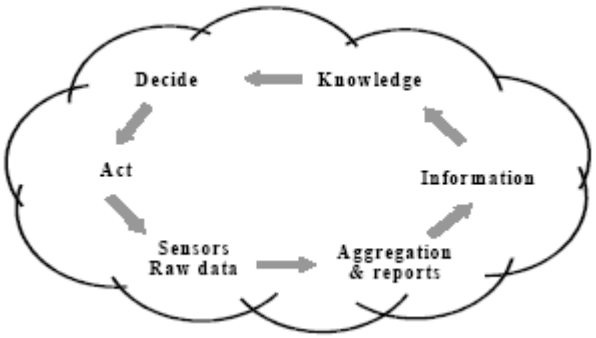


Figure 2.5 : OODA Model

In the following, we will explain the classes of data/information (Figure 2.5) at different levels namely: sensors, aggregated reports, information, knowledge, decide and act.

Sensor data: Sensor data or raw data is obtained by the platoons. This data is in a very crude unprocessed form.

Aggregation and reporting: In the platoons, the raw data is aggregated and condensed into summary reports which typically give information such as enemy's physical location, velocity, contacts, incidence, status, attacks, moves, etc. These reports usually do not contain enemy intent.

Information: At this level, information contains such things as terrain line-of-sight (LOS), weapon effective range (footprints), speed limits in terrains, slope gradients in terrain, etc. They can be further summarized and turned into tables (matrices) of discrete data informing the commander of things like: enemy movements, contacts, detail notes about enemy intent, friendly forces' health, morale, status and so on. This information is supposed to give an adequate level of understanding about the situation. From this, the commander is then able to assess the situation.

Knowledge: In the knowledge level, the commander will generate plans based on the information and facts derived using a cognitive model. In our case, a cognitive model based on CWA is used to evaluate the COA based on the particular scenario. Subsequently, the commander is able to generate plans, execute plans and monitor the progress of team activities throughout execution of the plans.

Decisions: In decide level, the commander needs to select the most viable plan for a scenario. It is desirable to have a selection process that includes human factors such as personality and importance. This will give a degree of variability in terms of how human decision is biased by personality differences.

Act: The commander must act upon the COA in a plan which was selected by the decision processes.

There will be following teams & actual combat units in the scenario
Teams: (Command Agent type team)

- **Battalion Commander**
- **Company Team**
- **Platoon Team**
- **Section Team**

Actual combat Units as Agent (Non Team Type)

- **Soldiers Agent (Actual combat Units)**

2.3 Observe ,Orient ,Decide and Act Information model (OOAD)

A similar four-step decision making process is used by researchers in military applications and is called the Observe – Orient – Decide - Act (OODA) loop or four box method [16]. The cycle was developed from Boyd's studies of air-to-air combat in the Korean War to assist pilots to achieve knowledge superiority and avoid information overload in order to win a battle.

Each of the team of command Agent type will follow information model shown in figure below:

OOAD Loop (see figure 2.6) , which stands for :

- Observe (O)
- Orient (O)
- Decide (D)
- Act (A)

This is the first level of a agent based model in which tangible and intangible functions are classified. For instance, an commanding agent (company level) entity observes the environment when looking out for threats, infantry in sections and platoons will send out sensor data (section causality, morale, suppression status ,speed, minefield cross status ,friendly force location ,enemy location, enemy detection status) and contact reports in a very crude form.

Similarly, the dispatcher agent role (Soldier) is to deliver the raw data such as location, status , morale, causality status, minefield cross status etc to its section team. The section team role is to process data to gain adequate level of situation awareness (belief based military doctrine) of battlefield environment & take necessary action to be performed by the actual combat entities (soldiers).

These reports are processed and the CA needs to orientate itself and assess the situation. From this, the CA acquires an adequate level of situation awareness. Next, information/knowledge is processed based on the commander beliefs such as friendly unit position, status, morale, mine crossing status, enemy position, enemy's intent, weapon, front line status, etc. Commander agent stores all above information in its belief structure. Changed or modified belief may sometime lead to triggering of some critical events, which may further lead to new goal or sub goal generation fro the combat units. Present friendly, enemy status parameter & current battlefield situational parameters are matched with commander's past combat result beliefs and the most accurate matched belief is retrieved as solution. This solution is the most viable plan as per the current environmental, combat condition & constrains In the "decide", the CA determines which plan is the most viable one to be deployed.

Finally, in 'Act' layer, the CA executes the courses of actions specified in the plan. In this level, the COA (which is translated to WG simulation commands) is executed via the interface between CA and the WG simulation.

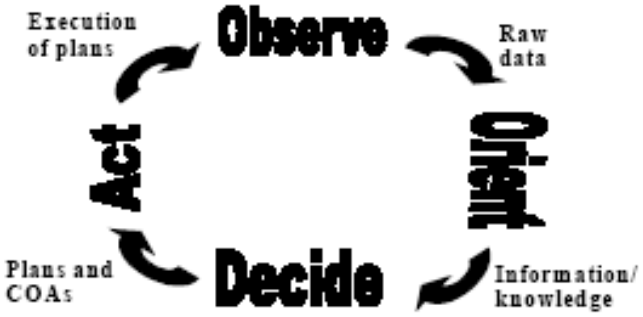


Figure 2.6: Flow diagram of OODA Model

2.3.1 Command Agent Belief about Minefield crossing status of the soldiers (Example Scenario)

This example scenario depicts the decision phase of Platoon command agents ,when its sections are crossing the enemy mine fields. Initially the

section has belief that none of its soldiers has crossed the minefield (Table 2.1). But during the course of time, if any soldiers encounters the minefield , it informs its respective section about mine filed crossing status. The section then reorients its soldiers position to arrange in rod formation so as to get less casualty further in battlefields. The section also informs its platoon team commander about its mine filed cross status. The platoon commander then updates (Table 2.2) its belief about section minefield cross section. The changed minefield status of the platoon triggers an event to other section, which eventually change their formation to rod so as to minimize causality.

Each platoon has following belief structure , which stores the minefield belief crossing status of its three sections as;

```
Belief   PlatoonSectionsMinefieldCrossBelief  extends Openworld
{
Value field  int  section1_pre_status;
Value field  int  section2_pre_status;
Value field  int  section3_pre_status;
Value field  int  section1_cur_status;
Value field  int  section2_cur_status;
Value field  int  section3_cur_status;

Post event ChangeFormationtoRodEvent ev;
;
;
}
```

PlatoonSectionsMinefieldCrossBelief has value in its Tuple:

section1 _pre_status	section2 _pre_status	section3 _pre_status	section1 _cur_status	section2 _cur_status	section3_ cur_status
0	0	0	0	0	0

Table 2.1 : Mine Cross Old Belief

If section 1 crosses the mine filed then the platoon belief PlatoonSectionsMinefieldCrossBelief is updated to:

section1 _pre_status	section2 _pre_status	section3 _pre_status	section1 _cur_status	section2 _cur_status	section3_ cur_status
0	0	0	1	0	0

Table 2.2 : Current Mine Cross Belief

In **PlatoonSectionsMinefieldCrossBelief** belief structure , there is a callback , which is triggered as soon as a new belief about minefield is updated by any of its sections :

In this callback following condition triggers/posts the event to platoon ,which inturn directs other sections to change their formation to rod

#post event ChangeFormation ev;

```
If(section1_pre_status== 0 & section2_pre_status==0& section3_pre_status
==0 & section1_pre_status==1 & section2_pre_status==0 &
section3_pre_status==0)
{
// event ev handled by other sections to change their formation to rod

post(ev.ChangeFormation());
;
}
```

In next iteration , the belief tuple of platoon is updated to :

section1 _pre_status	section2 _pre_status	section3 _pre_status	section1 _cur_status	section2 _cur_status	section3_cur _status
1	0	0	1	0	0

Table 2.3 : Updated Mine Cross Belief

Note : Since the belief is again changed , but the above condition does not hold , so no event will be fired.

2.4 Team Roles

Figure 2.7 depicts the relationships between company team, commander-team, platoon-team and section-team. Notice that we have made the distinction of “Teams” and “agents”. In contrast to Teams, agents also have the same BDI framework except they do not have team behaviors. The dispatcher agent’s role is to deliver raw data from the war-game simulation to the section-team. The section-team’s (which consists of 3 sections) role is to process the raw data, to gain an adequate level of situation awareness (orientate) and consequently, inform the respective platoon-team agent the status of both friendly and opposition forces. The platoon-team’s (which consists of 3 platoons) role (see table 1) is to process the raw data, to gain an adequate level of situation awareness (orientate) and consequently, inform the company-team agent the status of both friendly and opposition forces. The Company-Team’s role (which consists of 3 platoon leaders and a commander) is therefore to generate plans and COA, and then submit them to the Commander-Team (which consists of the company commander and its teams) who will decide which plan and COA are supposedly the most viable one to take given the constraints. In return, the Company-Team then sends out movements (orders) to the platoon-team.

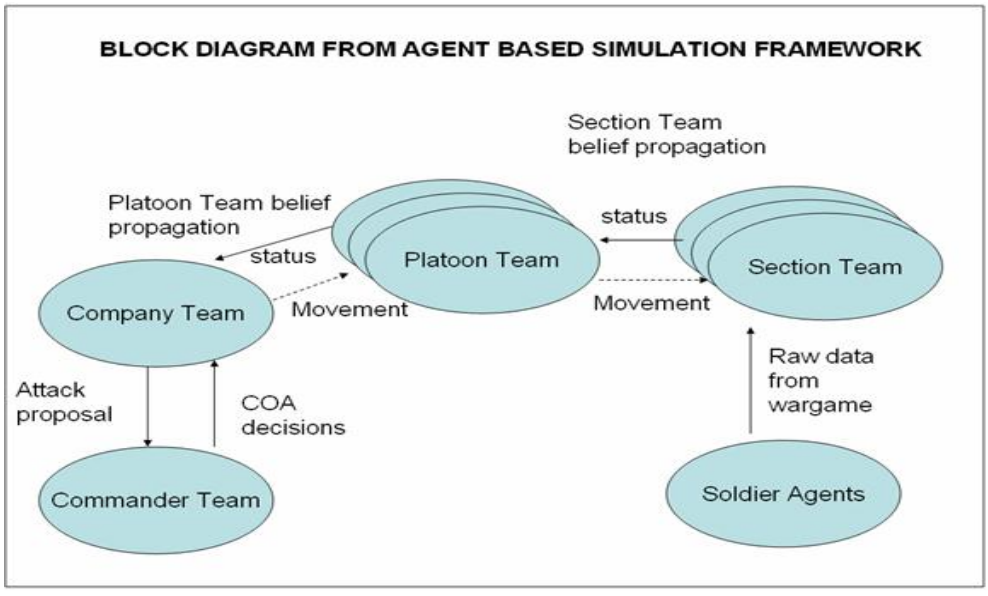


Figure 2.7 : Mission order delegation between commander-team, company team , Platoon-team and section-team

2.4.1 Role structure Requirement

Level in Hierarchy	Role requirement	Role performer (Team) /Agent	Role / Agent Instances
Battalion	CoympanyRole	CoympanyTeam	Company1, Company2, Company3
Company	PlatoonRole	PlatoonTeam	Platoon1, Platoon2, Platoon3
Platoon	SectionRole	SectionTeam	Section1, Section2, Section3
Section	--	Soldier Agent	(1..10 Soldiers)

Table 2.4 : Role structure Requirement

2.5 Software architecture of command Agent

2.5.1 Command agents

The command agents (CA) operate within a well-defined command and control structure which can be modeled as a hierarchy of teams. This command and control structure is also expressed within the war game simulation at the lower levels constructive entities and by human players at the upper levels.

As an example, one may choose to model battalion behavior using human players, company and platoon behavior using command agents and platoon member behavior using OTB entities. In this situation, the platoons are represented in both the simulation layer and the command agent layer. Decisions regarding platoon behavior (eg move to form up point, retreat) are made by the command agents, but decisions relating to individual platoon member behaviors (eg maintain formation, contact drill) are made by the behavior models within the simulation. The platoon command agents are

aware of which simulation entities are under their control and monitor their position and status. This information is used to make local decisions. The platoon CAs interact with their company command agent; they receive commands and send aggregated reports in accordance with appropriate military doctrine. The company CA is responsible for planning and executing a mission; in this regard it determines the tasks that are required to be performed and allocates them to platoons.

The command agent architecture is summarized in Figure 2.9. Note that the focus of this diagram is the command agent layer.

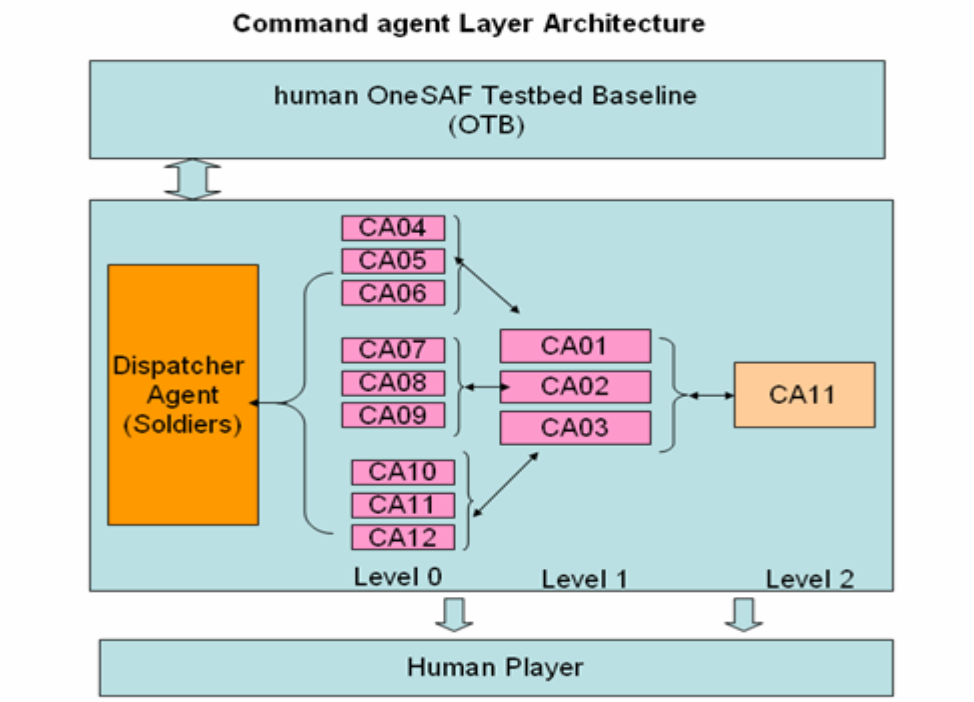


Figure 2.8: Command agent Layer Architecture. Levels 0 and 1 refer to the level within the command and control hierarchy (eg Platoon and Company)

In Figure 2.8, Company CA11 (level 2), consists of three Platoons, CA01 ,CA02, CA03 (level 1). The members for CA01 to CA12 are modeled in the simulation layer and are not shown. Platoons CA01 consists of three sections namely CA04, CA05 and CA06 (level 0) modeled as Commanding agents. Similarly Platoons CA02, CA03 are modeled as Commanding Agents. Each section further consists of dispatcher agents (10-12 Soldiers) . Orders and

reports are exchanged between platoon agents CA01, CA02 , CA03 and their members via the dispatcher over bi-directional links. Similarly Within each command agent, we identify three key capabilities:

1) Planning for action,
2) Control of action and
3) Reporting on action.

These three capabilities operate on a **shared belief structure** that contains the command agent's current beliefs regarding the world. The architecture is summarized in Figure 2.9

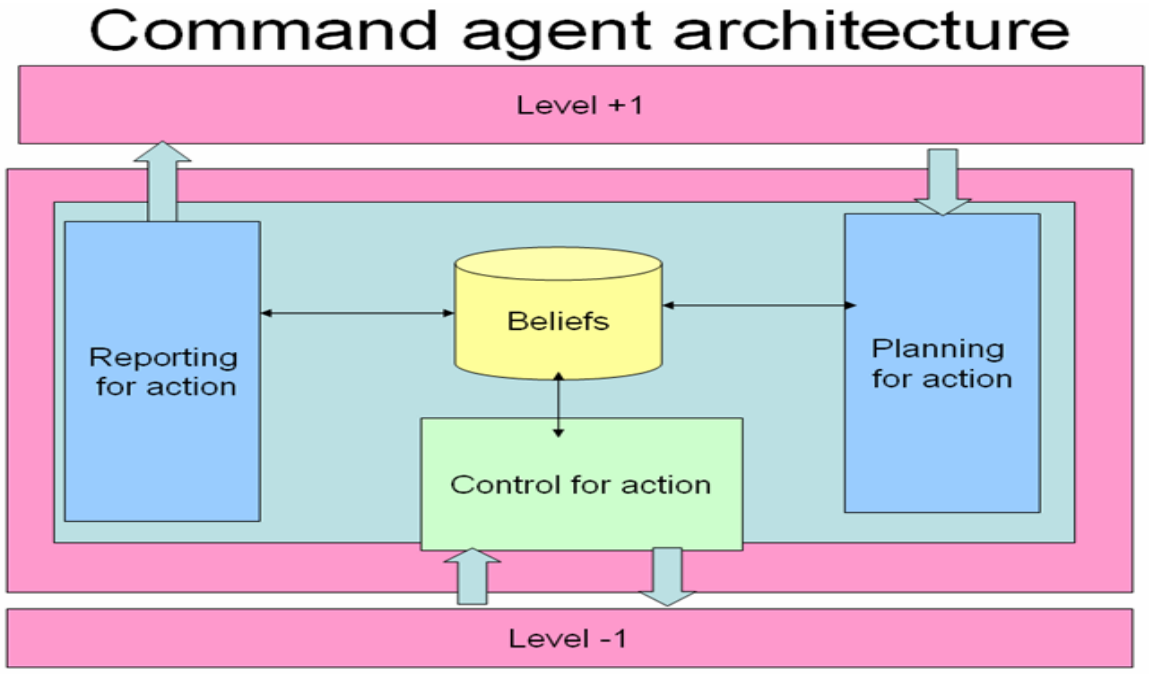


Figure 2.9 : Command Agent Architecture

The planning-for-action capability allows the command agents to take a command from the level above and by using the appropriate military doctrine, generates commands for the entities under its direct control. The progress of

the resulting action is then monitored by the control of action capability. Implementation of the planning and control capabilities is described in further Section titled JACK Teams.

The reporting on action capability provides reports back to the command level. The content of these reports is derived from messages provided by the subordinates using the message aggregation process.

2.5.2 (Reporting for action) : Message aggregation

Message aggregation (MA) performs the role of aggregating data derived from the entity models into reports issued at platoon or higher levels. MA Reporting serves two purposes:
1) Reducing the volume of raw data on target acquisitions produced by the entity models which otherwise cannot be used by the operator or agent, and
2) Formatting the data into accepted military reports.

The following reports are produced

- Situation Report,
- Location Status Report,
- Hostile Air Report,
- End of Hostility Report, and
- Contact/Incident Report.

2.6 Modeling Framework concept

The proposed modeling framework is an extension to JACK targeted the modeling of “systems with internal organization”. To this end , the modeling language includes the concept of teams as reasoning entities that form organizational; structure by taking on roles within enclosing teams. This organization modeling includes the means to capture both static & long term obligation structure, such as those that compose the military command & control hierarchy, and the transient skill based groupings that are formed to

perform individual missions. An obligation structure is then firstly defined as a type, with each team type including the definition of its inner structure in terms of roles it requires to perform team tasks. An actual obligation structure is established by instantiating individual teams and sub teams, and then linking them to each other in accordance with roles taken on by the sub teams. Sub teams which are last in the hierarchy may also take actual Agents (not teams) in order to perform the mission objectives. For example the scenario implemented in our the study, the three teams type are identified as Company Team, platoon team, section team. These team type are role performer as well as role requirer at the same time. For example the platoon team performing Platoon Role , but at the same time requires three section team and one platoon H Qtr team to perform the task of platoon team. The section last in military hierarchy contains 1-10 soldier agents acting as combatant entities. Soldier agents perform their low level tasks such as simple move towards an objective, attaining a particular formation (rod, two up, one up) ,moving across the mine fields/ wire mesh , encountering obstacles, moving across the river , detecting engaging enemy in range , reporting to its section teams etc . At the same time these soldier agents also fulfill higher level tasks (move towards an objective, support fire role , stop move, change formation etc) given by the section team commander. These agents are part of sub teams and are governed by the order given by the section team.

The modeling framework mentioned in this study allow the lower level entities to propagate their belief / data [casualty, morale, leadership, fatigue, suppression factor, location etc) to upper level teams (containing teams). In this way the higher level teams are updating their latest information / intelligence of battlefields by synthesizes the belief derived from lower level sub teams. The belief of higher level teams may also be inherited to lower level teams. The soldier agents inform their belief data [causality, morale, leadership, fatigue, suppression factor, location etc) to their section commander, which in turn propagates the section belief to its platoon.

In addition to the modeling of team structure, the modeling framework includes statements for expressing how a team operates by way of the concerted activities of its sub teams. A team reasons about the coordination between its members, and ultimately decides upon appropriate team plans by which the members in concert achieve the required missions.

The expression of team activity, and its coordination, includes all the performance primitives of JACK agents [17,18,19]. In addition, it offers statements of parallel activity and issuing of directives or sub goals to its to its sub teams. Notably, rather than combining the activities of cooperating agents into emergent teamwork, the activity of a team is directly attributed to it. Also , it is modeled as team activity separately from the sub teams (performing their roles). The consequential benefit is that coordinated activity can be programmed and explored with reference only to roles involved, independent of the sub teams eventually performing the activities.

2.6.1 Applications of benefits of the modeling infrastructure

The initial aim for the proposed modeling framework was to support the modeling of tactics in computer simulation of military operations. These tactics are typically team's tactics that involve coordination of sub teams activity. The modeling framework also includes the mechanism of belief propagation by lower level teams to its upper level teams ,thereby keeping the containing teams abreast of the battlefield latest situation
This latest information / belief help the containing teams to take dynamic reactive decisions under any unforeseen conditions.

2.6.2 .Command Agents Interaction with war-game simulation

A command agents is defined to be an intelligent agent capable of receiving situational awareness from a war game (through belief propagation

mechanism of lower level teams) or simulation and to use this information to carry out some planning and then interact back into the simulation environment (figure 2.10) to effect some change in as individual simulation entity or units (here soldier agent). Their role is to replace the crude, in built, behavioral mechanism within a war game or simulation with a more flexible, doctrine based reasoning agent that can autonomously plan and control interactions.

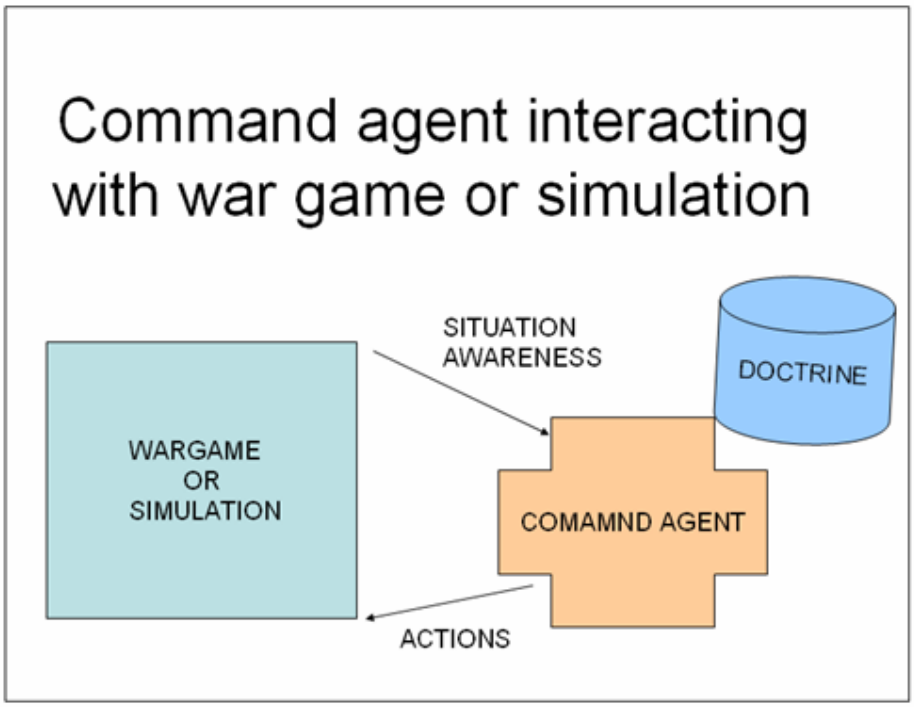


Figure 2.10 : Command agents interaction with war-game simulation

The concept of command agent is introduced to reduce the workload of the human war game operator by allowing the agent to autonomously plan and execute a high level instruction passed to war game units. The unit is typically composed of a number of individual entities with a command component, and hence the employed modeling framework handles the team interactions and coordination, which is required to carry out the order.

As an example, consider a command to a mounted infantry company to attack a specific enemy position. The command agent (CA) dealing with this instruction must follow doctrine in order to plan a basic concept for the

execution. It might then pass on sob - goals for this concept to its sub team members (who are command agents) who can the plan their components of the attack. One unit may be assigned the role of providing covering fire , whilst another may flank the third unit who will carry out the actual assault.

This model aligns well with the concept of directive command , employed by most military forces , where a commander directs his intentions to his sub ordinate commanders , leaving them to work out their own plans and not giving them direct instructions on what to actually do to execute the mission. By using a hierarchy of command agents, linked through the team modeling framework, it is practicable to break down the actual planning and problem solving into smaller chunks, and focus on each military aggregated unit in turn, rather than try and map the whole command process into one agent representation.

2.7 Model Assumptions

2.7.1 Team structure

Only static pre defined team structure is used for team formation.

Command team up to company team is taken in the simulation

Soldier , the actual combatant entity are defined as agent , having behavior such as move towards objective, move through minefield, stop move, adjust position as per section order , detection , engagement etc.

The complex belief generation derived from Belief propagation mechanism of the teams only takes three main sub teams (sec1,sec2, sec3) into account .Therefore for the decision making of command agent , the belief of only three sub teams is used.

2.7.2 Combat attrition model

The detection is based on maximum firing range of the soldier.

The detection range for all soldier in red and blue forces is taken as same.

Plain terrain has been considered in this scenario.

The suppression of soldier is linearly proportional to their vicinity to enemy firing range (i.e if they are within minimum firing rage , suppression id 100%, otherwise the suppression is decreasing as they are moving away from enemy firing range).

The soldier casualty in enemy mines and due to enemy engagement is based on Monte Carlo model (probability based).

The mine density throughout the mines is assumed to be same.

The morale of the soldier is assumed to be inversely proportional to suppression.

Initially fatigue factor of all soldiers is assumed to be 0 %.

The leadership factor of all soldier initially assumed to be 100%

The effect of fatigue & leadership is not taken in simulation. Only they are used for belief propagation mechanism.

Chapter 3. Team –Oriented Programming

3.1 Teamwork

Teams are more than just a collection of individuals pursuing their own goals. A commonly accepted definition of teamwork is a collection of (two or more) individuals working together inter-dependently to achieve a common goal [20]. The structure of a team may range from rigid, with clearly defined roles and a hierarchical chain-of-command, to flexible, where individuals all have similar capabilities, tasks are allocated flexibly to the best available team member, and decisions are made jointly by consensus.

Teamwork is a central feature of many activities in the modern military. Many of the activities in modern military combat operations involve teamwork. Teams exist both within well-defined units, as well as cutting across echelons. For example, a command staff at battalion and brigade levels consists of multiple officers working together to help the commander make decisions, and interactions and coordination between officers in the same staff section (e.g. G2/S2-Intelligence) at different levels can also be characterized as teamwork. At the lowest level, an infantry platoon, reconnaissance squad, or even tank crew works as a team to achieve a variety of tactical objectives. At the highest level, combined-arms operations and joint operations rely on integrating various assets and capabilities for maximum effectiveness. Accurate models of teamwork, including distributed decision making and information flow, are needed for developing and evaluating new equipment and procedures through human-behavior representation (HBR) studies. Teams are viewed as groups of inter-dependent individuals working together to accomplish a common goal. Team members must possess a mutual awareness (shared mental model), which enables them to interact, anticipate each other's actions and needs, and carry out team processes like communication, coordination, and helping/back-up. These processes underlie more advanced teamwork activities, such as distributed situation awareness and command and control, of particular relevance to the military.

3.2 Shared Goals

The notion of shared goals is essential to teamwork because it is what ties the team together and induces them to take a vested interest in each other's success, beyond acting in mere self-interest. Members of a team do not just act to achieve their own goals, possibly at the expense of others, but rather they look for synergies that can benefit others and contribute to the most efficient overall accomplishment of the team goal. In addition to this positive cooperativity, members of a team also have incentive to actively try to avoid interfering with each other. Furthermore, commitment to shared goals leads to other important team behaviors, such as backing each other up in cases of failure. For example, if one team member assigned to do a task finds that he is unable to complete it, other members of the team are willing to take over since they ultimately share the responsibility.

The Teams extension provides a team-oriented modeling framework. Team-oriented programming is an intuitive paradigm for engineering group action in multi-agent systems. Team-oriented programming is conceptually powerful, as it allows the software engineer to specify:

- What a team is capable of doing;
- Which components are needed to form a particular type of team;
- Whether a team is willing to take on a particular role within another team;
- Coordinated behavior among the team members; and
- Team knowledge.

In short, the concept of team-oriented programming serves to encapsulate coordination activity. It extends the agent concept by associating tasks with roles. However, the flexibility of multi-agent systems is retained. Although team members act in coordination by being given goals according to the

specification, they are individually responsible for determining how to satisfy those goals.

A team's structure can contain teams in any combination and in any number. The hierarchy is not restricted to a two-level design or in fact to a hierarchy. Layers of teams can be encapsulated within other levels, and the structure can be added to or altered at any time during the process. In other words, teams can be created in many layers, where each layer is encapsulated within the next layer, and so on.

Both conceptually and explicitly in a model, teams entities exist independent of their team members. For instance, teams can reason about how they belong as members in enclosing teams, or about which teams they include as sub-teams. The teams concept encapsulates coordination activity, and extends the agent concept by associating tasks with roles.

3.3 Relationship between Teamwork and Command & Control

Teamwork is often associated with command-and-control (C2). Historically, C2 has been seen as a hierarchical process of commanders directing their subordinates on the battlefield (though generalized command-and-control also has many non-military applications as well). However, more recently there has been an increasing appreciation of the distributed nature of information collection, often done by a staff in communication with various Recon elements in the field that supports decision-making. Often decisions must be coordinated laterally between multiple adjacent units involved, and occasionally there is a need to push decisions further down to smaller units closer to the battle, who have a better sense of tactical opportunities and consequences of actions. Hierarchical command is now even viewed by some as inflexible and sub-optimal. It was previously necessary for maintaining control in chaotic environments, but is no longer so clearly necessary with the advent of more powerful C3 networks and information technology, enabling instantaneous consultation and coordination over a

distance. See further discussion in the report “The Command Post is Not a Place” [21].

Command-and-control is a complex topic in its own right [22]. In a military context, C2 can be defined as the control of (spatially) distributed assets (weapons and sensors) in the most effective way to achieve tactical goals, which in the case of ground combat involves containing, attacking, defending, clearing, or denying enemy access to areas of 2D terrain (including assets on it, such as towns, airstrips, communication towers, ports, etc.)

One of the best known NDM models of C2 is the Recognition-Primed Decision-Making (RPD) model [23]. According to this model, the C2 process consists of a series of stages, beginning with:

- information gathering and situation assessment
- detection or identification of the situation as one a small number of expected “types”
- proposal of a solution (some appropriate response drawn from experience or practice)
- evaluation and refinement of the solution by projection of consequences (how the situation is expected to develop) and events into the near future (via “mental simulation”)
- execution of the response and continued monitoring of the situation to ensure it proceeds as desired.

3.4 Team Processes

To better understand how teams work, researchers often make a distinction between taskwork and teamwork [20]. Taskwork refers to activities individuals do in the course of performing their own parts of the team’s mission, more or less independently from others. Team members must of course train for these activities as a pre-requisite to working in the team. However, teamwork refers to those activities explicitly oriented toward interactions among team members and are required for ensuring the

collective success. Teamwork processes include: communication, synchronization, load balancing, consensus formation, conflict resolution, monitoring and critiquing, confirming, and even interpersonal interactions such as reassurance. It is argued that these activities must be practiced as well to produce a truly effective team. It is an unfortunate reality that most training in industry and the military focuses on training individuals for taskwork (such as acquiring knowledge of individual procedures in a cockpit), while relegating teaching of teamwork to on-the-job training (e.g. indoctrination by peers) in the operational environment.

3.5 Simulating Team Behavior with Multi-Agent Systems

Recent advances in intelligent agent research have opened up possibilities for more sophisticated simulations of teamwork and cooperative behavior. Agent models of teamwork are based on key concepts such as joint intentions [24] and shared plans [25], which formally encode how teams do things together. These concepts are derived from the BDI framework [26], which postulates the importance of representing and reasoning about mental states such as beliefs, desires, and intentions when interacting with other agents. Jennings' (1995) GRATE system exemplifies how useful BDI concepts (especially joint responsibilities) can be to producing complex coordinated behaviors (the main application of GRATE is a distributed industrial manufacturing and distribution system). Another popular environment for developing and evaluating models of agent teamwork is robotic soccer [27].

Perhaps the most widely known agent-based teamwork system is STEAM [24]. STEAM is multi-agent system built on top of SOAR, a production-system-based agent architecture, to which it adds rules for establishing and maintaining commitments to joint intentions. STEAM produces robust behaviors even in unanticipated situations by automatically generating communications among team members to reconcile beliefs about achievability of goals and to re-assign tasks. For example, this was illustrated in the behavior of a simulated company of Army attack helicopters in a situation

where the lead aircraft gets shot down; with STEAM, the company was able to re-group and continue with the mission. STEAM is also used in TacAirSoar [28], which is a module that can be used to control aircraft and produce tactical behavior in distributed simulations of air combat missions

Other multi-agent systems that employ some form of teamwork include RETSINA [29], SWARMM [30], and CAST. All of these have been applied to military combat simulations. In RETSINA, agents work to support humans by gathering information or constructing plans that will achieve goals in a combat environment. The agents' activities are fairly de-coupled, each working more or less independently on separate parts of a task; opportunities for helping each other are discovered through a "match-making" intermediary. RETSINA has been incorporated into the CoABS grid (http://coabs.globalinfotek.com). SWARMM was specifically designed as a system for simulating air combat teams. It breaks teams of fighters down into well-defined roles, such as lead aircraft (commander) and wingman, which determines each team members' actions in a plan (mission or maneuver). In CAST, more general role assignment is permitted through a flexible language for team structure and process description. The agents decide dynamically during a scenario who is the most appropriate member to carry out a task among several that can play the role, and the others then automatically play backup. CAST also uses the description of the team as a rudimentary form of a shared mental model to automatically infer information exchange opportunities and derive information flow based on analysis of needs of teammates. An alternative model of teamwork is developed within JACK Teams TM which instead of requiring shared goals and intentions amongst members, introduces a concept , team entity. It is this team entity that holds the team goal and executes the team plans. The team entity then coordinates the team members in doing their parts to achieve the team goal. The model is hierarchical, so team members may themselves be teams.

Chapter 4. Intelligent Agents

4.1 Software Agents

Agents are characterized by their situatedness, autonomy and flexibility. Situatedness refers to the interaction that the agent has with its environment including communication with other agents. Autonomy refers to the ability of an agent to initiate and control its own actions, and flexibility refers to the ability of the agent to deal with new or unexpected situations.

4.1.1 Beliefs Desires and Intentions (BDI)

The belief-desire-intention (BDI) model of reasoning agents [31] is derived from folk psychology and cognitive science. It explains the behavior of rational agents (human or artificial) in terms of the concepts of beliefs, desires and intentions, and uses this as a model to generate rational behavior. The use of intelligent agents focused on the modeling of human reasoning. The power of this model is the ability to describe folk-psychological notions of **belief, desire and intention(BDI)**, which helps to describe some aspects human decision making. In terms of implementation, BDI agents are goal-oriented, meaning that once they are set a goal (or desire), that goal will persist until it is achieved. The agents use reactive planning to determine how to reach that goal, usually by setting themselves a number of sub goals. If they fail to reach any of these sub goals they will try other alternatives that may eventually lead to the success of their original goal.

BDI agents differ from traditional artificial intelligence (AI) models with the concept of intentionality. This rationale allows the search space to be pruned and action to be taken, thus allowing efficient real-time behavior. It is the flexibility of the BDI architecture however that makes it appropriate for an agent that must display human-like behavior. The world is complex and in general cannot be planned for because something unexpected will always

arise. The goal-directedness of BDI agents enables them to deal falling in a heap.

4.1.2 JACK Teams Concepts

Jack Intelligent AgentsTM is an extension of the Java programming language, which provides agent oriented programming, constructs for developing agent applications. It also provides a goal oriented execution engine which persists in trying all possible ways to achieve a goal choosing the method most suitable at the current situation. It is based on the Beliefs, Desires, and Intentions (BDI) model [32]. The BDI agent model is an event-driven execution model providing both reactive and proactive behavior. In this model, an agent has certain beliefs about the environment, has goals (desires) to achieve, and has plans (intentions) describing how to achieve goals. JACK is one of a family of implemented BDI systems which include PRS [33], JAM [34], dMars [35] and Jadex [36]. JACK Teams is an extension of JACK Intelligent Agents TM which provides constructs and support for Team Oriented Programming In JACK Teams a team is a distinct entity with its own representation. It incorporates the standard BDI reasoning mechanisms of JACK and other similar systems, with respect to behaviors such as choice of plans and persistence of goals if a particular plan fails. The team is in fact the core entity in JACK teams and an individual agent is simply represented as a team with no team members. We describe here some of the key concepts in the team model implemented by JACK Teams.

4.1.3 Implementation Approach to the JACKTM Agent Language

The JACKTM Agent Language extends JavaTM to provide agent-oriented programming support. These extensions are both syntactic and semantic. Syntactic extensions include keywords (e.g. *Agent, Plan, Event*) and attributes that define relationships such as which plans can be triggered by a given event signature. Semantic extensions support the specification of reasoning methods that conform to the BDI paradigm, rather than Java's imperative model: each step is interleaved within the BDI execution model, allowing a

plan to be interrupted in response to new events.

The JACKTM Agent Compiler maps JACKTM Agent Language constructs onto pure JavaTM classes and statements that can be used by other JavaTM code. The JACKTM Agent Kernel is a set of classes that, amongst other things, manages task concurrency and provides a high-performance communications infrastructure for inter-agent messaging. This kernel also supports multiple agents within a single process, allowing agents that share much of their code to be grouped together.

4.1.4 JACKTM Agent Language Components

This section provides a brief overview of the JACKTM Agent Language [19].

A minimal JACKTM application is defined in terms of one or more *agents* / *teams*, *plans*, *events*, and either *belief sets* or *views*. Optionally, the application can also include *capabilities*. Agents and teams are used to represent the autonomous computational entities of an application. The Team class is used to encapsulate the coordinated aspects of (multiple) agent behavior. Teams include much of the functionality of agents; for convenience, we refer to such functionality as being a property of *agent*/*teams*. Programming constructs in the JACKTM Agent Language include:

Team

Teams are an extension of the BDI paradigm that facilitate the modeling of social structures and coordinated behavior. JACKTM introduces the notion of teams as separate reasoning entities (separate from team members). Teams are characterized at the highest level by the roles they can perform, and the roles they require their team members to perform. They also contain a set of team plans for doing tasks related to achieving specific goals, or reacting to specific events. A team has a set of members which are (or can be) in a long term relationship to the team. In JACK Teams the team members are specified as belonging to a role container. Team members may be added and removed dynamically. These members can be assigned to (or requested to participate in) particular tasks (via JACK's task teams), according to the

role(s) they can perform, and the roles required by a task as defined in the team plan. JACKTM includes the communication facilities needed for executing coordinated activity in an application.

Agent

Because JACKTM is based on the BDI paradigm, a JACKTM agent has beliefs, desires and intentions. These are part of the internal state of the agent and are not directly accessible by other agents in the system. Beliefs, as described by Bratman et al. [37] are represented by the agent's plans, belief sets and views. These define the knowledge that the agent has—procedural knowledge in the case of plans, and facts in the case of belief sets and views. The agent's procedural knowledge defines the action sequences that can achieve its desires. Although JACKTM does not have an explicit representation of desires, at any given moment in time a JACKTM agent's desires are embodied in the set of plans that are applicable to the current internal state of the agent. Each applicable plan loosely corresponds to a desire, i.e. an activity the agent would embark upon if other desires were not also competing for the same computational resources. When an applicable plan is selected it becomes an intention, i.e. the agent commits to satisfying the desire using the selected plan.

Capability

Capabilities are used to organise the functional components of an agent (events, plans, belief sets and other capabilities) so that the components can be reused across agents. Since capabilities can contain sub-capabilities, an agent's competence can be defined as a hierarchy of capabilities. Capabilities were added to JACKTM in response to a pressing software engineering requirement to support the development of libraries of agent-oriented functionality that can be re-used across applications.

Team Members

Team members can be either teams - sometimes called sub-teams - or individuals. An individual is represented in JACK as a team that does not contain any members and does not require any roles. As team members can themselves be teams, a team can be a hierarchical (or more complex) structure.

Roles

A role specifies the part that a member plays, or can play, within a team. It is defined in part by the goals for which that role is able to be responsible (or equivalently the tasks which it can achieve, or the events which it can respond to). In JACK Teams the beliefs or knowledge of the agent required for the role are also specified as part of the role. JACK Teams also associates a role with the events or goals generated by that role.

Plan

Plans are *procedures* that define how to respond to events. When an event is generated, JACKTM computes the set of plans that are *relevant* to the event (i.e. those plans that *match* the event). Each relevant plan is further filtered by its *context condition*, i.e. a statement that defines the conditions under which the plan is applicable. The set of relevant plans whose context condition is satisfied by the current situation then becomes subject to a process of *deliberation*, where the agent selects the plan that will form its next *intention*. The JACK runtime infrastructure guarantees that plan step execution (including reasoning method execution) is atomic.

Team Plans

Team plans are a set of steps specifying how a task is to be achieved by members performing particular roles. Before a team plan can be executed. it must be established which team members, in which roles, will participate in

this particular task. JACK Teams provides an establishment method which can be customized if desired. This method assigns team members that can perform the roles required within the plan. This sub-group is called a task team within JACK Teams. Steps in the team plan are assigned to task team members via the roles as used within the team plan. JACK Teams provides a construct to allow members to perform steps in parallel if desired. As with standard JACK plans, additional Java code can be incorporated within the team plan if necessary. Team plans (like standard JACK plans) are associated with a single goal to be achieved, event to be reacted to, or message to be responded to.

Goals, Events and Messages

Goals and events to some extent capture respectively the proactive and reactive character of agents (and teams). Messages capture the communication between agents (or teams which are not related to each other in the team hierarchy) which also requires some reaction or response.
In JACK Teams (as in JACK) these are all represented by a similar data structure (Event and its subclasses) which contains arbitrary fields, and can thus be used for passing whatever information is needed beyond the particular goal/event/message type.

Event

Events are the central motivating factor in agents/teams. Without events, the agent/team would be in a state of torpor, unmotivated to think or act. Events can be generated in response to external stimuli or as a result of internal computation. The internal processing of an agent/team generates events that trigger further computation. JACKTM has two main categories of event: *Normal Events* and *BDI Events*. Normal Events are used to represent ephemeral phenomena such as environmental percepts; if the agent/team does not successfully handle the event with its first attempt, the event is discarded because the world will have changed in the interim.

In contrast, BDI Events are used to represent *goals* rather than transitory stimuli. When an agent/team services a BDI Event, it commits to successfully handling the event; this can involve trying a number of alternative solution paths until the goal is satisfied.

Beliefs

Beliefs in agent systems generally refer to all information the agent has about both its environment and its own state. Belief sets are used to represent the agent's declarative beliefs in a first order, tuple-based relational form. The value fields of a belief set relation can be of any type, including primitive Java types and user-defined classes. A belief set can be either *open world* or *closed world*, and the JACKTM kernel ensures its logical consistency. Belief sets provide a number of useful functions over and above standard information retrieval, for example, an event can be automatically generated on beliefset update, leading the agent to consider whether it should change its activities. Joint beliefs, as mentioned earlier are the beliefs held by all members of a team. Joint beliefs are not particularly important or supported in JACK Teams and its underlying model of teamwork, although they can be realized by belief propagation both up and down the team hierarchy. JACK, and also JACK Teams provides a specialized data structure called a belief set which is represented and can be accessed in similar ways to relations in a relational database. JACK Teams allows specification of how beliefs are to be propagated between a team and its members.

View

A *view* is a data abstraction mechanism that allows agents to use heterogeneous data sources without being concerned with their interface. In essence, they make the interface to an external data source the same as a belief set.

4.2 Agent / Team Execution Model

4.2.1 Agent Execution Model

A JACKTM application is made up of one or more autonomous agents/teams. The execution proceeds as follows:

– Add any newly generated events to the event queue.

– Compute the set of plans that match the event at the head of the event queue.

– Select one of these plans for execution (i.e. create an intention).

– If the selected plan matched a BDI Goal Event, then add the intention to the intention stack that generated the BDI Goal Event. Otherwise, create a new intention stack for the intention.

– Select an intention stack to execute. Select the intention from the top of the stack and execute the next step in that intention. This step may involve the generation of a new event.

– Repeat the cycle.

In the case of a BDI Goal Event, the selection of the plan that will form the new intention (part of the *deliberation* process) can be quite complex. Meta-level plans can be used to make an intricate choice from the set of applicable plans. On plan failure, the agent can also reconsider alternative plan choices in an effort to satisfy the goal. Alternatively, it can re-compute the applicable plan set (in the new context) and exclude the failed plan.

4.2.2 Team Execution Model

In contrast to agents, the team execution model consists of two phases, an initial team formation phase and a loop that corresponds to the agent execution model (but includes extra team-specific operations). In the initial phase the team is formed by selecting the team members. A team definition includes a number of *roles*, i.e. definitions of the events that entities must handle if they are to fill the *tendered* role. Each prospective team member has a corresponding definition of the events a team *tenderer* (i.e. the *containing* team) must handle if it is to take on the entity as a role *filler*. At runtime, team

formation is triggered by the posting of a TEAMFORMATIONEVENT by the JACK Teams infrastructure. This event is handled by a plan that selects the actual instances that will fill the tendered roles. If the user does not provide a plan to handle the TEAMFORMATIONEVENT, a system provided default plan is used. This initialization process is triggered automatically as part of the team instance construction. Furthermore, role fillers (also referred to as *sub-teams*) can be detached and attached at runtime, thereby supporting dynamic team formation and re-formation.

4.2.3 JACK Teams Plan Execution

When a team decides to execute a particular team plan, the first step is to establish which team members will participate in the team plan. JACK Teams calls this establishing the task team.
This is done by assigning team members from the relevant role containers, to each required role within the plan. An establishment method can be defined to choose amongst the members within a role container. Additional members can also be added to the team dynamically, in order to allow them to be used for the particular task.

Once the relevant team members have been identified for the particular task, the team plan can start execution. Steps within the plan request members to achieve particular goals. Requests are essentially messages containing the goal data structure, which has fields that can contain information relevant to the goal. This can also be used to pass back relevant information once the goal is achieved. Steps complete by either succeeding, in which case execution proceeds, or failing, in which case execution terminates, and a fail plan is executed. Failure of any step in a plan causes the plan to fail, at which point a new plan is searched for to achieve the same goal. When a team member receives a request to achieve a goal it uses its own reasoning processes to determine how to achieve that goal - including using its own team members to delegate to. The team entity is not concerned with how the

member carries out its responsibilities.

The control flow available in JACK Team plans includes the standard Java sequential, selective and repetitive constructs, plus also a parallel block. The parallel block allows various nuances. An AND variant requires all branches to succeed, whereas an OR variant requires only one. There are also variations regarding whether branches are terminated if a sibling branch succeeds or fails, as well as exception handling details.

4.3 Belief propagation

Once the team formation phase is complete, the team execution model repeats a cycle that is very similar to that for agents. However, team execution includes a belief propagation step that handles dissemination of information up and down the team hierarchy. A team can have access to a *synthesized* belief set that is derived from the beliefs of its sub-teams. JACKTM supports the definition of *filters* that determine if and when the propagation should occur, and what subset of beliefs should be propagated to the containing team. Similarly, sub-teams can inherit a synthesized subset of the beliefs of the containing team. Belief propagation is triggered by changes to a team or team member's belief set.

4.4 JACK Development Environment

JACKTM was augmented with a set of graphical tools that support the design, implementation and tracing of agent applications. The JACKTM Development Environment (JDE) provides a set of graphical tools for building agent-oriented applications. In this graphical interface, agents, team structures, and their components are represented by icons connected by lines that show their relationship to one another. This diagrammatic representation uses natural language to describe the goals, contexts, reasoning steps, and actions of agents/teams. The graphical and natural language descriptions can then be

fleshed-out by programmers to produce executable behavior models whose computational structure maps closely to the SME/analyst specifications.

Chapter 5. Tropos: An Agent-Oriented Software Development Methodology

5.1 Agent Oriented Programming

Agent Oriented Programming (AOP, from now on) is most often motivated by the need for open architectures that continuously change and evolve to accommodate new components and meet new requirements. More and more, software must operate on different platforms, without recompilations, and with minimal assumptions about its operating environment and users. It must be robust, autonomous and proactive. Examples of applications where AOP seems most suited and which are widely quoted in literature [38-40] are electronic commerce, enterprise resource planning, air-traffic control systems, personal digital assistants, and so on. To qualify as an agent, a software or hardware system is often required to have properties such as autonomy, social ability, reactivity, and proactivity. Other attributes which are sometimes required [40] are mobility, veracity, rationality. The key that makes a software system possess these properties is that it is conceived and programmed at a knowledge level [41]. Thus, in AOP, we talk of mental states and beliefs instead of machine states, of plans and actions instead of procedures and methods, of communication, negotiation and social ability instead of interaction and I/O functionalities, of goals, desires, and so on. Explicit representations of such mental notions provide, at least in part, the software with the extra flexibility needed in order to deal with the intrinsic complexity of applications such as those mentioned earlier. The explicit representation and manipulation of goals and plans facilitates, for instance, a run-time adaptation of system behavior in order to cope with unforeseen circumstances, or for a more meaningful interaction with other human and software agents.

An Agent-Oriented Software Development Methodology (Tropos) ,allows us to exploit all the flexibility provided by AOP.

In a nutshell, the two novel features of Tropos are:

1. The notion of agent and related mentalistic notions are used in all software development phases, from early requirements analysis down to the actual implementation. Our mentalistic notions are founded on belief, desire, and intention (BDI) agent architectures [42].

2. A crucial role is given to early requirements analysis that precedes the prescriptive requirements specification of the system-to-be. This means that we include in our methodology earlier phases of the software development process than those supported by other agent or object oriented software engineering methodologies

5.2 The Tropos Methodology: An Overview

The Tropos [43-45] methodology is intended to support all analysis and design activities in the software development process, from application domain analysis down to the system implementation. In particular, Tropos rests on the idea of building a model of the system-to-be and its environment, that is incrementally refined and extended, providing a common interface to various software development activities, as well as a basis for documentation and evolution of the software.

Tropos is intended to support five phases of software development:

- Requirements analysis
  - Early Requirements
  - Late Requirements
- Architectural Design
- Detailed Design
- Implementation

Requirements analysis in Tropos is split in two main phases: Early Requirements and Late Requirements analysis. Both share the same conceptual and methodological approach. Thus most of the ideas introduced for early requirements analysis are used or late requirements as well. More precisely, during the first phase, the requirements engineer identifies the domain stakeholders and models them as social actors, who depend on one another for goals to be achieved, plans to be performed, and resources to be furnished. By clearly defining these dependencies, it is then possible to state the why, beside the what and how, of the system functionalities and, as a last result, to verify how the final implementation matches initial needs. In the Late Requirements analysis, the conceptual model is extended including a new actor, which represents the system, and a number of dependencies with other actors of the environment. These dependencies define all the functional and non-functional requirements of the system-to-be.

The **Architectural Design** and the Detailed Design phases focus on the system specification, according to the requirements resulting from the above phases. Architectural Design defines the system's global architecture in terms of sub-systems, interconnected through data and control flows. Sub-systems are represented, in the model, as actors and data/control interconnections are represented as dependencies. The architectural design provides also a mapping of the system actors to a set of software agents, each characterized by specific capabilities.

The **Detailed Design** phase aims at specifying agent capabilities and interactions. At this point, usually, the implementation platform has already been chosen and this can be taken into account in order to perform a detailed design that will map directly to the code.

The **Implementation activity** follows step by step, in a natural way, the detailed design specification on the basis of the established mapping between the implementation platform constructs and the detailed design notions

5.3 The Key Concepts

Models in Tropos are acquired as instances of a conceptual metamodel resting on the following concepts/relationships:

Actor, which models an entity that has strategic goals and intentionality within the system or the organizational setting. An actor represents a physical, social or software agent as well as a role or position. While we assume the classical AI definition of software agent, that is, a software having properties such as autonomy, social ability, reactivity, proactivity, as given, for instance in [46], in Tropos we define a role as an abstract characterization of the behavior of a social actor within some specialized context or domain of endeavor, and a position represents a set of roles, typically played by one agent. An agent can occupy a position, while a position is said to cover a role.

Goal, which represents actors' strategic interests. We distinguish hard goals from softgoals, the second having no clear-cut definition and/or criteria for deciding whether they are satisfied or not. According to [47], this different nature of achievement is underlined by saying that goals are satisfied while soft goals are satisfied. Soft goals are typically used to model non-functional requirements. For simplicity, In the rest of the paper goals refer to hard goals when there is no danger of confusion.

Plan, which represents, at an abstract level, a way of doing something. The execution of plan can be a means for satisfying a goal or for satisfying a softgoal.

Resource, which represents a physical or an informational entity.

Dependency, between two actors, which indicates that one actor depends, for some reason, on the other in order to attain some goal, execute some plan, or deliver a resource. The former actor is called the **depender**, while the

latter is called the **dependee**. The object around which the dependency centers is called **dependum**. In general, by depending on another actor for a dependum, an actor is able to achieve goals that it would otherwise be unable to achieve on its own, or not as easily, or not as well. At the same time, the depender becomes vulnerable. If the dependee fails to deliver the dependum, the depender would be adversely affected in its ability to achieve its goals.

Capability, which represents the ability of an actor of defining, choosing and executing a plan for the fulfillment of a goal, given certain world conditions and in presence of a specific event.

Belief, which represents actor knowledge of the world.

5.4 Modeling Activities

Various activities contribute to the acquisition of a first early requirement model, to its refinement and to its evolution into subsequent models. They are:

Actor modeling, which consists of identifying and analyzing both the actors of the environment and the system's actors and agents. In particular, in the early requirement phase actor modeling focuses on modeling the application domain stakeholders and their intentions as social actors which want to achieve goals. During late requirement, actor modeling focuses on the definition of the system-to be actor, whereas in architectural design, it focuses on the structure of the system to- be actor specifying it in terms of sub-systems (actors), interconnected through data and control flows. In detailed design, the system's agents are defined specifying all the notions required by the target implementation platform, and finally, during the implementation phase actor modeling corresponds to the agent coding.

Dependency modeling, which consists of identifying actors which depend on one another for goals to be achieved, plans to be performed, and resources to

be furnished. In particular, in the early requirement phase, it focuses on modeling goal dependencies between social actors of the organizational setting. New dependencies are elicited and added to the model upon goal analysis performed during the goal modeling activity discussed below. During late requirements analysis, dependency modeling focuses on analyzing the dependencies of the system-to-be actor. In the architectural design phase, data and control flows between sub-actors of the system-to-be actors are modeled in terms of dependencies, providing the basis for the capability modeling that will start later in architectural design together with the mapping of system actors to agents.

A graphical representation of the model obtained following these modeling activities is given through actor diagrams , which describe the actors (depicted as circles), their goals (depicted as ovals and cloud shapes) and the network of dependency relationships among actors (two arrowed lines connected by a graphical symbol varying according to the dependum: a goal, a plan or a resource). An example is given in Figure 5.1.

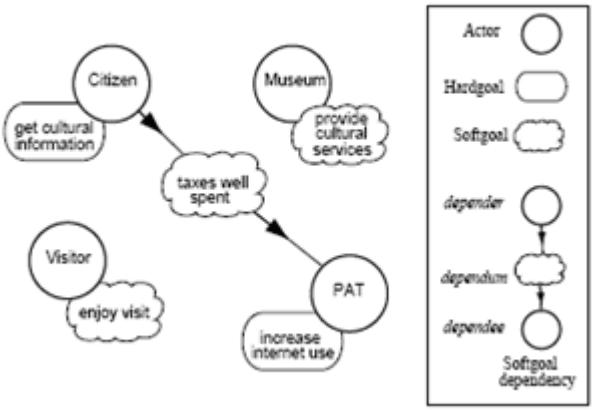


Figure 5.1 : Actor Diagram Modeling the stakeholder of the e Culture System

Goal modeling rests on the analysis of an actor goals, conducted from the point of view of the actor, by using three basic reasoning techniques: means-

end analysis, contribution analysis, and AND/OR decomposition. In particular, means-end analysis aims at identifying plans, resources and soft goals that provide means for achieving a goal. Contribution analysis identifies goals that can contribute positively or negatively in the fulfillment of the goal to be analyzed. In a sense, it can be considered as an extension of means-end analysis, with goals as means. AND / OR decomposition-combines AND and OR decompositions of a root goal into sub-goals, modeling a finer goal structure. Goal modeling is applied to early and late requirement models in order to refine them and to elicit new dependencies.
During architectural design, it contributes to motivate the first decomposition of the system-to-be actors into a set of sub-actors.

Plan modeling can be considered as an analysis technique complementary to goal modeling. It rests on reasoning techniques analogous to those used in goal modeling, namely, means-end, contribution analysis and AND/OR decomposition. In particular, AND/OR decomposition provides an AND and OR decompositions of a root plan into sub-plans.

Capability modeling starts at the end of the architectural design when system sub actors have been specified in terms of their own goals and the dependencies with other actors. In order to define, choose and execute a plan for achieving its own goals, each system's sub-actor has to be provided with specific ''individual'' capabilities. Additional ''social'' capabilities should be also provided for managing dependencies with other actors. Goals and plans previously modeled become integral part of the capabilities. In detailed design, each agent's capability is further specified and then coded during the implementation phase.

Following Stake-holder were identified for the MTS System.

Stake-holder

Players: who will actually the war-game to enhance their skills

War-Game centre : war game facility provider. War game centre wants Govt. funding to build / improve their service to the players.

Actor Diagram (Figure 5.2) specifying the stake holders and their main goal dependencies

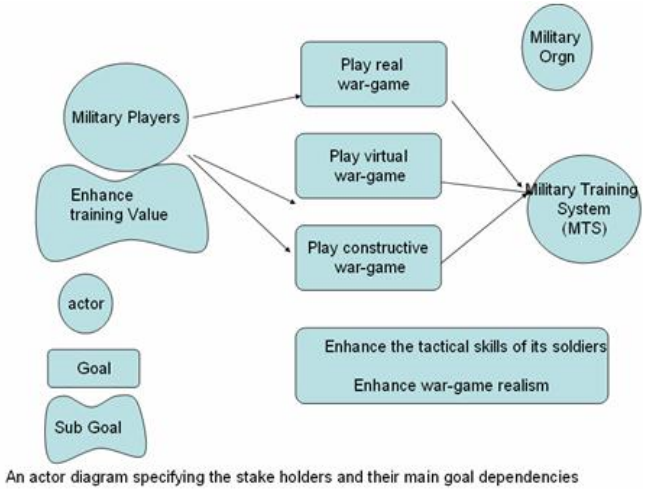


Figure 5.2: Actor Diagram specifying the stake holders and their main goal dependencies

Rational Diagram (Figure 5.3) for the proposed system are as follows:

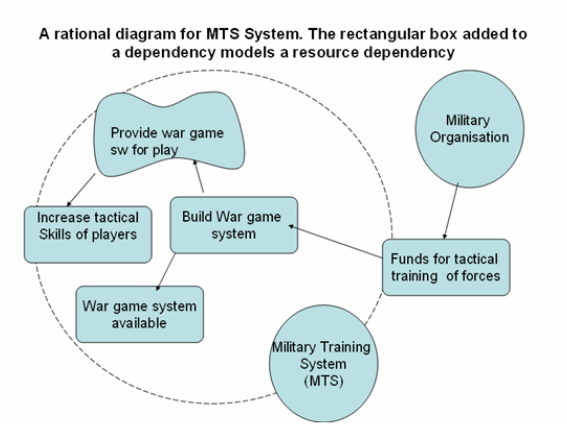


Figure 5.3 : Rational Diagram for the Military Training System (MTS)

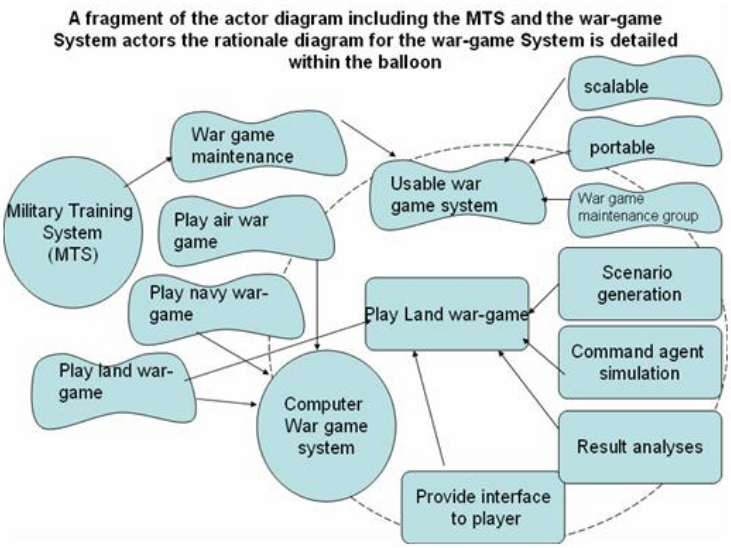


Figure 5.4 : Actor Diagram for MTS System

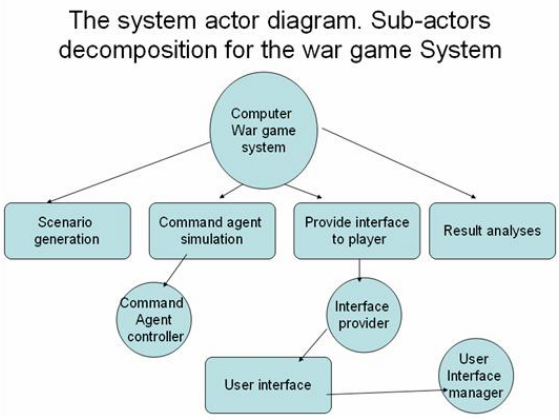


Figure 5.5 : Sub Actor Decomposition for War – game System

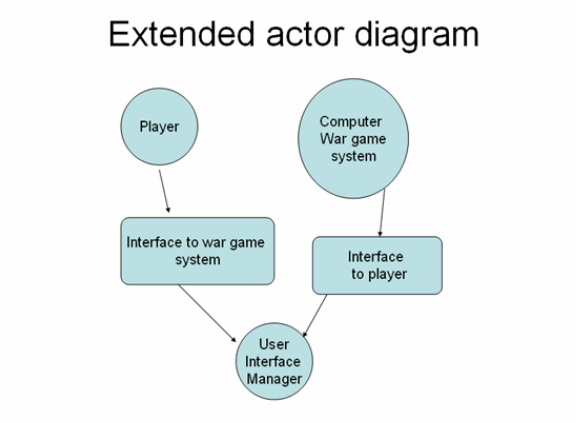


Figure 5.6 : Extended Actor Diagram

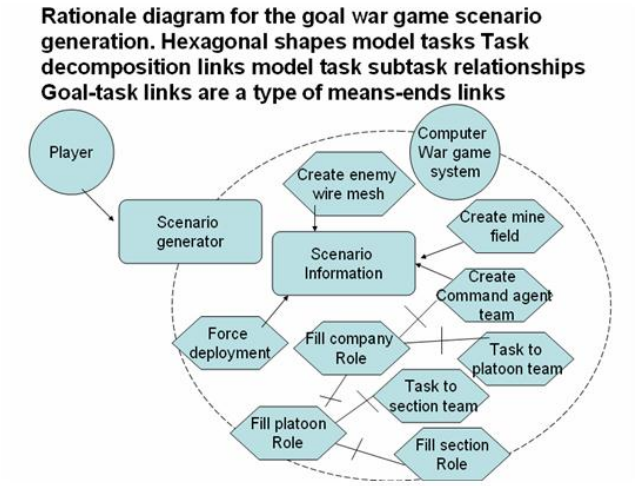


Figure 5.7 : Rationale Diagram for the Goal Scenario Generation

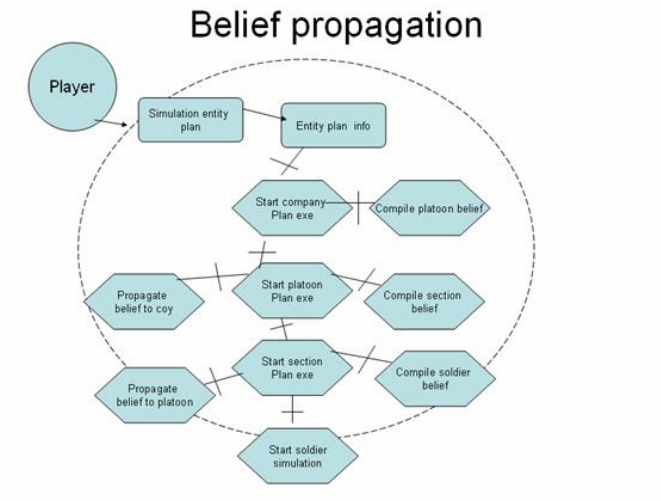


Figure 5.8 : Rationale Diagram for the Belief Propagation from section → Platoon→Company

Next phase is detailed designing. In Detailed design each agent of the system architecture is defined in further detail in terms of internal and external events, plans, beliefs and agents communication protocols. For detailed designing, Agent Unified modeling language (AUML), an extensions to the UML has been used for the design of agents and are implemented in the JACK language. Agent Unified modeling language is described in next chapter in detail.

Chapter 6. Agent UML (AUML) based Designing using Jack Agent Design Tool

6.1 Agent UML (AUML)

Mainstreaming and industrializing agent technologies requires suitable methodological and technological support for the various engineering activities associated with managing the complexity of any software system development. Despite its origins in object oriented software engineering the UML provides a rich and extensible set of modeling constructs that can be applied to agent oriented technologies. UML is a standard software engineering modeling language and when supported with tools such as Rational's ROSE provides the basis for much of the organizations software development. Agent Unified modeling language (AUML) [48][49] , is an extensions to the UML and is used for the design of agents that are to be implemented in the JACK language. These extensions provide the capacity to model the behaviour of agents for the purposes of design and, though the extensions are language specific, future generalisation and application to other agent languages can be supported as a industry-wide consensus about the nature of agency emerges over the next few years.

6.2 The Components of a JACK Agent

A JACK agent is a computational implementation of the above BDI model and as such it provides a reasoning framework with a specific set of language constructs. These constructs are just one possible implementation of the BDI model but provide the programmer with a modeling framework that is a mix of the high-level representational abstraction of the BDI model and the low level detail of the JAVA language. This paper will concentrate on the modeling of the specific high level agent concepts and ignore the modeling of the JAVA aspects of JACK programming . The following section lists and briefly

describes the major components of a JACK agent that need to be considered at design time.

AGENT The agent is the primary entity within the BDI model and is implemented within JACK as an autonomous module composed of capabilities, plans, databases, and events. Agents can address other agents and post events to them thus modelling inter-agent communication.

DATABASE The Database is the JACK implementation of the beliefs of the agent. These represent the agents view of the world as first order relational statements that are maintained as consistent through the specification of constraining key fields.

EVENT Events are those things that an agent responds to. They arise internally to an agent as reasoning progresses, as a result in a change in the agents beliefs, or on receipt of a communication from another agent.

PLAN A plan is a specification of a sequence of actions to undertake in response to an event. The plan contains a #handles event declaration that defines the event that the plan is suitable for. The system selects from amongst a number of suitable plans through examination of the context. The context method defines in detail the exact agent states under which the plan is applicable. The main part of the plan is its body. The body is a function that can mix standard JAVA code with Jack Agent Language (JAL) statements that can alter the agents beliefs, post new events, or send messages. From a design perspective the plans are the modular procedures that provide the building blocks for specifying the behvaiour of the agent.

CAPABILITY Capabilities are sets of plans, events, and databases that are functionally grouped to provide a specific capability to an agent.

6.3 Stereotypes: The UML Extension Mechanism

The main UML extension mechanism is that of the stereotype. A stereotype can be considered as a label that is applied to a UML modeling element that changes the meaning of the element to a definition specified by the domain in which the stereotype is applied in. A UML stereotype is indicated by placing the name in double angled brackets (<<stereotype>>), and can be applied to any UML element including classes, associations, use cases, attributes, and methods.

6.4 Jack Extensions to the UML

Work in extending the UML to accommodate agent oriented systems such as the AUML [1], has to date concentrated on high level architectural issues such as communication protocols between multiple agents. To date, there haven't been any extensions that will allow a software engineer to design an agent oriented system with UML down to the detailed design level. This is partly due to the fact that agents are a relatively young technology and there hasn't been any dominant agent languages (such as C++, Java, Python and Eiffel in the OO world) that have a large enough user base to influence international standards such as the UML. UML can be extended to accommodate Jack specific constructs through the definition of some Jack specific stereotypes. Although the stereotypes can be generalized to allow the modeling of any BDI based agent system using UML, in this paper the focus is on Jack. Generalizations will be looked at a later date. Due to the fact that Jack is built on top of an object oriented foundation in Java, extending UML to handle Jack is quite easy. Although it is possible to extend UML to accommodate a BDI language such as dMARS[35] it requires a bit more work than the Jack case. So what type of stereotypes do we need to add to the UML so that we can design Jack agents before getting into the code? The stereotypes required fall into a number of general categories.

High Level Jack Constructs - we require stereotypes to describe high level Jack modeling components such as an agent, plan, database, event and capability.

Associations - we need stereotypes to describe the special Jack relationships between plans, events, databases etc.

Low Level Constructs - we need some stereotypes to handle low level Jack constructs such as database fields, reasoning methods etc.
Once these stereotypes have been defined a mapping between the UML notation and Jack code can be built. This mapping can then be used by the detail designer or programmer to progress the agent system from architectural design, to detailed design and eventually to code.

High Level Stereotypes

High level constructs in Jack include agent, plan, database, event and capability. Since a capability is used to group functionally related agents, plans, databases and events, a <<capability>> stereotype can be defined for UML packages. In object oriented applications UML packages are usually used to group functionally related classes into packages, subsystems or modules. In addition to Jack specific constructs – agents, plans, databases and events can have attributes and methods just like any other Java class. Hence, it make sense to define class level stereotypes for these Jack constructs as defined in Table 6.1

Stereotype	Description
<<agent>>	Class level stereotype that defines a Jack agent.
<<plan>>	Class level stereotype that defines a Jack plan.
<<database>>	Class level stereotype that defines a Jack database.
<<event>>	Class level stereotype that defines a Jack event.
<<capability>>	Package/Subsystem level stereotype defining a Jack capability.

Table 6.1: High Level UML Stereotypes for Jack

Association Level Stereotypes

When building Jack agents, developing a set of plans, events and databases isn't enough to define the behavior of an agent. We need to show how these entities are related to each other. For example, we need to know which plans and databases are used by what agents, which plans handle specific events and what databases plans can modify. In UML uni-directional associations can be used to define relationships between agents, plans, databases and events. A set of stereotypes defined in Table 6.2 can be used to label these associations according to specific type of Jack relationship that needs to be represented.

Stereotype	Description
<<posts>>	Indicates a database posting an event.
<<uses>>	Indicates an agent using a plan.
<<modifies>>	Indicates a database which a plan can modify.
<<handles>>	Indicates an event handled by a plan.
<<private database>>	Indicates a private database owned by an agent.
<<uses agent>>	Indicates a plan using an agent implementing an interface.

Table 6.2: Association Level UML Stereotypes for Jack

Low Level Stereotypes

In addition to the class/package and association level stereotypes, some stereotypes at the attribute and method level are also required. For example, fields specified in Jack databases need to be distinguished between key and value fields. Attribute level <<key field>> and <<value field>> stereotypes are defined to achieve this distinction. Similarly in Jack plans, the ability to distinguish between regular Java methods and Jack reasoning methods (where special Jack commands can be used) is required. Again, we can define a method level stereotype <<reasoning>> that can be used to decorate a method to distinguish it from a regular Java method. A non exhaustive list of attribute and method level stereotypes is shown in Table 6.3.

Stereotype	Description
<<key field>>	Attribute level stereotype indicating a key field in a database.
<<value field>>	Attribute level stereotype indicating a value field in a database.
<<indexed query>>	Method level stereotype indicating an indexed query in a database.
<<posted as>>	Method level stereotype indicating how an event gets posted.
<<reasoning>>	Method level stereotype to indicate a reasoning method in a plan.

Table 6.3: Method Level UML Stereotypes for Jack

6.5 Software Detailed Design

6.5.1 Detailed design: The behavior of each architectural component is defined in further detail.

Detailed design process includes:

• Agent overview and capabilities
• Process diagrams for view of agent processing
• Develop plans and plan structure
• Details of plans, events, (messages, percepts), data/knowledge

Following teams & actual combat units were identified in the scenario:

Teams: (Command Agent type team)

- **Company Team**
- **Platoon Team**
- **Section Team**

Actual combat Units as Agent (Non Team Type)

- **Soldiers Agent / Actual combat Units**
- **Detector Agent**

- **Kill Damager Agent**
- **Master Agent**

In detailed design phase , details of capability, plans, events, (messages, percepts), data/knowledge of each team entity (company , Platoon & Section Team) and actual combat Agent (soldier, Detector, Kill Damager ,Master agent) were identified in fully detail. For detailed designing , The JACK™ Development Environment (JDE), is used for detailed designing of agents / teams.

The ***JACK™ Development Environment*** *(JDE)* is a cross-platform graphical editor suite written entirely in Java for developing JACK™ agent and team based applications. Extensive use of drag-and-drop and context-sensitive menus assist the construction of agents. The JDE allows the definition of projects, aggregate agents and teams, and their component parts under these projects.

The JDE is a purpose-built toolkit that facilitates the construction of agent/team models. In many situations an application will consist of a single model. However, the JDE also supports the co-operative development of the models required for an application. It also supports the reuse of model components and in the case of co-operative development, the sharing of components.

Company Team capability, event and plans and belief data base are :

```
public team CompanyTeam extends Team
   {
   // team declarations and definitions
   // company team perform company Role
        #performs role  CompanyRole;

   // company team requires follwong Roles to perform its tasks

          #requires role PlatoonRole pla1(3,3);
          #requires role PlatoonRole pla2(3,3);
          #requires role PlatoonRole pla3(3,3);
         #requires role PlatoonRole coy_hqr(1,1);

// events required fot company Team
```

```
    #handles event StartCoyExe;
    #uses plan StartCoyExeplan;

    #posts event StartCoyExe pfv11 ;

    #posts event ReadStatus1 rfv1;
    #handles event  ReadStatus1;
    #uses plan ReadStatusPlan1;


    #handles event UpdateCoyStatus ev;
    #uses plan UpdateCoyStatusPlan;

   // to update other belief from coy status belief ,CoyLevelStatusInfo  coystatusinfo()

    #handles event UpdateCoyOtherBeliefs;
    #uses plan UpdateCoyOtherBeliefsPlan;

    // post Update Coy cas belief event
     #posts event UpdateCoyCasBeliefEvent;

   // Handle Update cas belief event
     #handles event UpdateCoyCasBeliefEvent;
     #uses plan UpdateCoyCasBeliefPlan;

  // send msg to support fire to platoons to support Ist Platoon

      #sends event PlatoonSuportFireEvent;

    #synthesizes teamdata CompanyStatus
coy_status(pla1.pla_status,pla2.pla_status,pla3.pla_status,coy_hqr.pla_status);

    // this bel contains all platoons bel

    #private data CoyLevelStatusInfo coystatusinfo();

  // CASULITY BEL

     // post event GetCoysIstPlatoonSupEvent through belief set
"CoyLevelPlaCasStatusInfo.Bel"

     // when ist sec of paltoon suffers heavy loss , it requires fire support

      #posts event GetCoysIstPlatoonSupEvent;
      #handles event GetCoysIstPlatoonSupEvent;
      #uses plan CoyIstPlaSupPlan;

  // Other Belief made from main status belief , plstatusinfo();

    // Coy level pla cas belief
    #private data CoyLevelPlatoonCasStatusInfo CoyPlaCas();

      int No_Of_Soldiers;

 // four platoons name holder incl HQr platoon

      public String RolePerformer[]= new String[4];
```

```
    // all JACK agent declarations can also be used

    // constructor
    public CompanyTeam (String name,int No_Of_Soldiers1)
    {
    super(name);
    No_Of_Soldiers=No_Of_Soldiers1;

    }
  public void Start_Company_Exec(String s1)
  {
        postWhenFormed(pfv11.CoyExec(s1));
}
}
```

6.5.2 AUML Diagrams

6.5.2.1 Company capability

Design: Coy_capabilities

CompanyTeam capabilities consisting of events , plans as behaviour, and private / synthesised data / belief .

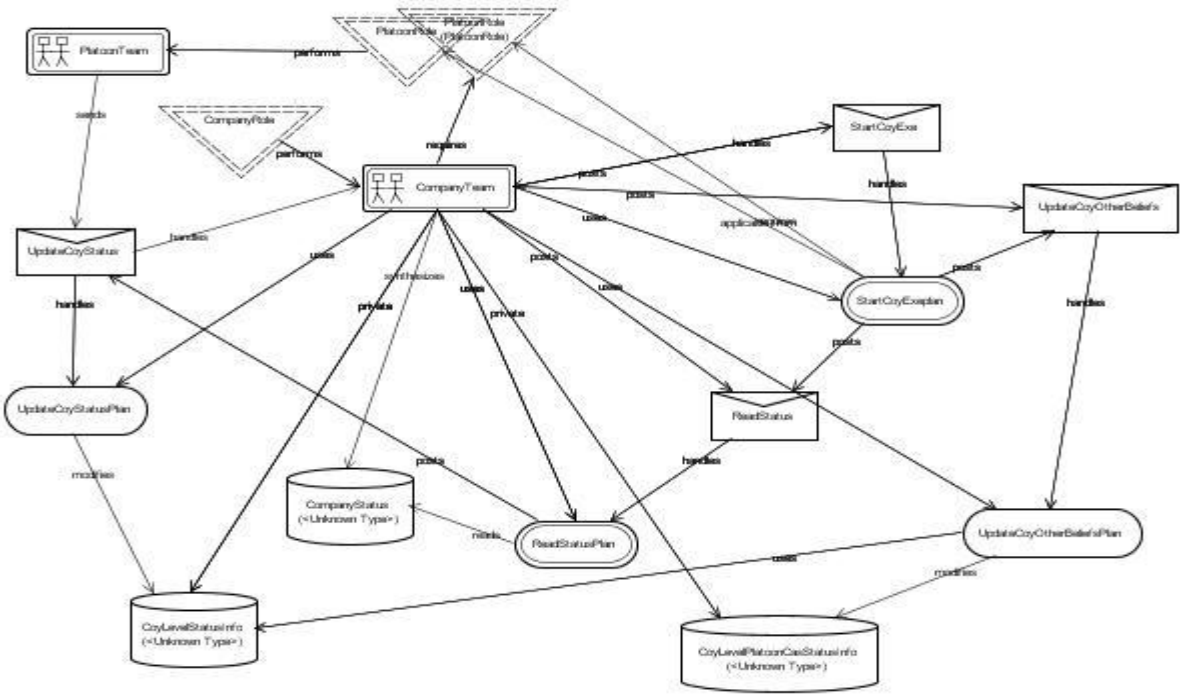


Figure 6.1 : Company capability

Company team performs company role & requires four platoon teams performing Platoon Roles. It has its own private belief structure

CompanyStatus.Bel (Figure 6.1) used to synthesize/ derive belief of its Platoon Teams .It also contain individual information about its four platoons in belief structure CoyLevelStatusInfo.Bel. This belief is updated frequently by company Team whenever the Platoon teams sends the event UpdateCoyStatus to company team. The event UpdateCoyStatus is handled by company plan UpdateCoyStatusPlan modifies the belief CoyLevelPlatoonStatusInfo.Bel. CoyLevelPlatoonStatusInfo.Bel contains information of all platoons, such as platoon name, morale,leadership value,suppression factor, fatigue value ,causality value,loc_x & loc_y .

Initially company team posts itself an event StartCoyEvent ,which is handled by StartCoyExePlan. This plan in turn posts ReadStatus event to itself repeatedly. This event is handled by ReadStatusPlan which update the synthesized belief (CompanyStatus.Bel) of its platoon Team.

Company Team also posts event UpdateCoyOtherBelief, repeatedly to update other belief of Company. For example the company has casualty belief of all three Platoon in CoyLevelPlatoonStatusInfo.Bel.

The UpdateCoyOtherBeliefPlan handles the event UpdateCoyOtherBelief & modifies the belief CoyLevelPlatoonCasStatusInfo.Bel containing the casualty information of its platoons. This Belief is used by company commander to keep track of its platoon's casualty. if the casualty value of the assault platoon increases beyond threshold level, the company take decision to send the other platoon (having high morale) for fire support Role.

6.5.2.2 Company team initiating platoon team formation (Team Platoon1,pletoon2,platon3,Coy HQr)

Design: Coy_level_platoon_team_formation

coy team initiating platoon team formation (Team platoon1,pletoon2,platon3,Coy HQr)

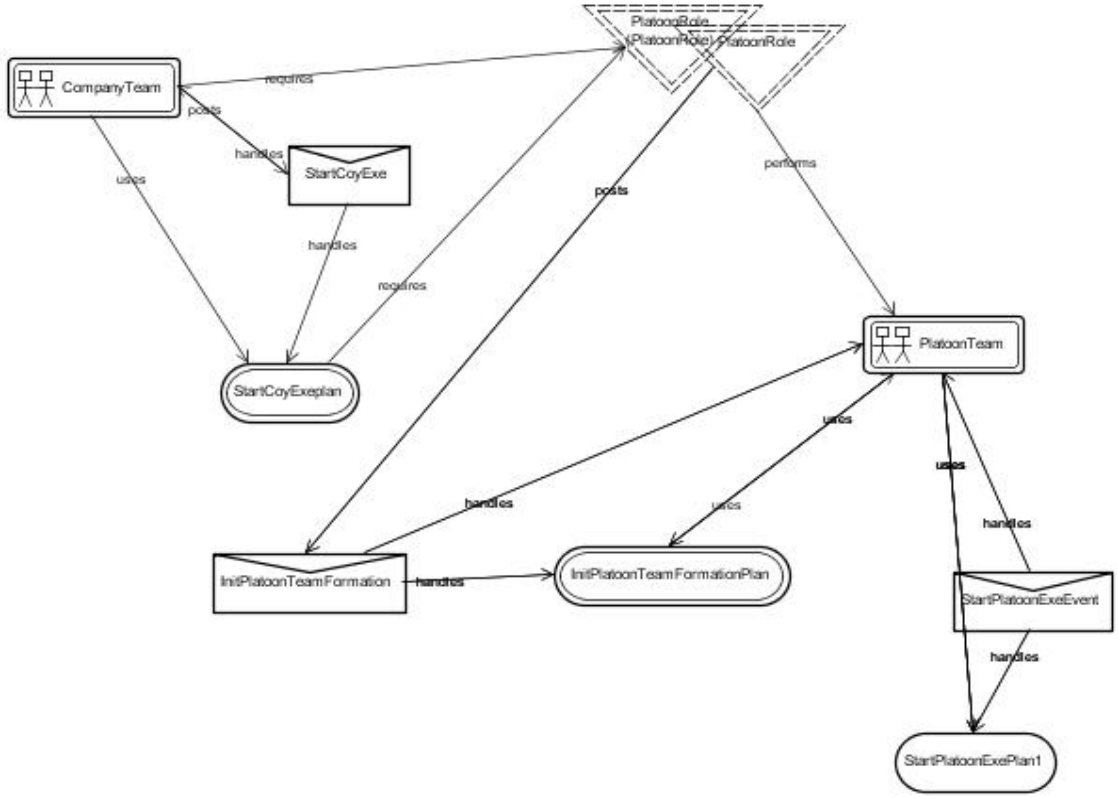


Figure 6.2 : Company team formation

Initially company team posts to itself startCoyExe event , which is handles by its StartCoyExeplan . This StartCoyExeplan plan (Figure 6.2) is responsible company team formation forming four team (platoon1,platoon2,platoon3,coy_Hqr platoon) based on role requirement . Plan StartCoyExeplan requires platoon Role to perform its tasks of team formation. Platoon Role is applicable from plan StartCoyExeplan . This plan in turn triggers event InitPlatoonTeamFormation through platoon role , which is handled by the plan InitPlatoonTeamFormationPlan of Platoon team, performing that role.

6.5.2.3 Task delegation down the Hierarchy

InitPlatoonTeamFormationPlan of Platoon team in turn triggers the event StartPlatoonExeEvent to platoon team. This event is responsible for Platoon level Team formation. The plan StartPlatoonExePlan of Platoon team handles this event. The plan StartPlatoonExePlan requires the four Section Role to perform its tasks. Section Role is applicable from the plan StartPlatoonExePlan of Platoon Team. The plan StartPlatoonExePlan of Platoon team sends the event startsection (through Section Role) to Section team performing Section Role. Event startsection of Section Team is handled by the StartPlan , which in turn sends the event start_move_event to all its Soldiers .

Design: MissionOrderDelegationDownHairarchy

Mission order delegation from Coy to Platoon ,Platoon to section ,Section to Soldiers.

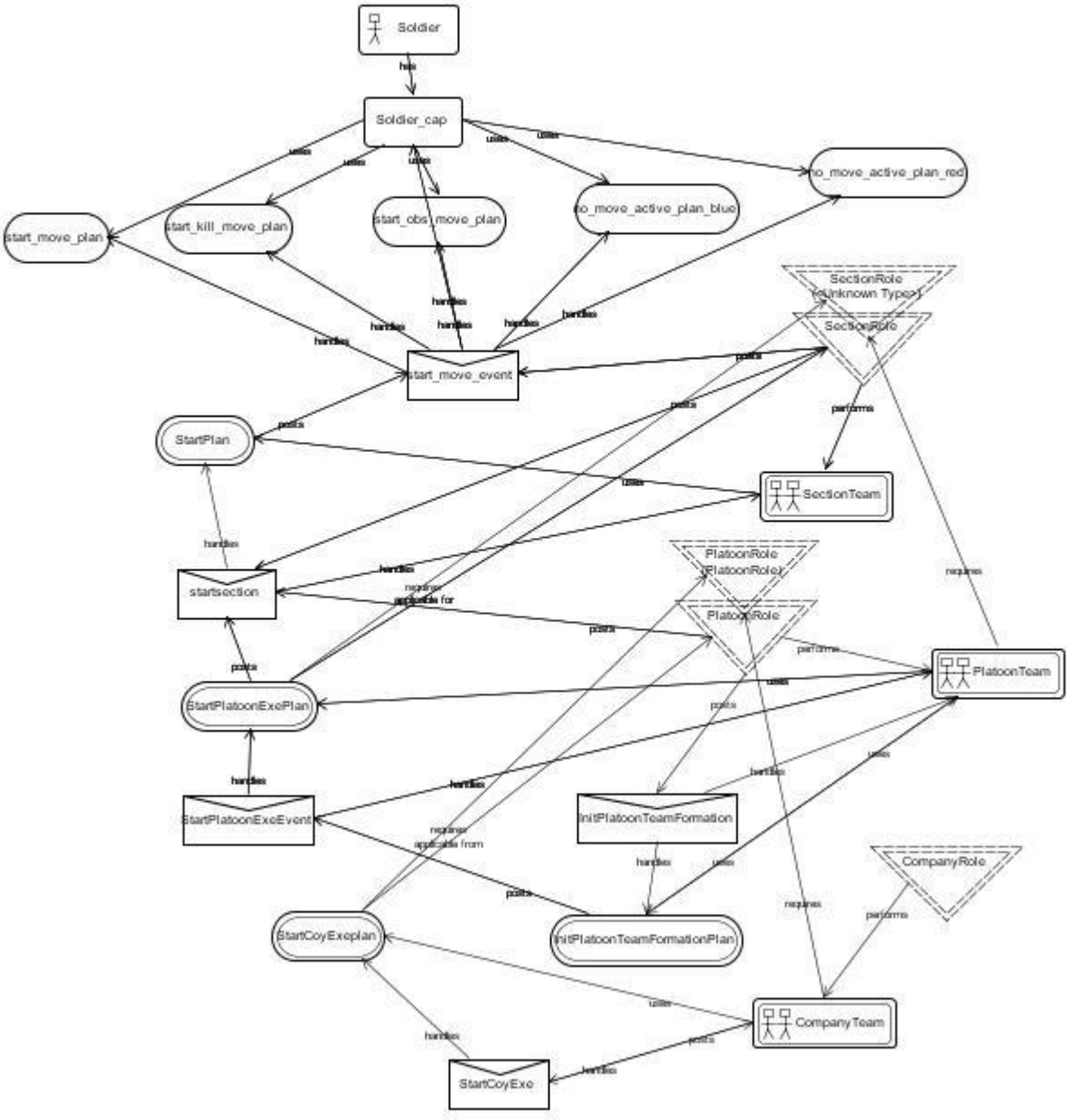


Figure 6.3: Task delegation down the Hierarchy

6.5.2.4 Belief propagation from Section Team to Platoon Team using Role structure

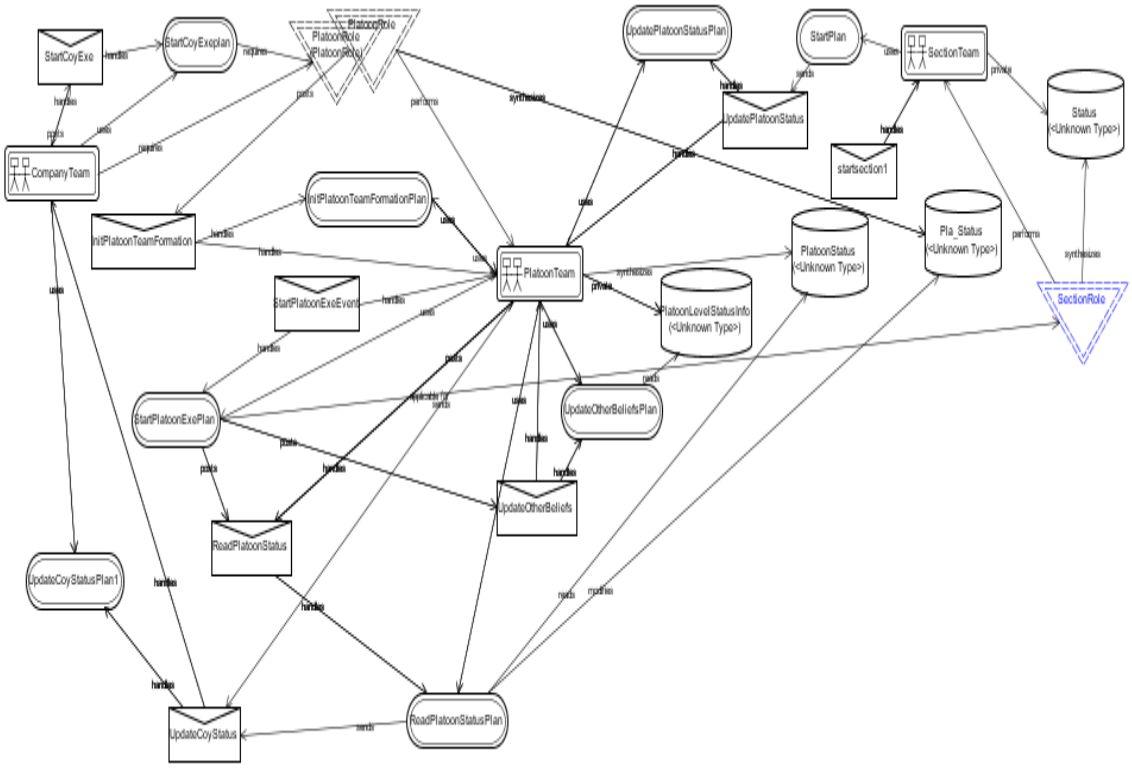


Figure 6.4 Belief propagation form section to Platoon

Section team has **private data Status section_status.** Section Team performing SectionRole (Figure 6.4) synthesizes teamdata Status section_status. The beliefset section_status **of** Belief type **Status** extends ClosedWorld, and has the **#propagates changes;**

declaration in its definition.

Platoon team class **synthesizes teamdata PlatoonStatus** in its class definition as **:**

#synthesizes teamdata PlatoonStatus;

platoon_status(sec1.section_status,sec2.section_status,sec3.section_status,plhqr.section_status);

teamdata PlatoonStatus is special team data **extending from Status,** containing the code connection method & team synthesizing method of its lower level units.

6.5.2.5 Arty fire start / stop request from platoon team on mine cross/ crossed event.

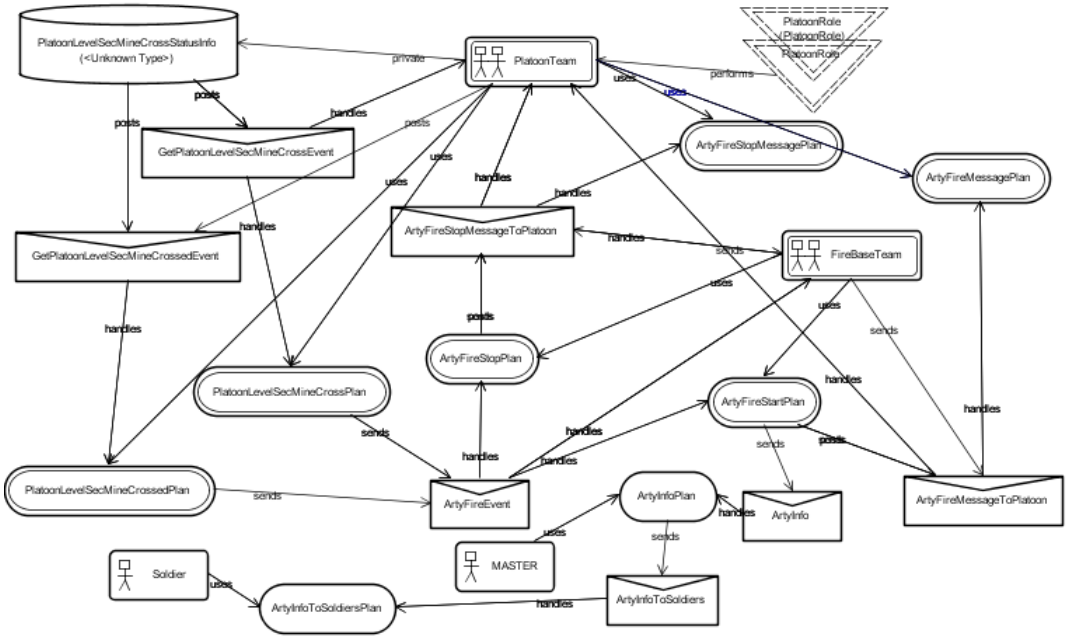


Figure 6.5 Arty fire request from platoon team on mine cross

6.5.2.5.1 Platoon triggering arty fire start message

PlatoonLevelSecMineCrossStatusInfo.Bel of Platoon team (generated from the synthesized belief PlatoonLevelStatusInfo.Bel) triggers event GetPlatoonLevelSecMineCrossEvent whenever the any of the platoon assault section encounters mine first time while movement to its objective. The event GetPlatoonLevelSecMineCrossEvent is handled by plan PlatoonLevelSecMineCrossPlan. This plan is only valid for assault section and further triggers event ArtyFireEvent to Firebase Team for support Arty fire.

Arty fire event ArtyFireEvent from platoon team is handled by Plan ArtyFireStartPlan of the Firebase team. This plan ArtyFireStartPlan sends the ArtyFireInfo event to MASTER agent to inform fire status. This arty fire status is further sent to all soldier agents by MASTER agent for arty fire information.

The soldiers engages the enemy if and only if the arty fire status is off. If on , the soldier fires if the position of soldier is not in arty fire range.

6.5.2.5.2 Platoon triggering arty fire stop message to Firebase team

Similarly PlatoonLevelSecMineCrossStatusInfo.Bel of Platoon team (generated from the synthesized belief PlatoonLevelStatusInfo.Bel) triggers event GetPlatoonLevelSecMineCrossedEvent whenever the platoon's assault section crosses mine completely. GetPlatoonLevelSecMineCrossedEvent is handled by plan PlatoonLevelSecMineCrossedPlan. This plan is only valid for assault section and further triggers event ArtyFireEvent to Firebase Team for stopping Arty fire.

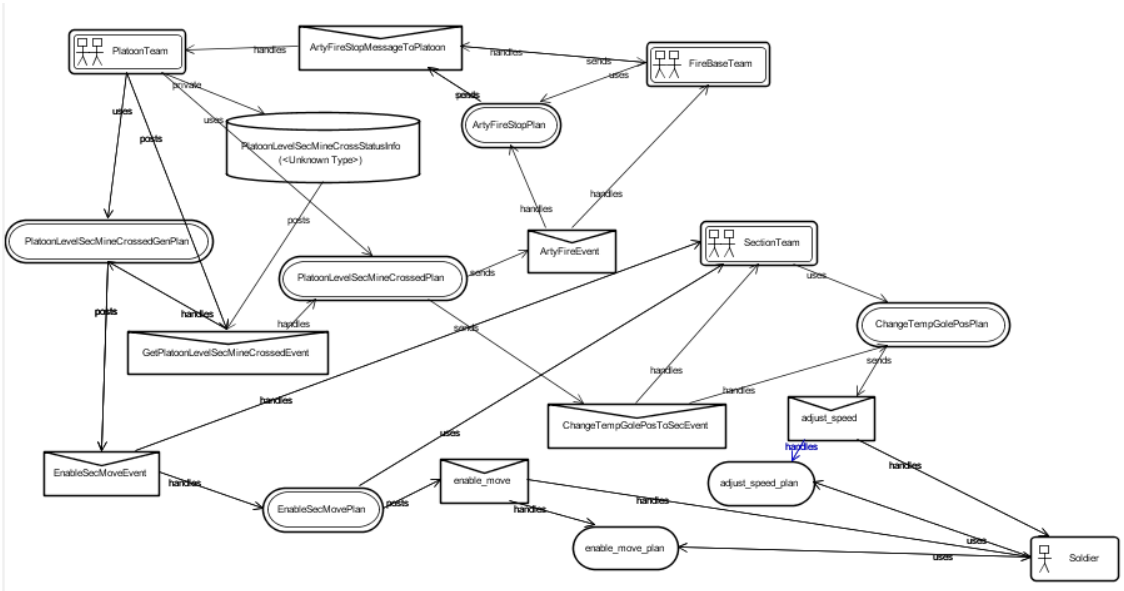


Figure 6.6 : Platoon triggering arty fire stop message to Firebase team

6.5.2.5.3 Platoon ordering to take rod position on mine cross event

The plan PlatoonLevelSecMineCrossPlan also sends the event ChangeTempGolePosToSecEvent to other section to change the goal (assault section position) temporarily so as to adjust in rod formation to avoid heavy casualty .

When any of the section adjusting to the rod position, encounters the mine field, this plan PlatoonLevelSecMineCrossPlan sends the original goal of platoon to its other section other then assault section.

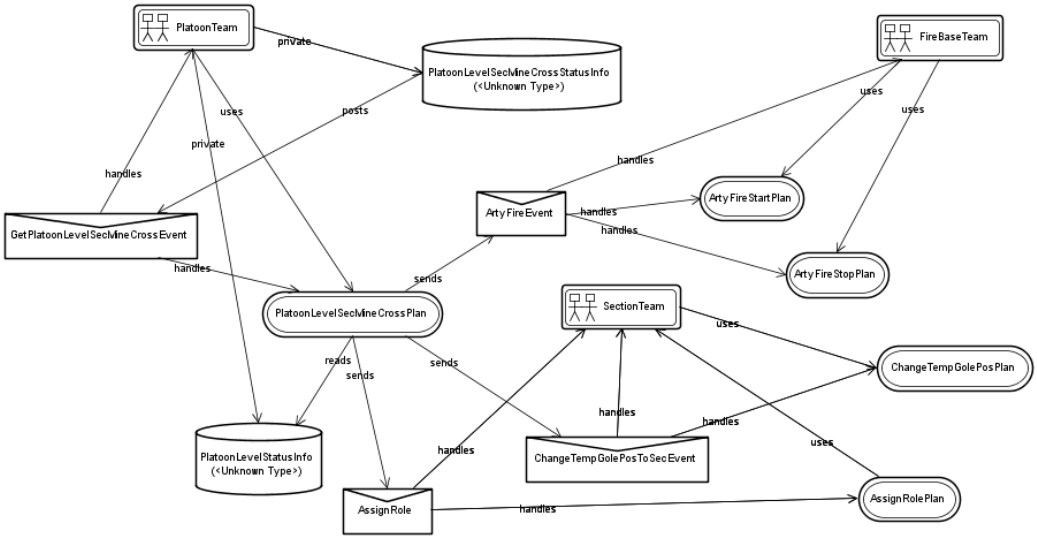


Figure 6.7 : Platoon ordering sections to take rod formation

6.5.2.6 Soldier Capability

Soldier capabilities including the behaviour of simple move, move across mine field, move across wire mesh,move across obastacle triggred by start_move event

Start_move event is handles by the appropriate plan satisfying the context according to soldier belief/data.

Soldier Capability

Soldier Capability consists of following events : ***start_move_event***.

Soldier Capability (Figure 5.16) consists of following plans :

start_move_plan,
start_kill_move_plan,
start_obs_move_plan,
no_move_active_plan_red,
no_move_active_plan_ blue ,
default_plan.

All above plans handles the start_move_event depending upon the active context condition satisfaction. For example if the soldier is in mine, (kill_zone=1) , then the context field of the start_kill_move_plan (containing kill_zone==1 ,move==1)as condition) will only be active , while all other plans will be invalid for this event.

Similarly if the soldier is not in mine, (kill_zone=0 & move=1) , then the context field of the start_move_plan (containing kill_zone==0 ,move==1 as condition) will only be active , while all other plans will be invalid for this event.

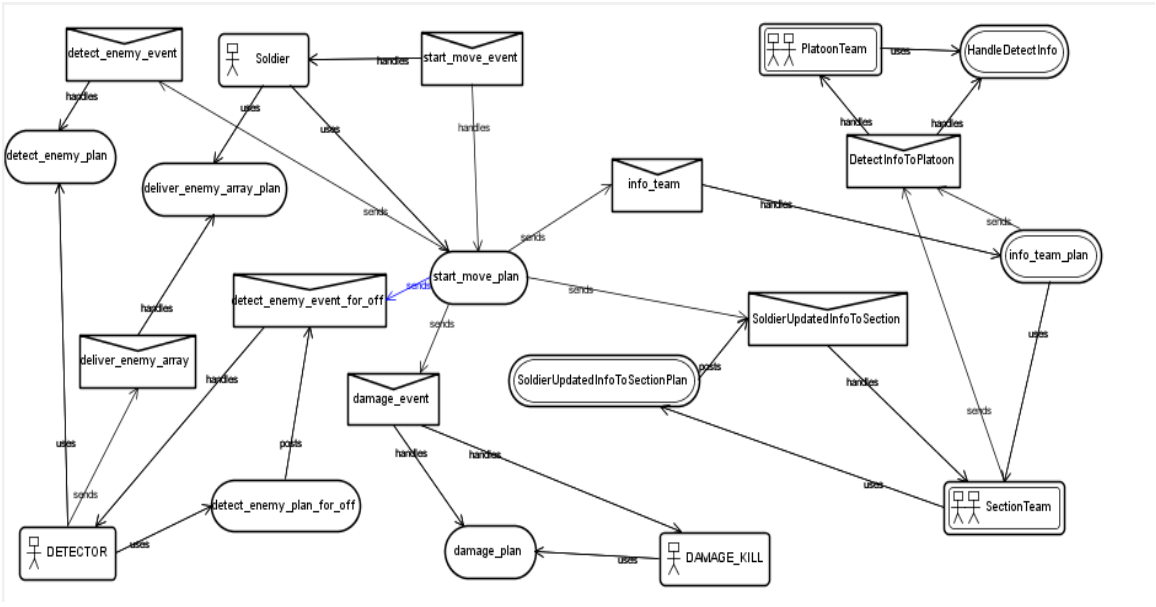


Figure 5.16 : Soldier Capability

Soldier agnet receves the event start_move_event from Section team..Soldier agent while executing plans (start_move_plan, start_kill_move_plan, start_obs_move_plan, no_move_active_plan_red, no_move_active_plan_ blue) , sends the start enemy detection event detect_enemy_event to DETECTOR Agent. The DETECTOR Agents has plan detect_enemy_plan to handle this event. From this plan the DETECTOR detects nearest enemy to soldier and sends back the detected result (enemy index, range etc) to soldier agents by sending the event deliver_enemy_array.The soldier agents has handler plan deliver_enemy_array_plan for the event deliver_enemy_array sent by DETECTOR.

Soldier agent also sends the events to DAMAGE_KILL agent to update the enemy kill status by sending the event damage_event. The DAMAGE_KILL agent has plan handler for this event as damage_plan. The DAMAGE_KILL agent updates the enemy kill status through this plan.

6.5.2.7 Section Capabilities

Platoon team requires Section Role to accomplish its mission tasks. Platoon team sends event startsection to all Section teams through plan StartPlatoonExePlan.

The description of SectionRole is as follows:

```
public role SectionRole extends Role
  {
    // declarations of events handled by the role performer
    // declarations of events posted by the role performer
    // declarations of teamdata synthesized from the role
    // performer
    // declarations of teamdata inherited by the role performer
    // declarations of role container methods and members
    // other Java methods and members


   #handles  event startsection1 wm;
   #synthesizes teamdata Status section_status;

  }
```

Event startsection passes to section through **SectionRole** (see Figure 6.9) is handles by plan **StartPlan** of Section team. The StartPlan plan of section is

responsible for further passing the section order to all its soldiers agents by sending the event **start_move_event**. The event start_move_event is handled by soldier agent by many plans such as start_move_plan, start_kill_move_plan, start_obs_move_plan, no_move_active_plan_red,no_move_active_plan_blue and default_plan.

The plan StartPlan also sends event Req_Killed_Soldier_Status to Soldier aganets to keep track of killed soldiers. Req_Killed_Soldier_Status_Plan plan of soldier agents handles the event **Req_Killed_Soldier_Status**.

All soldier agents of section sends the event **SoldierUpdatedInfoToSection** to its section at regular intervals. **SoldierUpdatedInfoToSectionPlan** of section handles the event SoldierUpdatedInfoToSection posted by soldier agent. This plan compiles the data / belief about morale, leadership, fatigue, suppression, casualty value, location of all live soldiers at regular step intervals defined by simulation.

The plan StartPlan of section team also modifies the section belief Status section_status compiles from all live soldiers agents.

Finally the StartPlan of each Section team sends event **UpdatePlatoonStatus** to its platoon to update the belief of section in platoon belief / data. The plan **UpdatePlatoonStatusPlan** of Platoon team handles this event **UpdatePlatoonStatus** and modifies the belief **PlatoonLevelStatusInfo.Bel** of Platoon Team. The belief **PlatoonLevelStatusInfo.Bel** contains the morale, leadership, fatigue, suppression, casualty value , location of all sections of platoon.

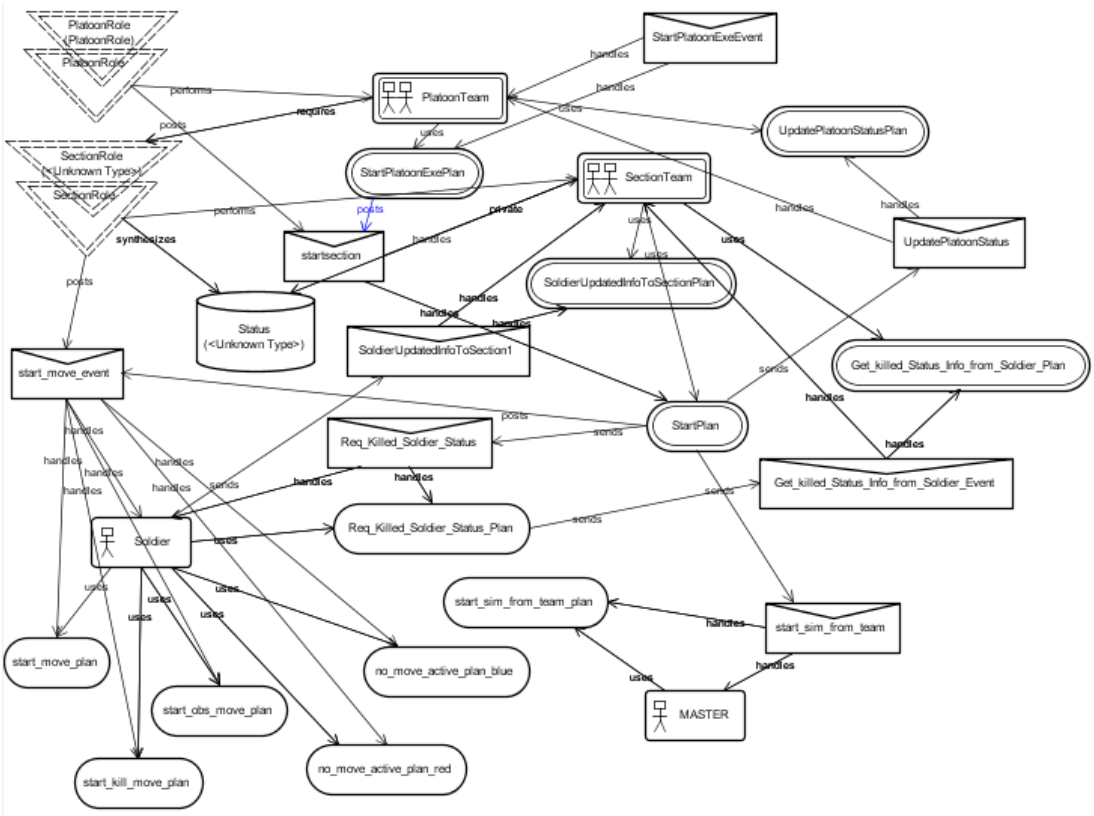


Figure 6.9 : Section Capability

Chapter 7. Implementation & Experimental Results

7.1 Environmental Setup

We have used the following configuration while finding the experimental results

Hardware Configuration

Processor	:	Intel Core 2 Duo
Processor Speed	:	2.20GHz
Main Storage	:	4GB RAM
Hard Disk Capacity	:	80GB
Monitor	:	Samsung 17”5’ Color

Software Configuration

Operating System	:	Windows 7
Front end	:	Java , JACK Tool

7.2 Implementation

Following functionalities have been implemented using JACK Tool:

- Team Formation At Company Level & Platoon Level
- Belief Propagation From Section To Platoon & From Platoon To Company
- Arty Fire & Platoon Formation Change (Rod) Event Trigger On Mine cross Belief Change
- Soldier And Other Agents Capabilities
- Complex Belief Generation From Synthesized Belief

- Company Level Decision Making Based On Company's Belief (Platoon Casualty Belief) Updation

7.2.1 TEAM FROMATION AT COMPANY LEVEL & PLATOON LEVEL

The Team Framework (Teams and Roles)

A structural relationship between teams is catered for via the *role* concept. A role defines the means of interacting between a containing team (a *role tenderer*) and a contained team (a *role performer* or *role filler*). The role defines which goals the role tenderer may request the role performer to achieve, and it also defines the counter-goals that the role performer may require from the role tenderer.

The team-role structure is defined by statements specifying which roles a team can perform, and which roles must be performed by sub-teams. These declarations are specified in the team's type definitions, where the containing team requires certain roles to be filled, and the contained team must be able to perform certain roles.

A team can perform roles for a containing team and can also contain sub-teams which perform roles on its behalf. The sub-teams can in turn contain sub-teams which can perform roles on their behalf etc.

The following code segments illustrate how these team and role definitions may look.

```
public team CompanyTeam extends Team
  {
       // team declarations and definitions
        #performs role  CompanyRole;

         #requires role PlatoonRole pla1(1,1);
         #requires role PlatoonRole pla2(1,1);
         #requires role PlatoonRole pla3(1,1);
         #requires role PlatoonRole coy_hqr(1,1);
        ;
}

public role PlatoonRole extends Role
{
```

```
    // declarations of events handled by the role performer

    #handles event InitPlatoonTeamFormation wmp;

    // declarations of events posted by the role performer

    // declarations of teamdata synthesized from the role  performer
        #synthesizes teamdata Status pla_status;
}
```

The team-role declarations determine which team-team structures can be built at run time.

Role Definition

The role definition does not contain implementation – only a description of the facilities that the two participants in the role relationship must provide. A role definition has two parts: first, a downwards interface that declares the events an entity must handle to take on the role, and second, an upwards interface that declares the events the team entity requiring the role needs to handle.

A role definition, such as the PlatoonRole definition shown above, results in two Java classes being generated by the compiler. One is named by the given RoleType type. The second generated class is a specialised 'container' for instances of a RoleType called RoleTypeContainer. The latter is referred to as a *role container*, as its purpose is to contain RoleType objects. In the case of the PlatoonRole definition shown above, the compiler would automatically generate the two Java classes PlatoonRole and PlatoonRoleContainer.

When the declaration is made that a team requires a given role, the result is a role-defined container to be filled by sub-teams. The #requires role RoleType reference(min,max) statement adds a field to the team class of name reference and type RoleTypeContainer. The #requires role declaration allows the specification of bounds for the container, which results in team formation constraints.

The arguments min and max in the #requires declaration specify the lower and upper bounds for the number of performers in order for the team to be

considered formed. A zero upper bound dictates an unlimited upper bound. Note that these bounds are not enforced by the infrastructure in order to allow dynamic attachment/detachment of sub-teams. In practice, a role container can contain an unspecified number of role objects.

In the team definition illustrated above, the declarations state that a Company team requires four sub-teams able to perform the PlatoonRole role. Furthermore, the Company team is declared to be a performer of the CompanyRole role, which would be a role required by some other team type.

It should be noted that the declarations above define how an actual team structure may look, but they do not identify the actual team instances, or what the team types are in the actual team structure.

Team Formation

The overall lifetime of a team has two phases. The first phase is for setting up an initial *role obligation structure*. The second phase constitutes the actual operation of the team.

At run time, teams undergo a team formation phase intended to identify the particular sub-team instances that take on roles in a team. This first phase is initiated via a TeamFormationEvent that is posted by the kernel when each team is constructed. By default, the TeamFormationEvent is handled by a plan that identifies the role fillers according to an initialisation file in JACOB format. The following is an example of an initialisation file (**scenario.def**):

```
<Team :name "COY_R"
    :roles (
        <Role :type "platoonteam.PlatoonRole" :name "pla1"
           :fillers (
                    <Team :name "PLATOONR1@%portal" >
                     <Team :name "PLATOONR2@%portal" >
                     <Team :name "PLATOONR3@%portal" >

                )
        >
        <Role :type "platoonteam.PlatoonRole" :name "pla2"
```

```
            :fillers (

                        <Team :name "PLATOONR1@%portal" >
                        <Team :name "PLATOONR2@%portal" >
                        <Team :name "PLATOONR3@%portal" >

                    )
            >
            <Role :type "platoonteam.PlatoonRole" :name "pla3"
                :fillers (

                        <Team :name "PLATOONR1@%portal" >
                        <Team :name "PLATOONR2@%portal" >
                        <Team :name "PLATOONR3@%portal" >

                    )
            >

  <Role :type "platoonteam.PlatoonRole" :name "coy_hqr"
                :fillers (

                        <Team :name "PLATOONR4@%portal" >

                    )
            >

        )
  >
```

Note : The Role tendering company Team (COY_R) requires four Roles as PlatoonRole referenced by pla1,pla2 ,pl3 & coy_hqr.

The Role performer platoon team PLATOONR1, PLATOONR2 & PLATOONR3 & PLATOONR4 are acting as role filler for Roles (PlatoonRole) required by company team refrenced as pla1,pla2,pla3 & coy_hqr.

The Teams framework is flexible at this point, but it includes the notion of a fully formed team as a team for which all necessary role performers have been identified.

The framework will allow a team instance to complete its team formation phase without necessarily satisfying all the role filling constraints. However, the team will only be considered formed when its role containment constraints are all filled. This is a state that a program may query.

At this stage the initial role obligation structure has been constructed. It is possible to dynamically modify this structure during program execution.

Task Teams

Task teams are dynamically formed sub-groups within a team, created to perform a team task. When chosen to handle an event, the initial step of a teamplan is to establish the task team, by selecting which role performers to use from within the team for the various roles needed within the task/plan.

Task teams are not defined separately, but are contained within the teamplans defining the team tasks. A teamplan uses #requires and/or #uses declarations to declare the roles needed for the task team. The teamplan may also include an establish() reasoning method that defines how the task team is to be established for the task. This is illustrated in the code segment below:

```
import aos.extension.parallel.ParallelMonitor;
teamplan StartCoyExeplan extends TeamPlan
{
   #handles event StartCoyExe pfv1;
   #posts event ReadStatus1 rfv1;
   #uses role PlatoonRole pla1 as PLA11;
   #uses role PlatoonRole pla2 as PLA22;
   #uses role PlatoonRole pla3 as PLA33;
#uses role PlatoonRole coy_hqr as COY_HQR;
#reasoning method
      establish()
      {
      // code to establish the task team for the task
      Vector busy = new Vector();
      PLA11 = (PlatoonRole) pickRole( busy,pla1);
      PLA11 != null;
      PLA22 = (PlatoonRole) pickRole( busy, pla2);
      PLA22 != null;
      PLA33 = (PlatoonRole) pickRole( busy, pla3);
      PLA33 != null;
      COY_HQR = (PlatoonRole) pickRole( busy, coy_hqr );
      COY_HQR != null;
      System.out.println("Team established for Company ");
      }
```

```
   Role pickRole(Vector busy,RoleContainer rc)
   {
      for (Enumeration e = rc.tags(); e.hasMoreElements(); ) {
         Role r = rc.find( (String) e.nextElement() );
         if ( !busy.contains( r.actor ) ) {
            busy.add( r.actor );
            return r;
         }
      }
      System.out.println(" retrun null");
     return null;
   }
       body()
      {          // body of the plan to perform the task
      }
}
```

The establish step of a teamplan is a proper plan step, and may involve any amount of reasoning by the team entity, as well as negotiations with the candidate sub-teams. The outcome is either a complete assignment of sub-teams to the roles required by the teamplan, or a plan failure allowing the team to choose an alternative plan for handling the same event. If there is a fail() reasoning method associated with the plan, it does not get executed if the establish() method fails.

There is a default establish() method which fills the required roles uniquely at random, if possible. However, the default establish method only assigns the #requires roles and not the #uses roles.

Behaviour

The concepts of teams requiring roles and teams performing roles provide a framework where group behaviours and individual behaviours can be clearly separated. Group behaviour is specified in terms of the roles that are required to achieve the desired behaviour. This behaviour is specified independently of the actual teams performing the roles. However, the team has access to its

sub-teams through the role container, so it is able to perform reasoning based on the actual team membership when necessary.

The team is a separate entity and has its own teamplans for the specification of team behaviour. Within these teamplans, the @teamAchieve statement can be used to help coordinate the behaviour of the sub-teams.

@teamAchieve

The @teamAchieve statement is used to activate a sub-team by sending an event to the sub-team. The team that sent the @teamAchieve then waits until the event has been processed by the sub-team.

In combination with the JACK @parallel statement, a wide range of team behaviours can be implemented.

Belief Propagation

In addition to communicating via the normal message/event passing in agent-oriented programming, Teams also provides a capability for the propagation of team beliefs. This propagation can be both from team to sub-team and from sub-team to team. In the latter case, the capability is provided within Teams to combine the propagated sub-team beliefs within the team. The use of Team beliefs in conjunction with the Team coordination statements enables sophisticated team behaviours to be implemented.

Similarly , the Platoon acts as Role tenderer , as it also requires four Roles to be performed by four different teams.

```
public team PlatoonTeam extends Team
  {
       // team declarations and definitions

        // role performer declaration

            #performs role PlatoonRole;
```

```
            // role requirer declaration

              #requires role SectionRole1 sec1(3,3);
              #requires role SectionRole2 sec2(3,3);
              #requires role SectionRole3 sec3(3,3);
              #requires role SectionRole4 plhqr(1,1);
             ;
}
```

In the team definition illustrated above, the declarations state that a Platoon team requires one sub-teams able to perform the SectionRole1 role, another sub-team able to perform the SectionRole2, one sub-teams able to perform the SectionRole3 role and one sub team to perform SectionRole4 role (HQtr Role). Furthermore, the Platoon team is declared to be a performer of the PlatoonRole Role, which would be a role required by Company Team .

Note : The Role tendering platoon Team (PLATOON_1) requires four Roles as SectionRole1, SectionRole2, SectionRole3, SectionRole4 referenced by sec1,sec2 ,sec3 & plhqr.

In the scenario..def , the structure of the Platoon Team PLATOON_1 will be as follows:

```
<Team :name "PLATOONR1"
      :roles (
            <Role :type "sectionteam.SectionRole1" :name "sec1"
               :fillers (
                    <Team :name "SEC_1@%portal" >
                    <Team :name "SEC_2@%portal" >
                    <Team :name "SEC_3@%portal" >
                   )
            >
            <Role :type "sectionteam.SectionRole2" :name "sec2"
               :fillers (
                    <Team :name "SEC_1@%portal" >
                    <Team :name "SEC_2@%portal" >
```

```
                <Team :name "SEC_3@%portal" >
                )
            >
            <Role :type "sectionteam.SectionRole3" :name "sec3"
                :fillers (
                    <Team :name "SEC_1@%portal" >
                    <Team :name "SEC_2@%portal" >
                    <Team :name "SEC_3@%portal" >
                )
            >
<Role :type "sectionteam.SectionRole4" :name "plhqr"
                :fillers (

                    <Team :name "SEC_10@%portal" >
                    )
            >
        )
  >
```

Note: The Role performer platoon team SEC_1, SEC_2 & SEC_3 & SEC_10 are acting as role filler for Roles required by platoon team (PLATOON_1) namely sec1,sec2, sec3 & plhqr.

7.2.2 BELIEF PROPOGATION FROM SECTION TO PLATOON & FROM PLATOON TO COMPANY

Synthesizing Belief Connection Definition

A synthesizing team belief connection maps sub-teams' beliefs into corresponding beliefs at the containing team level. This is achieved by propagating information from the sub-team beliefsets to the containing team(s). In order to create a synthesizing team belief connection, appropriate declarations must be included in the

- role that provides the sub-team/team linkage
- the sub-teams that are the source for the connection
- the team that is the target for the connection.

In addition

- a teamdata definition must be provided for the target team
- the source beliefsets must include #propagate changes statements.

Role Declarations

To associate a synthesizing belief connection with a role, the following statement form is used:

#synthesizes teamdata *stype sref*;

stype and sref identify a **source** beliefset that will be involved in a synthesizing belief connection – the target for the connection is **not** specified. Multiple declarations are allowed within a role definition.

Recall that a role defines a team/sub-team interface. Within a role type definition, the #synthesizes teamdata declaration declares that any sub-team that performs this role must provide a data item named sref of type stype. Likewise, any team that requires this role should have a target data declaration that involves this particular data item or it will be unable to receive the propagated beliefs.

Source Declarations

1. A sub-team becomes a source (SectionTeam) in a synthesizing belief connection by filling a role (SectionRole) that contains a #synthesizes teamdata declaration.

```
public role SectionRole extends Role
  {
    // declarations of events handled by the role performer
    #handles  event startsection1 wm;
    // declarations of teamdata synthesized from the role performer
```

```
    #synthesizes teamdata Status section_status;
  }
```

2. A sub-team becomes a source (SectionTeam) must declare beliefsets **Status** as pvt member :

```
public team SectionTeam extends Team
  {
    // team declarations and definitions
    #performs role SectionRole1;
   #private data Status section_status();
 }
```

3. The source beliefsets **Status** of Section Team must include #propagate changes statements

```
public beliefset  Status extends  ClosedWorld
{
      #value field double morale_value;
      #value field double leadership_value;
      #value field double suppression_value;
      #value field double fatigue_value;
      #value field double casuality_value;
      #value field double minecross_value;
       #value field double x;
      #value field double y;
      // get complete information
      #linear query get(logical double morale_value, logical double leadership_value, logical double suppression_value,  logical double fatigue_value, logical double casuality_value, logical double minecross_value, logical double x,logical double y);
       #propagates changes;
```

```
}
```

Thus the sub-team must include an appropriate #performs role declaration and fill the role in the containing team's Role Obligation Structure. Also a data item with the type and the reference specified within the role must be defined either directly within the sub-team definition, or indirectly through the sub-team's capability structure. The data item can be defined either through a #private data declaration, a #exports data declaration in a capability, a #synthesizes teamdata or through a #inherits teamdata declaration. The latter two cases require that the sub-team is the target for another belief connection.

Target Declarations

The target beliefsets **PlatoonStatus** of Platoon Team is derived from Status as follows:

```
public  teamdata PlatoonStatus extends Status {

    Hashtable Status_s = new Hashtable();

    Status status(String team)
    {
      return (Status) Status_s.get(team);
    }

    #connection method(boolean added, String team)
    {
      if (added) {
        if (Status_s.get(team) == null ) {
            Status status = new Status();
            status.attach(getHandler());
            Status_s.put(team, status);
        }
      } else {
        Status_s.remove(team);
      }
    }
  #synthesis method
            (String team,
             boolean asserted,
             BeliefState tv,
             Status__Tuple is,
             Status__Tuple was,
             Status__Tuple lost)
  {
    Status status = (Status) Status_s.get(team);
    if (asserted)
```

```
status.add(is.morale_value,is.leadership_value,is.suppression_value,is.fatigue_value,is.casu
ality_value,is.minecross_value,is.x,is.y, tv);
      else

status.remove(is.morale_value,is.leadership_value,is.suppression_value,is.fatigue_value,is.c
asuality_value,is.minecross_value,is.x,is.y, tv);


      double morale = 0;
      double leadership  = 0;
      double suppression = 0;
      double fatigue = 0;
      double casuality = 0;
      double minecross_value=0;
      double sum_x = 0;
      double sum_y = 0;
      int n = Status_s.size();

      for (Enumeration e = Status_s.elements();
          e.hasMoreElements(); ) {
        Status status = (Status) e.nextElement();

            logical double morale1;
        logical double leadership1;
            logical double suppression1;
        logical double fatigue1;
        logical double casuality1;
        logical double minecross_value1;
        logical double x;
        logical double y;

        status.get(morale1, leadership1, suppression1, fatigue1,
casuality1,minecross_value1,x,y);

        morale += morale1.getValue();
        leadership += leadership1.getValue();
        suppression += suppression1.getValue();
        fatigue += fatigue1.getValue();
        casuality  += casuality1.getValue();
        minecross_value += minecross_value1.getValue();

        sum_x += x.getValue();
        sum_y += y.getValue();

      }

         morale /=n;
          leadership /=n;
          suppression /=n;
          fatigue /=n;
          casuality /=n;
        minecross_value/=n;

        sum_x /= n ;
        sum_y /= n ;
      add(morale,leadership,suppression,fatigue,casuality,minecross_value,sum_x,sum_y);
    }
```

}

Note : beliefsets **PlatoonStatus** has methods connection method & synthesis methods.

- a #connection method which defines the behaviour when teams are added to or removed from the connection, and
- a #synthesis method which defines the computation to be performed on receipt of a propagated belief. This method is invoked **regardless** of whether the connection is synthesizing or inheriting.

A team becomes a target in a synthesizing belief connection by requiring a role that contains a #synthesizes teamdata declaration. Thus the team must include an appropriate #requires role declaration and a #synthesizes teamdata declaration that binds the data item specified in the role with the role container that contains the sub-teams that fill the role.

A #synthesizes teamdata declaration has the following form in the team definition:

#synthesizes teamdata ttype tref(rcref1.sref1,rcref2.sref2,...);

where

ttype is the type of the target teamdata
tref is the name of the target teamdata reference
rcrefi is the name of the ith role container reference
srefi is the name of the ith source data item reference which is to be synthesized.

As indicated above, teamdata can be synthesized from beliefs specified in more than one role. In this case, multiple #requires role statements will be required in the team and the types of the source beliefs must be the same. Note that ttype refers to the type of the target teamdata, not the type of the source data.

Recall that the teamdata type is typically achieved by extending a beliefset type. Depending on the application, that beliefset type may be the same as the source data type or it may be different.

```
public team PlatoonTeam extends Team

 {

 // team declarations and definitions

#performs role PlatoonRole;

#synthesizes teamdata PlatoonStatus
platoon_status(sec1.section_status,sec2.section_status,sec3.section_status,plhqr.section_st
atus);

}
```

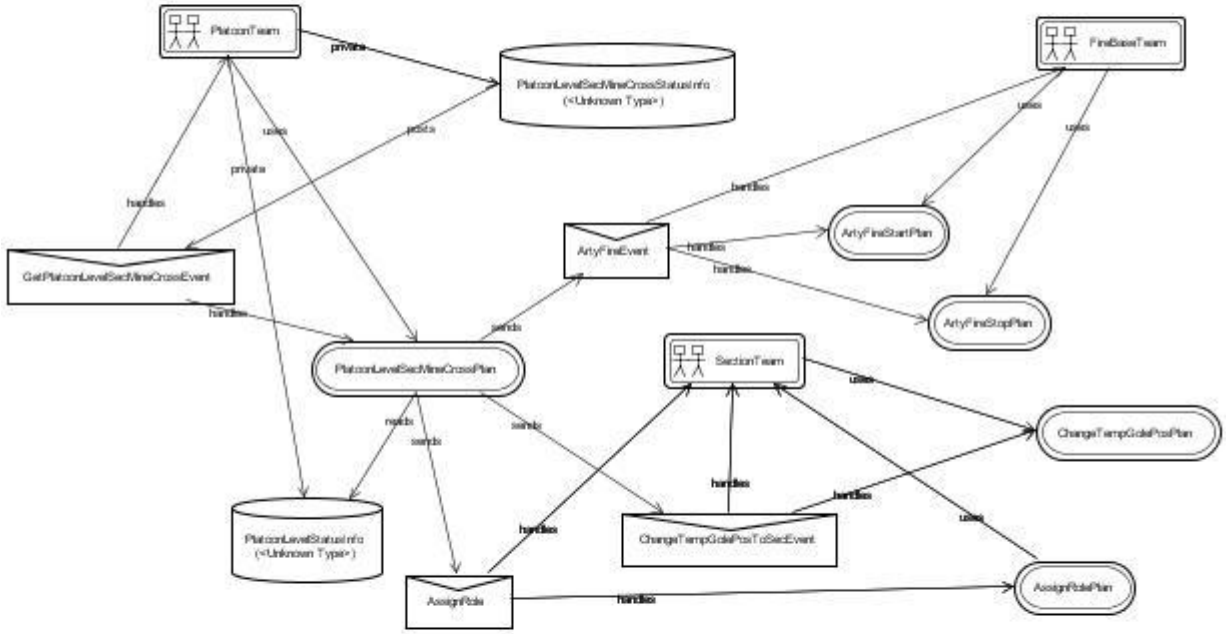
The above declaration results in the creation of a teamdata instance (platoon_status). The intention is that the data to be contained in this instance will be provided solely from the data sources for the connection – hence there is no mechanism to populate the instance at construction time. This teamdata instance is then accessible to the target team and through the #uses data declaration, to the target team's capabilities and plans as though it had been declared as #private data. In particular, a teamdata instance can be used as a source belief for another belief connection.

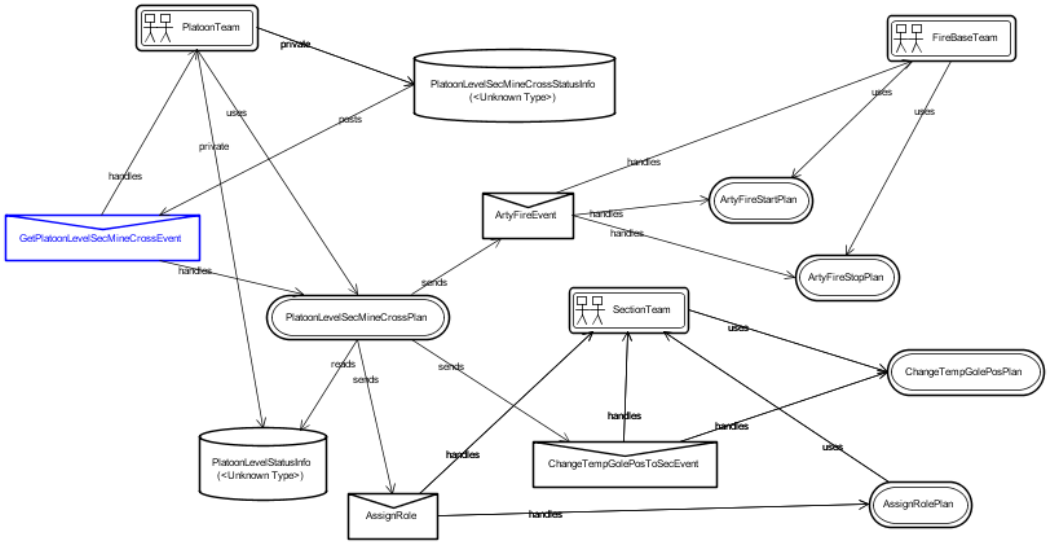
The #synthesizes teamdata statement results in code that ensures that when role fillers are added to or removed from any of the indicated role containers the corresponding beliefset change propagation path is added or removed. The actual synthesizing computation is defined separately (via the #synthesis method of the teamdata definition). Although a connection is defined in terms of role filling, it is maintained on a sub-team basis. Thus if a connection involves multiple roles and one sub-team fills more than one of the roles, a change to that sub-team's beliefset is propagated only once to the teamdata, and not once for each role container that contains the sub-team.

7.2.3 ARTY FIRE & PALTOON FORMATION CHANGE (ROD) EVENT TRIGER ON MINECROSS BELIEF CHANGE

Design: Platoon_level_section_mine_cross

platoon trigerring arty fire start/stop message & ordering sections poition to rod position





ARTY Fire support from FireBase Team

When any of the assault section of the platoon encounters the minefield , it immediately update the mines cross belief of the platoon belief (PlatoonLevelSecMineCrossStatusInfo.bel) , which posts an events PlatoonMineCrossedEvent to itself & this event is handled by plan PlatoonLevelSecMineCrossPlan.

The PlatoonLevelSecMineCrossStatusInfo.bel belief is as follows:

```
package platoonteam;
public beliefset  PlatoonLevelSecMineCrossStatusInfo extends  ClosedWorld
 {
      // field of the PlatoonLevelSecMineCrossStatusInfo Belief

       #value field double sec1_pre_minecross_value;
       #value field double sec1_minecross_value;
       #value field double sec2_pre_minecross_value;
       #value field double sec2_minecross_value;
       #value field double sec3_pre_minecross_value;
       #value field double sec3_minecross_value;

      // get complete information of a named Team
       #linear query  get(logical double sec1_pre_minecross_value, logical double
sec1_minecross_value,logical double sec2_pre_minecross_value, logical double
sec2_minecross_value,logical double sec3_pre_minecross_value, logical double
sec3_minecross_value );

// when any of the assault section first encountered minefields
```

```
      #posts event GetPlatoonLevelSecMineCrossEvent PlSecMineCross;
// when any of the assault section first fully crosses minefields
       #posts event GetPlatoonLevelSecMineCrossedEvent PlSecMineCrossed;

     // when whole platoon crossed the minefield
      #posts event PlatoonMineCrossedEvent pmce;

 // first section going into mine field

      public void newfact(Tuple t,BeliefState is,BeliefState was)
      {
        // Note that PlatoonLevelSecMineCrossStatusInfo__Tuple contains two underscores

       PlatoonLevelSecMineCrossStatusInfo__Tuple pt =
(PlatoonLevelSecMineCrossStatusInfo__Tuple) t;

 if (pt.sec1_pre_minecross_value==0 && pt.sec1_minecross_value >0 &&
pt.sec2_pre_minecross_value==0 && pt.sec2_minecross_value ==0 &&
pt.sec3_pre_minecross_value==0 && pt.sec3_minecross_value == 0 )
        {
          // code to post the trigger event. This code will
          // be executed whenever a Team playing sec1 crosses enemy mines first time is
          // added in  belief set. For example:

         // for arty fire start & platoon changing its formation to Rod
          postEvent(PlSecMineCross.MineCross(pt.sec1_minecross_value,0,1));


        }

 // second section going into mine field
;
// third section going into mine field
;
// first section first coming out from mine field

  if (pt.sec1_pre_minecross_value>0 && pt.sec1_minecross_value ==0 &&
pt.sec2_pre_minecross_value>0 && pt.sec2_minecross_value >0 &&
pt.sec3_pre_minecross_value>0 && pt.sec3_minecross_value > 0 )
          {
          // code to post the trigger event.

       postEvent(PlSecMineCrossed.MineCrossed(pt.sec1_minecross_value,0,0));

      }
// second section first coming out from mine field
;
// third section first coming out from mine field
;
}
```

Note : when the belief contains the condition when first section crosses the mine field first , it triggers the event GetPlatoonLevelSecMineCrossEvent handled by PlatoonLevelSecMineCrossPlan.

This plan PlatoonLevelSecMineCrossPlan intern sends an event ArtyFireEvent to its Firebase Team for arty fire support. Platoon team also passes target coordinates in event ArtyFireEvent. The event ArtyFireEvent handled by Plan ArtyFireStartPlan of the Firebase Team.

The ArtyFireEvent is defined as follows:

```
event ArtyFireEvent extends MessageEvent
{
  public String s1;
  public double Val;
  public double x;
  public double y;

  #posted as
  MsgToArty(String p, double Val1,  double  x1,  double  y1)
    {
         s1 = p;
         Val=Val1;
           x=x1;
           y=y1;
    }
}
```

The Plan ArtyFireStartPlan of the FireBase Team is as follows:

```
package firebaseteam;
import platoonteam.ArtyFireEvent;

teamplan ArtyFireStartPlan extends TeamPlan {
     #handles event ArtyFireEvent af;
     #uses interface FireBaseTeam fteam;
     #sends event ArtyFireMessageToPlatoon aftp;

static boolean relevant(ArtyFireEvent af){
return( af.Val==1);
}

context()
{
( fteam.mode.equals("red") );
}
    body()
    {

      System.out.println(" Fire Base Team  " + fteam.name1+ " receiving arty fire request from
its Platoon  "+ af.s1);

      System.out.println(  fteam.name1+ " Starting  Arty Fire at enemy position " +af.x + " " +
af.y );

 // send message to platoon for acknowledging arty fire
```

```
     @send(af.s1,aftp.ArtyFireMsg(" Fire msg to Platoon Team"));

     fteam.start_firing=1;
     fteam.x=(int)af.x ;
     fteam.y=(int)af.y ;

   while(fteam.start_firing==1)
   {

System.out.println(  fteam.name1+ " Firing  continuously at enemy position ..");

    @waitFor(elapsed(3.0));
   }
   }
}
```

Formation Change (Rod) Event Trigger On Mine cross Belief Update / Change

The belief triggered event GetPlatoonLevelSecMineCrossEvent ,handled by PlatoonLevelSecMineCrossPlan sends the event changeTempGolePosToSecEvent to its other sections who have not so far encountered the minefields.

The plan **PlatoonLevelSecMineCrossPlan** is follows:

package platoonteam;

```
teamplan PlatoonLevelSecMineCrossPlan extends TeamPlan {

   #handles event GetPlatoonLevelSecMineCrossEvent pc1;
   #uses interface PlatoonTeam  pteam;
   #sends event ArtyFireEvent af;
   #uses data PlatoonLevelStatusInfo plstatusinfo;
   #sends event AssignRole ar;
   #sends event ChangeTempGolePosToSecEvent ctgps;
   #reads data PlatoonLevelSecMineCrossStatusInfo PlSecMineCross;

// logical variables to retrieve plstatusinfo belief fields

       String TeamName;
       logical double  morale_value1;
       logical double  leadership_value1;
       logical double  suppression_value1;
       logical double  fatigue_value1;
       logical double  casuality_value1;
       logical double  minecross_value1;
       logical double x1;
       logical double y1;
       double x;
       double y;
```

```
context()
{
(pteam.FirstFire==0);
}
   body()
   {
     System.out.println(" Platoon Team " + pteam.name()+ " gets msg from mine_cross belief
set trigger.." + " mine crossing.. " + pc1.MineCrossValue + " Fire Val " + pc1.Val );

    // platoon team sending fire event (containing target coordinates) to firebase to start fire

@send(pteam.FireBaseTeam,af.MsgToArty(pteam.name1,pc1.Val,pteam.goal_x,pteam.goal_
y) );

     // invalidate  this plan context to execute in next iteration
     pteam.FirstFire=1;


// Update Platoon's  AssultRoleIndex section

pteam.AssultRoleIndex=pc1.crossed_sec_index;

  if(pc1.crossed_sec_index==0)
{

  // first section as assault section

@send(pteam.RolePerformer[0],ar.AssignRole("Assault"));

  // retreive the belief data of the section 1 team


plstatusinfo.get(pteam.RolePerformer[0],morale_value1,leadership_value1,suppression_valu
e1,fatigue_value1,casuality_value1,minecross_value1,x1,y1);

   // get the position of assault section  1

      x=x1.getValue();
      y=y1.getValue();
  @send(pteam.RolePerformer[1],ctgps.ChangeTempGolePosToSec("Change Sec 2
Gole",(int)x,(int)y,4));
  @send(pteam.RolePerformer[2],ctgps.ChangeTempGolePosToSec("Change Sec 3
Gole",(int)x,(int)y,3));

 System.out.println(" Platoon Team " + pteam.name() +  " changing goal of  two sections ,
sec2 , sec3  to sec1 position ");

// constantly check weather any one of  sec2 & sec 3 has encountered mines
// if yes then change their goals to platoon main goal.
while(1==1)
{
  // mine cross belief parameters

        logical double sec1_pre_minecross_value;
        logical double sec1_minecross_value;
        logical double sec2_pre_minecross_value;
        logical double sec2_minecross_value;
```

```
        logical double  sec3_pre_minecross_value;
        logical double  sec3_minecross_value;

  // retrieve old belief

PlSecMineCross.get(sec1_pre_minecross_value,sec1_minecross_value,sec2_pre_minecross
_value,sec2_minecross_value,sec3_pre_minecross_value,sec3_minecross_value);

  if( sec2_minecross_value.getValue()>0 || sec3_minecross_value.getValue()>0 )
{
   break;
}

}  // while

 // again issue original goal position to sections

 System.out.println(" Platoon Team " + pteam.name() +  " two sections , sec2 , sec3 changing
to original goal positions  ");
   @send(pteam.RolePerformer[1],ctgps.ChangeTempGolePosToSec("Change Sec 2
Gole",pteam.goal_x,pteam.goal_y,4));
  @send(pteam.RolePerformer[2],ctgps.ChangeTempGolePosToSec("Change Sec 3
Gole",pteam.goal_x,pteam.goal_y,3));

}
;
;
}
}
```

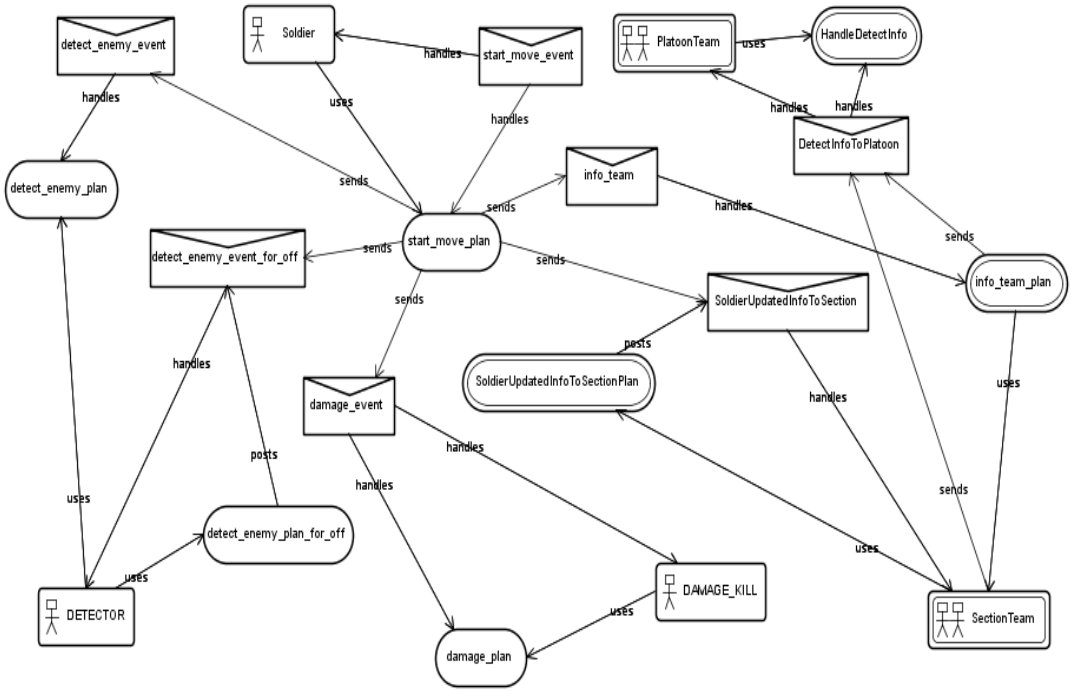
Platoon Belief plstatusinfo is read to get the positions of the mine encountering section (acting as assault section), then the other two teams goals are changed temporarily by sending the event ChangeTempGolePosToSecEvent to other section teams defined in platoon team by pteam.RolePerformer[1] and pteam.RolePerformer[2], considering that the assault section is pteam.RolePerformer[0].

7.2.4 SOLDIER AND OTHER AGENTS CAPABILITIES

The main agent-based analysis class that was identified was “SOLDIER”. The other analysis classes identified were “MASTER”, “DETECTOR”, and “DAMAGER”. These four classes, stereotyped as “agents”, had their own specific roles and capabilities:

SOLDIER agent: This represents tank entity and has the capability to simple move towards objective, move across mines and move through enemy wire

mesh . Once the SOLDIER agent has informed the MASTER agent about its own creation, it executes its relevant plans applicable in that context. While moving each SOLDIER agent keeps calling his DETECTOR agent (discussed next) by sending start detection message. SOLDIER agent while engaging enemy soldier, sends message to DAMAGER agent**,** which update the enemy kill status.



SOLDIER capabilities identified are as follows:

```
public capability soldier_cap extends Capability {

// for catering all type of move

   #handles external event start_move_event;

   #uses plan start_move_plan;
   #uses plan start_kill_move_plan;  // new added
   #uses plan start_wait_plan;
   #uses plan default_plan;
   #uses plan no_move_active_plan_blue;
   #uses plan no_move_active_plan_red;
```

```
    #sends event start_move_event;

// send the enemy detection info to section team

    #sends event info_team;

// for enabling move by section team

    #handles external event enable_move;
    #uses plan enable_move_plan;

// for enabling move speed by team cont

    #handles external event adjust_speed;
    #uses plan adjust_speed_plan;

// for detection purpose

// soldier agent receives the event deliver_enemy_array sent by Detector agent. The
deliver_enemy_array event contains nearest enemy detection information.

    #handles external event deliver_enemy_array ;
    #uses plan deliver_enemy_array_plan;

// soldier agent sends event detect_enemy_event (fr blue enemy) ,
detect_enemy_event_for_off (for red enemy )to DETECTOR agent to start detection process

    #sends event detect_enemy_event;
    #sends event detect_enemy_event_for_off;


// soldier agent sends event damage_event to DAMAGER agent for updating enemy kill
information/status
  #sends event damage_event;

// handle section team request for killed soldier info

   #handles external event Req_Killed_Soldier_Status;
   #uses plan Req_Killed_Soldier_Status_Plan ;

   // send the kiled soldier status to its team

   #sends  event Get_killed_Status_Info_from_Soldier_Event;

// for soldier changing position according to formation

  #handles external event SoldierChangePosEvent;
  #uses plan SoldierChangePosPlan;

// send soldier info at simulation loop to its section team
   #sends event SoldierUpdatedInfoToSection;

// Giving soldier creation info to MASTER
   #sends event info;

}
```

SOLDIER agent definition is as follows:

```
public agent SOLDIER  extends Agent {

   // All capabilities of soldier agents is in soldier_cap
     #has capability tanks soldier_cap;

// variable  name for assessing friendly enemy detector & damage assessor

   String  detector;
   String  damage_assesor;
   String other_damage_assesor;
   String  controller;

// Agents location variables

   public int svar1,svar2; // Soldier Start Points
   public int var1,var2;
   public int gvar1,gvar2; // Soldier Goal Points
   public int fvar1,fvar2; // Final Goal Points

   public int pre_var1,pre_var2; // Soldier previous point

// Used  for placing Mine Kill zone
  private Vector V_kill_zone = new Vector();

// Used  for placing Wire Mess zone
  private Vector W_kill_zone = new Vector();

// Agents attribute such speed,range & move active or not
   public int speed,range;
   public int kill;
   public int max_speed,min_speed;

   private int moveflag;

 // agents  enemy parameters used for computing soldier suppression
   public int en_x,en_y,en_range;

   // used for plan context purpose i.e which plan is active move, wait ,move in kill zone, ,etc

   public int move, wait, wait_time, killzone;

  // soldiers parameters to be used for upward synthesize to section level

 Public double morale_value ,leadership_value,,suppression_value,,fatigue_value
,minecross_value;

// constructor method
public SOLDIER(String name, int speed1,int range1,String detector1,String damager, String
master)
   {
    // soldier agent name
      super(name);
      name1=name;
```

```
    // detector agent name
        detector=detector1;

    // damager agent name
        damage_assesor=damager;

    // section team name
        controller=master;
   // initialize agents cur, goal, pre position
        svar1=0;
        svar2=0;
        var1=0;
        var2=0;
        gvar1=0;
        gvar2=0;
        fvar1=0;
        fvar2=0;
        pre_var1=0;
        pre_var2=0;

// soldier detection range
  range=range1;
// soldier max speed
  max_speed=speed1;
// solider kill status
  kill=0;
// soldier move flag enable
  moveflag=1;  // able to move
;
;
}

}
```

Other agents identified in this scenario are as follows:

MASTER agent: Keeps track of all joining entities in the simulation battlefield. It acts as a simulation controller and sends events at fixed interval to all joining entities present in the simulation. While SOLDIER agents are created, they also have to inform the MASTER agent.

```
public capability controllers extends Capability {

// for controlling agents

    #handles external event info;
    #uses plan info_plan;

    #handles external event start_sim;
    #uses plan start_sim_plan;

    #handles external event start_sim_from_team;
```

```
   #uses plan start_sim_from_team_plan;

}
```

MASTER AGENTS Class:

```
public agent MASTER extends Agent {

#posts event start_sim ev;
#has capability controllers controllers_cap;

// Master Agents name
   String name1;

// instance of soldier agent container
  agents_container agents_cont;

// Used  for placing created agents ames
    Vector agents_name = new Vector();

    public MASTER(String name,agents_container agents_cont)
    {
        super(name);
              name1=name;
        this.agents_cont = agents_cont;

        send_counter=-1;
 }

public String NameAt(int i)
{
return( ((String)agents_name.elementAt(i)) );

}

public  void Set_counter(String s)
{
 agents_name.addElement(s);
 System.out.println( " Master Counter incremented to " + count() );
}

public void start_sim()
    {
      System.out.println( " STARTING  SIMULATION   ");
       postEvent(ev.request(3));
    }

}
```

DETECTOR agent: Each force has one such agent. It is assigned the task to detect enemy. It keeps information about the enemy force within 60 meters range. For example Red's DETECTOR will keep information about red enemy

within 60 meters range and send this information to red /blue SOLDIER agents.

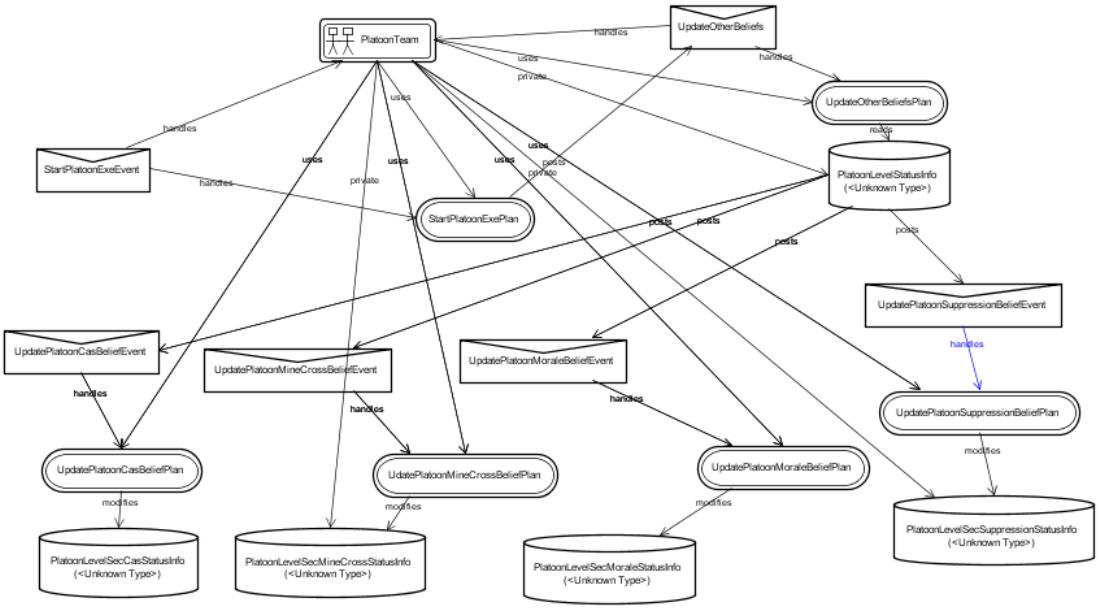
DAMAGER agent: Each force has one such agent. If the enemy tank gets killed, this agent updates its enemy database. For example, if any red tank kills enemy (blue) tanks, it informs this to red's / blue's DAMAGER agent to update enemy kill status.

An important design and analysis issue is that of dealing with concurrency. Multiple agents, each having multiple threads of execution (plans) have to be modeled. The complexity of designing the multi-threaded, multi-agent system places high demands on the system architect & programmer. However, concurrency is one of the most important potential advantages offered by multi-agent systems and these concepts are very easily handled by the various Java-based agent toolkits available commercially.

7.2.5 COMPLEX BELIEF GENERATION FROM SYNTHESIZED BELIEF

StartPlatoonExeEvent event is handled by StartPlatoonExePlan of Platoon team. The plan StartPlatoonExePlan, while executing platoon team plan further the event UpdateOtherBeliefs event for generating complex belief based on simple belief about platoon's sections stored in PlatoonLevelStatusInfo.Bel. This event UpdateOtherBeliefs is handles by plan UpdateOtherBeliefsPlan. This Plan UpdateOtherBeliefsPlan reads the data PlatoonLevelStatusInfo.Bel. From belief PlatoonLevelStatatesInfo.Bel ,this plan generates data for four type of complex belief namely:

- PlatoonLevelSecCasStatusInfo.Bel
- PlatoonLevelSecCasStatusInfo.Bel
- PlatoonLevelSecMoraleStatusInfo.Bel
- PlatoonLevelSecSuppressionStatusInfo.Bel.



The plan after generating separate data values for different belief posts four events (UpdatePlatoonCasBeliefEvent,UpdatePlatoonMineCrossBeliefEvent,Update PlatoonMoraleBeliefEvent and UpdatePlatoonSuppressionBeliefEvent).

The UpdatePlatoonCasBeliefEven event is handles by plan UpdatePlatoonCasBeliefPlan of platoon team and is responsible for updating / adding the derived belief about platoon's three section casualty value in PlatoonLevelSecCasStatusInfo.Bel.

Similarly the event **UpdatePlatoonMineCrossBeliefEvent** is handles by the plan **UpdatePlatoonMineCrossBeliefPlan** and is repossible for updating the mine cross status of the three sections of platoons stored in **PlatoonLevelSecMoraleStatusInfo.Bel** belief.

7.2.6 COMPANY LEVEL DECISION MAKING BASED ON COMPANY'S BELIEF (PLATOON CASUALTY BELIEF) UPDATION

The belief CoyLevelPlatoonCasStatusInfo.Bel containing the casualty information of its platoons. Whenever the casualty value of assault platoon crosses the given threshold value , an event GetCoysIstPlatoonSupEvent is generated by CoyLevelPlatoonCasStatusInfo.Bel and is handled by the Company Plan CoyIstPlaSupPlan.

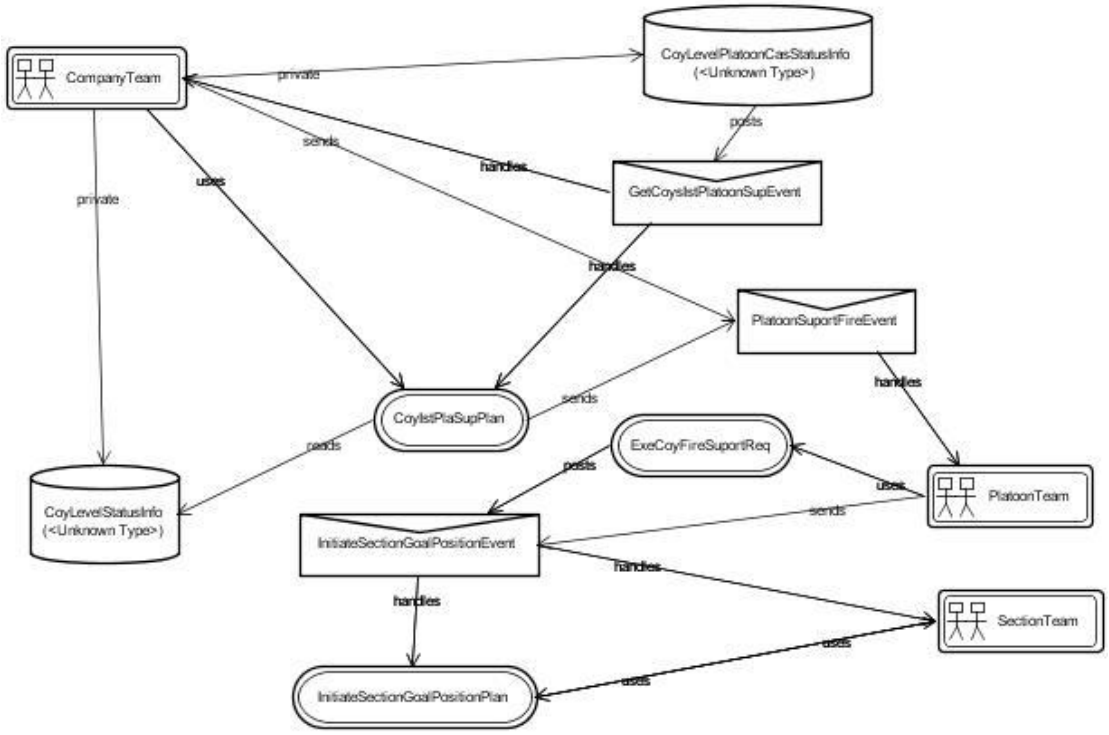
The beliefset CoyLevelPlatoonCasStatusInfo is as follws :

```
public beliefset CoyLevelPlatoonCasStatusInfo extends  ClosedWorld
 {

       #value field double pla1_cas_value;
       #value field double pla2_cas_value;
       #value field double pla3_cas_value;

        // get complete information of a named Team

      #linear query get(logical double pla1_cas_value, logical double pla2_cas_value,logical
double pla3_cas_value);


#function query int MaxCasPlatoon()
{
logical double a,b,c;

get(a,b,c);

int pla_index=0;

if( b.getValue()>c.getValue())
  pla_index =2;
else
  pla_index=3;

return(pla_index);

}


#function query int MinCasPlatoon()
{
logical double a,b,c;

get(a,b,c);

int pla_index=0;

if( b.getValue()<c.getValue())
```

```
  pla_index =2;
else
  pla_index=3;

return(pla_index);

}

       #posts event GetCoysIstPlatoonSupEvent IstPlaSup;


      public void newfact(Tuple t,BeliefState is,BeliefState was)
      {
        // Note that CoyLevelPlatoonCasStatusInfo__Tuple contains two underscores

       CoyLevelPlatoonCasStatusInfo__Tuple pt = (CoyLevelPlatoonCasStatusInfo__Tuple) t;


         if (pt.pla1_cas_value >= 40)
         {
          // code to post the trigger event. This code will

          // be executed whenever a Team playing pla1 cas incr above 40   mark is
          // added in  belief set. For example:

         //  postEvent(PlSecCas.Cas(pt.pla1_cas_value,1));


       // first platoon of coy suffered heavy cas ( > 40%) ,so requires fire support from other
platoons

        postEvent(IstPlaSup.IstPlaSupEvent(pt.pla1_cas_value,1));

         }
      }

}
```

Design: Coy_level_sup_platoon_event_generation

Coy team 's , platoon casuality belief triggering support fire event to platoons having high morale value.



The CoyIstPlaSupPlan reads data CoyLevelStatusInfo.Bel to retrieves the position of assault platoon so that fire support team can be sent to assault position for support. This Plan also reads data CoyLevelStatusInfo.Bel for getting the morale value of support platoons so that high morale platoons can be selected for fire support to assault teams.

Description of plan CoyIstPlaSupPlan is as follows:

```
teamplan CoyIstPlaSupPlan extends TeamPlan {
   #handles event GetCoysIstPlatoonSupEvent pc1;
    #uses interface CompanyTeam  cteam;
     #sends event PlatoonSuportFireEvent sfe;
     #reads data CoyLevelStatusInfo coystatusinfo;
     body()
   {
    System.out.println("   Coy Team " + cteam.name() + " receiving trigger event for First
platoon casualty > 30% + causality value  " + pc1.CasVal );
     logical double a1,b1,c1,d1,e1,f1,g1,h1;
     double g11,h11;
     coystatusinfo.get(cteam.RolePerformer[0],a1,b1,c1,d1,e1,f1,g1,h1);
     g11=g1.getValue();
     h11=h1.getValue();
    // select high morale Platoon
    int pla_index=2; // select 2 nad paltoon as default
   // get 2 nd Role Platoon's Morale Information
    logical double a2,b2,c2,d2,e2,f2,g2,h2;
    coystatusinfo.get(cteam.RolePerformer[1],a2,b2,c2,d2,e2,f2,g2,h2);
   // get 3 rd Role Platoon's Morale Information
    logical double a3,b3,c3,d3,e3,f3,g3,h3;
    coystatusinfo.get(cteam.RolePerformer[2],a3,b3,c3,d3,e3,f3,g3,h3);
   if ( a2.getValue() >  a3.getValue() )
    {pla_index=1;  System.out.println("  Morale of Platoon 2 is High " + a2.getValue() + " 3 rd
"+ a3.getValue()  );}
   else{pla_index =2;  System.out.println("  Morale of Platoon 3 is High " + a3.getValue() + "2
nd " + a2.getValue()  );}
```

```
System.out.println("  Coy Team selected high morale Platoon team " +
cteam.RolePerformer[pla_index]  + " for support Role  to Assault platoon " +
cteam.RolePerformer[0] );
@send(cteam.RolePerformer[pla_index],sfe.MsgToSupportFire("FireSupportMsgToPlatoon",(
int)g11,(int)h11));
  }
}
```

The plan then sends event PlatoonSuportFireEvent to the support platoon for providing fire support. The Platoon has plan handler ExeCoyFireSuportReq.plan for this event PlatoonSuportFireEvent sent by company team. The plan ExeCoyFireSuportReq of platoon in turn sends the event InitiateSectionGoalPositionEvent1 to all section teams for changing their goal given by support platoon team. The event InitiateSectionGoalPositionEvent is handled by plan InitiateSectionGoalPositionPlan of Section Team.

The description of plan ExeCoyFireSuportReq is as follows:

```
teamplan ExeCoyFireSuportReq extends TeamPlan {
    #handles event PlatoonSuportFireEvent sfe;
    #uses interface PlatoonTeam pteam;
// send platoon goal to all sections
 #sends event InitiateSectionGoalPositionEvent isgpe;

  context() {
( pteam.fire_support_once==1);
}
   body()
   {
        System.out.println(" Platoon team " + pteam.name1+ " receiving company  request for
providing fire support ta position  x "+ sfe.x  +  " position  y "+  sfe.y );
```

```
      pteam.goal_x= sfe.x;
      pteam.goal_y= sfe.y;
   // giving fire support
      pteam.fire_supporting=1;
      pteam.fire_support_once=0; // so that it can not trigger fire sup repeteadily
    System.out.println(" PLATOON TEAM  " + pteam.name1 + " giving support Role to pla 1  ");
       // initiate Platoon 's secion's soldier initial position and Goal position to all sections of red
section  only
 for(int i=0;i<4;i++)
  {
     System.out.println("Platoon Team New Goal Passing to section no. " + i);
 @send(pteam.RolePerformer[i],isgpe.MsgToInitateSecGoal(pteam.goal_x,pteam.goal_y)  );
 }
  }
}
```

Note : **Plan ExeCoyFireSuportReq** of Platoon team sends event **InitiateSectionGoalPositionEvent** to all sections to change their goal position to new goal position (assault platoon current position)

7.3 Experimental Results

(Red Platoon attacking a blue Section)

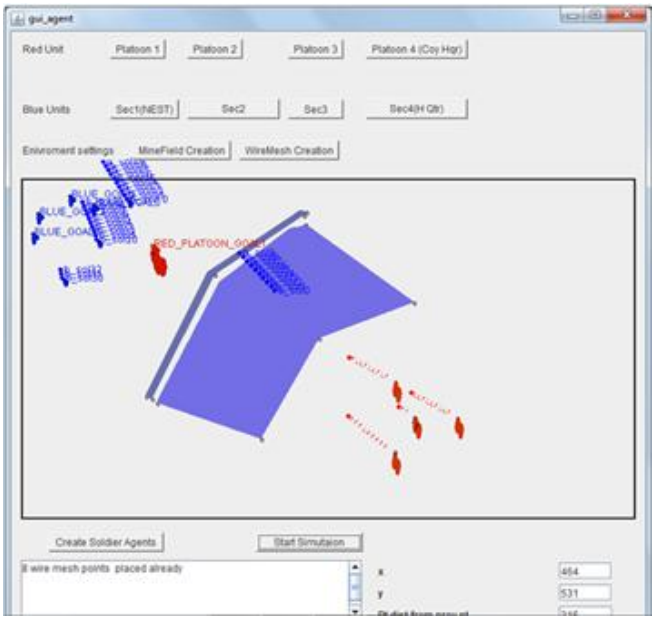


Figure 7.1 Red platoon is moving towards objective in one up formation

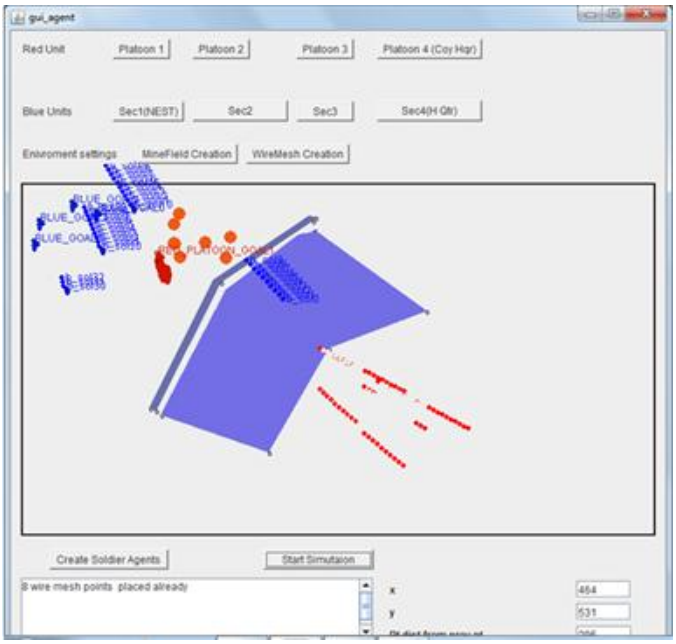


Figure 7.2 Assault section on encounters minefield request platoon commander for arty fire

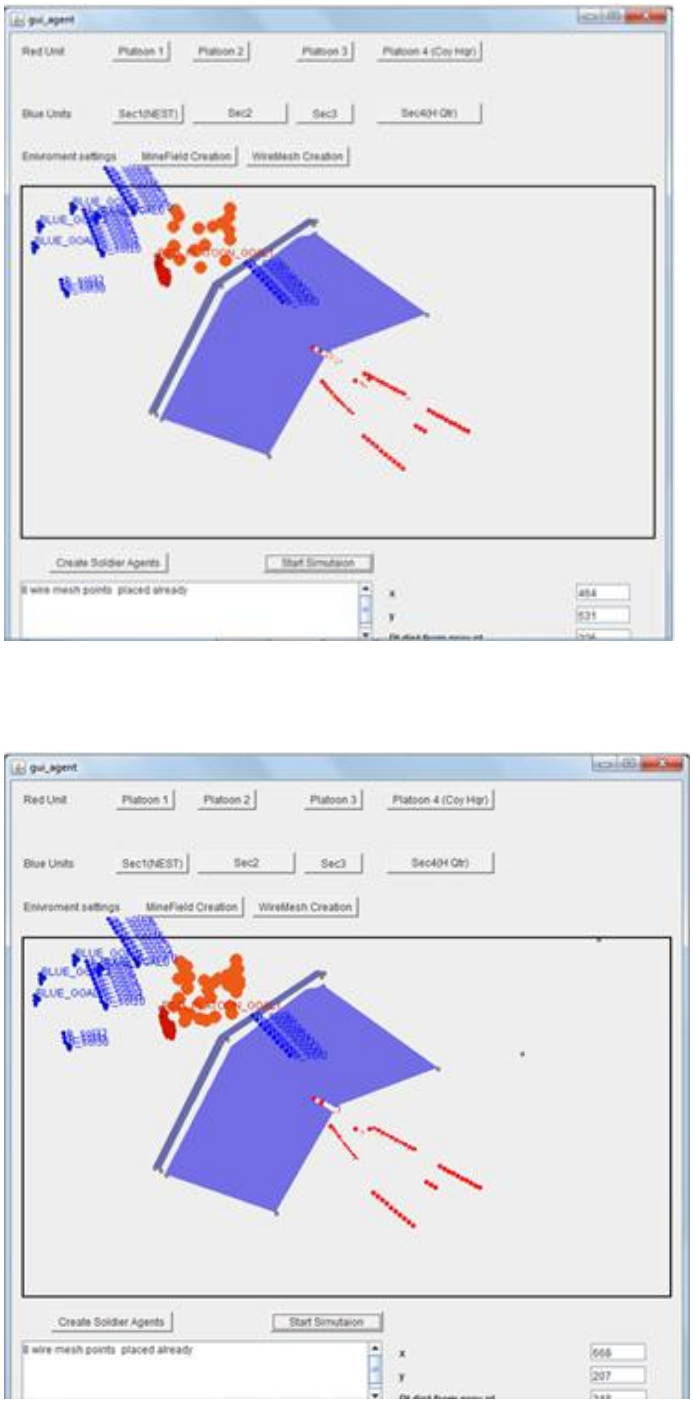


Figure 7.3 Platoon commander gives order to section to adjust in rod formation as soon as Assault section encounters minefield.

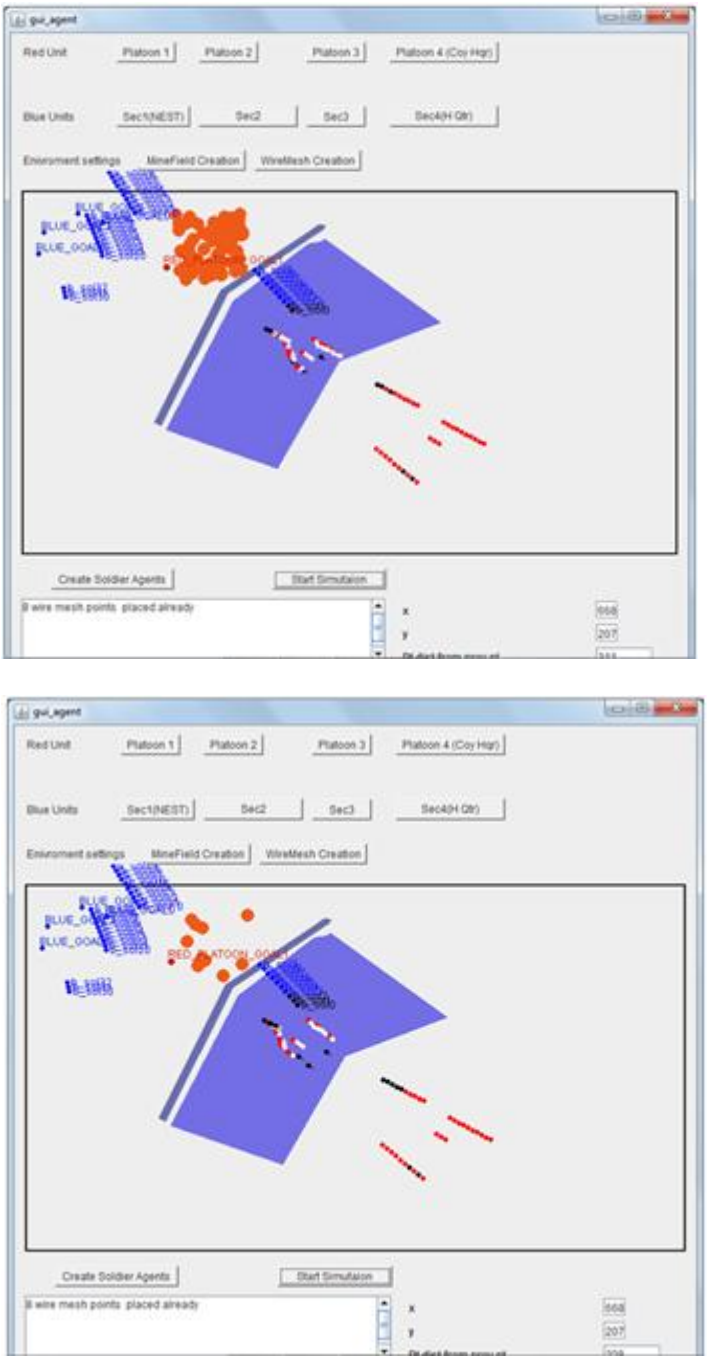


Figure 7.4 Assault section request to Platoon commander for fire support

Assault section suffers blue forces ambush fire, It request platoon commander for fire support from other sections. Section with high morale moves towards the enemy for engagement.

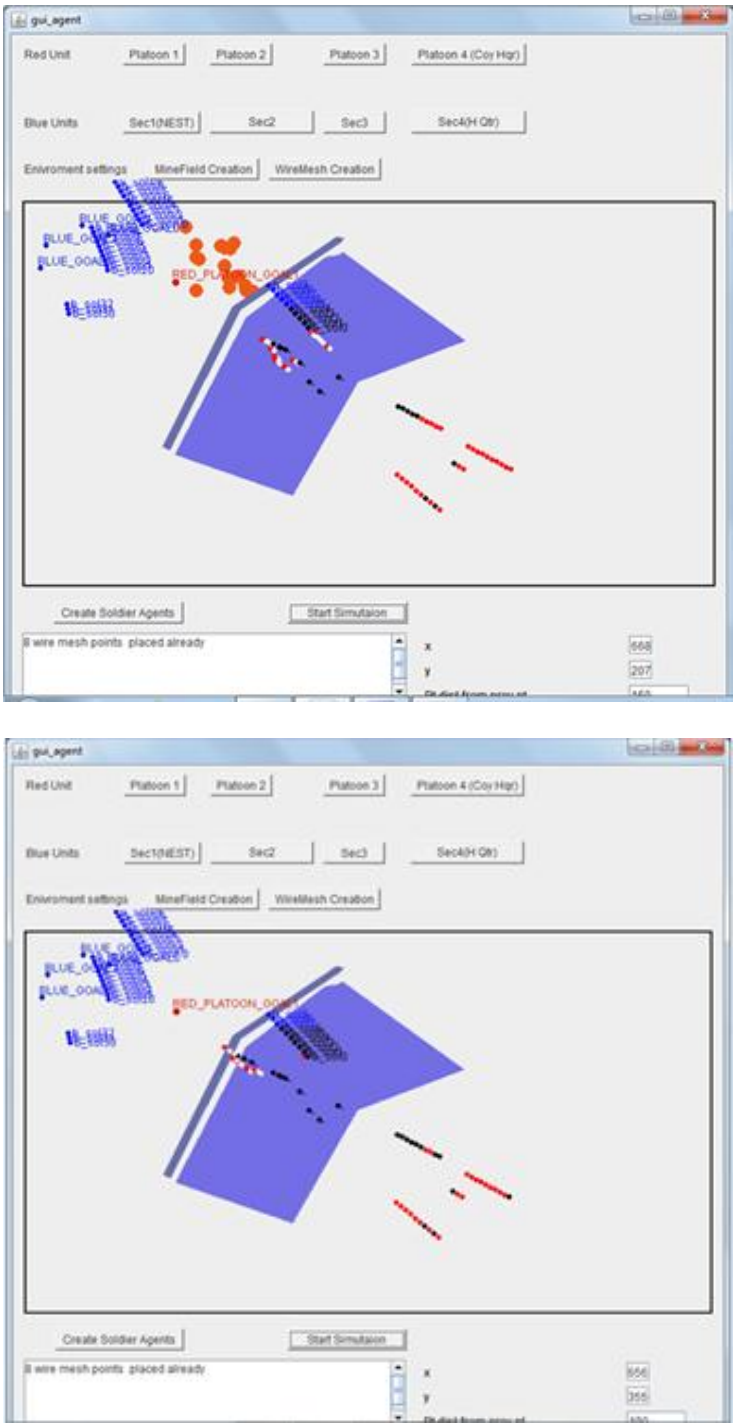


Figure 7.5 Assault section crosses mine field completely & request platoon commander to lift Arty fire and fire support.

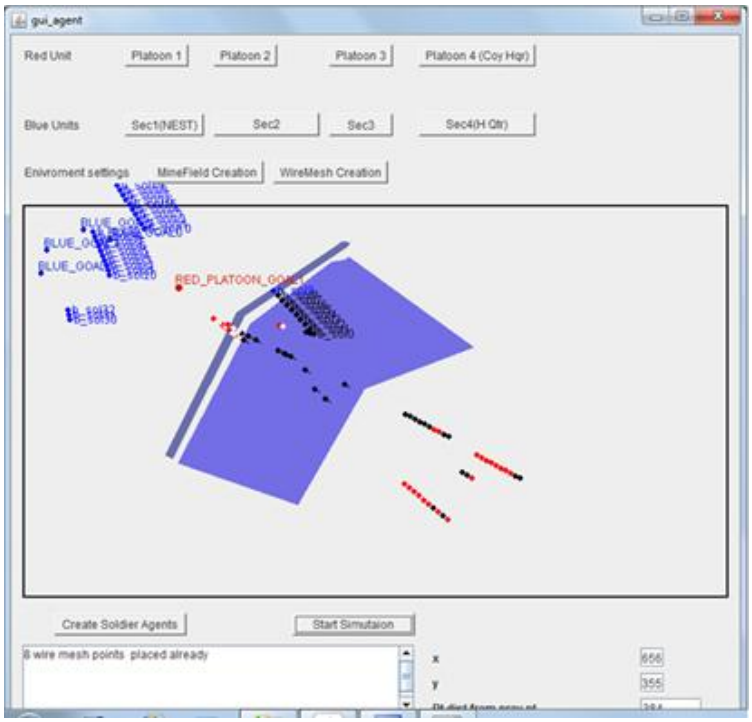


Figure 7.6 Fire support section on receiving platoon order stop enemy engagement & moves towards actual platoon objective.

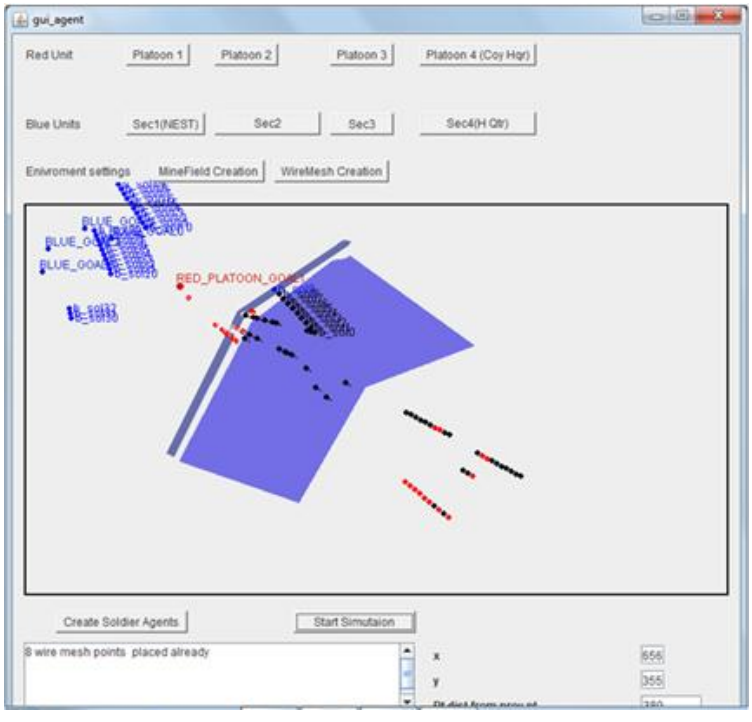


Figure 7.7 Red Platoon completely crosses the mine fields and adjust itself in lean on position covering blue enemy from front.

7.4 Summary

Agent Unified modeling language (AUML) , is an extensions to the UML and is used for the design of agents and teams . The AUML based detailed design is done using JACK design tool. The detailed designing of agents, team, capability, events, plan, belief, team data is then implemented using JACK language. The agent based modeling approach has been found suitable for representation of military C&C Hierarchy & tactical team behavior.

This Study has successfully shown the team formation according to required Roles, team task delegation from company to platoon and from platoon to section teams. From section teams the mission orders are passed to actual soldier agents, who actually interact with battlefield environment. All the required events & behavior invocation of all command agents teams has been validated from the log files generated from simulation. All normal & Critical BDI events are also generated & has been validated from log files.

This Study has shown that the team belief can be propagated upward the organizational hierarchy allowing the upper level teams (Command agents) to generate their complex belief derived from lower level sub teams leading to dynamic response to unforeseen battlefield situation.

Chapter 8. Conclusion & Future Scope

8.1 Conclusion

The intelligent agent is a valuable software concept which has the potential to be more widely used in defence command decision modeling as it overcomes a number of the limitations of present approaches to modeling human reasoning and team behaviour. Practical experience has shown that multi-agent architectures are suited to implementing simulation and decision support systems.

A number of agent based defence applications have demonstrated that BDI agents provide the most appropriate underpinning architecture for representing human decision making, including a formalism for expressing team structures and behaviors necessary to model C3.

From the results of the work done in thesis (red platoon attacking a blue section & red company attacking a blue platoon) , we can highlight some salient features of the agent based modeling approach:

Having an “agent-oriented mind-set” while modeling tactical scenarios will enable us to map the key abstractions and entities involved into teams of agents, which is separate from simulation-specific code, hence encouraging separation of concerns and re-use.

The work breakdown in the military hierarchy is mapped directly into roles of team members. Hence, this allows for hierarchical decomposition of tasks.

This approach allows the team-tactics of military operations to be captured and simulated with minimal effort. It encourages clear and concise description of coordinated activities

It allows the abstraction of what needs to be done from how it is done, i.e. the responsibilities of the team can be written down without consideration of how the roles would be fulfilled and implemented by the team members.

This Study has successfully shown that the team belief can be propagated upward the organizational hierarchy allowing the upper level teams (command Agents) to generate their complex belief derived from lower level sub teams leading to dynamic response to unforeseen battlefield situation. These computer generated forces (command agents) based on synthesized belief of lower level units takes effective decision under any unforeseen situation. The propagated belief of lower units helps the command agents to take account of current situation of friendly as well as rival units in battlefield.

JACK is a mature, cross-platform environment for building, running and integrating commercial-grade multi-agent systems. BDI is an intuitive and powerful abstraction that allows developers to manage the complexity of the problem. JACK provide support for team oriented programming. The current version supports the BDI model and SimpleTeam, an extension to support team-based reasoning.

8.2 Future Scope

In our scenario, we have made Company level as Commanding agent .This can be extendible to brigade level as CA with minimal effort provided the brigade level knowledge of decision making is available and can be stored in brigade belief.

The commanding agent can play vital role in network centric warfare (NCW) operation , where information is decentralized and available to the federate forces (Commanding agents) based on their capability.

The success of these applications has stimulated current research into dynamic team formation, detachment ,recognition of enemy intentions and learning.

It is important to capture the realistic aspects of human teams for Human-Behavior Representation(HBR) studies, such as the effects of workload on communication or coordination, or reaction to time-pressures and stress. While intelligent agents have a great potential for modeling teamwork in HBR simulations, much work remains to be done to accurately represent cognitive aspects of human team members, like making effective decisions, heuristics for dealing with uncertainty and workload limitations, and the effects of these cognitive aspects on team interactions in real, human teams.

REFERNCES

[1] Cioppa, T. M. 2003. Advanced experiment designs for military simulations. Technical Document TRAC-MTR- 03-11, U.S. Army TRADOC Analysis Center, Naval Postgraduate School, Monterey, California.

[2] Ilachinski, Irreducible Semi-Autonomous Adaptive Combat (ISAAC): An Artificial-Life Approach to Land Warfare, Center for Naval Analyses Research Memorandum CRM 97-61, June1997.

[3] Andrew Lucas, The Potential For Intelligent Software Agents In Defence Simulation, ,Agent Oriented Software Pty. Ltd.,Melbourne, Australia,Cal@Agent-Software.Com.Au

[4] Thomas M. Cioppa, Military Applications Of Agent-Based Simulations, Training and Doctrine,Command Analysis Center,Postgraduate School,Monterey, CA 93943, U.S.A. , Thomas W. Lucas, Susan M. Sanchez,Operations Research Department,Naval Postgraduate School,Monterey, CA 93943, U.S.A.

[5] Aparna Malhotra,Agent-Based Modeling in Defence
Institute for Systems Studies & Analyses, Metcalfe House, Delhi-110 054

[6] Ferber, J. 1999. Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Boston, Massachusetts: Addison-Wesley.

[7] Sumant Mukahjee,Agent based implementation of automated command and control process, ISSA,DRDO,Delhi, International Conference on Cognitive Sciences (ICCS),2004, Organized by NIIT at IIT, Delhi.

[8] Mapping Cognitive work analyses (CWA) to an intelligent Agents Software Architechture:Command Agants , F. Lui,Marcus Watson,Proceedings of the defense Human Factors special Interest Group (DHFSIG) 2002,DSTO Melbourne.Australia, 21-2 November.

[9] Dr. Ralph Rönnquist,Dr. Andrew Lucas,Mr. Nick Howden,The Simulation Agent Infrastructure (SAI) – Incorporating Intelligent Agents into the CAEN Close Action Simulator,Agent Oriented Software,221 Bouverie Street,CARLTON VIC 3053,AUSTRALIA

[10] Xiaocong Fan, John Yen,School of Information Sciences and Technology Modeling and Simulating Human Teamwork Behaviors Using Intelligent Agents, The Pennsylvania State University ,University Park, PA16802
Email: *f*zfan, jyen*g*@ist.psu.edu

[11] Richard Crowder1, Helen Hughes2,Yee W Sim1 and Mark Robinson2. An Agent Based Approach To Modeling Design Teams, 1 School of Electronics and Computer Science, University of Southampton, Southampton UK. 2 Leeds University Business School, Leeds, UK., International Conference On Engineering Design, Iced'09,24 - 27 August 2009, Stanford University, Stanford, CA, USA

[12] Aparna Malhotra, Sanjay Bisht, S.B. Taneja. Using Intelligent Agents to Simulate Battle Tank Tactics, International Conference on Cognitive Sciences (ICCS), 2004, Organized by NIIT at IIT, Delhi.

[13] Aparna Malhotra, S.B. Taneja, Sanjay Bisht. Simplifying the Tactical Modelling of Armour scenarios using Teams of Agents, 3rd International Conference on Quality, Reliability and Infocom Technology (ICRQIT'06), 2 - 4 Dec, 2006, New Delhi.

[14] Sanjay Bisht, Aparna Malhotra, S.B. Taneja. Modelling and Simulation of Tactical Team-behaviour, published in Defence Science Journal, Nov, 2007.

[15] Devasheesh Banerjee, Sanjay Bisht ," An Agent based team oriented tactical simulation" , 3rd International Conference on Quality,Reliability and Infocom Technology (ICRQIT'06), 2 -4 Dec, 2006, New Delhi.

[16] Tweedale, J., Sioutis, C., Phillips-Wren, G., Ichalkaranje, N., Urlings, P., Jain, L.: Future directions: Building a decision making framework using agent teams. In: Phillips-Wren, G., Ichalkaranje, N., Jain, L. (eds.) Intelligent Decision Making: An AI-Based Approach, pp. 387–408. Springer, Berlin (2008)

[17] JACK™ Intelligent Agents Evaluation Version, downloaded from site,

[18] JACK Intelligent Agents® Agent Manual,, Copyright © 1999-2011, Agent Oriented Software Pty. Ltd.

[19] JACK® Intelligent Agents Teams Manual ,Copyright © 2002-2011, Agent Oriented Software Pty Ltd, Release 5.3,21 November 2011.

[20] Salas, E., Dickinson, T.L., Tannenbaum, S.I., and Converse, S.A. (1992). Toward and Understanding of Team Performance and Training. in: *Teams, Their Training and Performance*. R.W. Swezey and E. Salas (eds.). Ablex: Norwood, NJ. pp. 3-29.

[21] Gorman, P. (1980). The Command Post is Not a Place. http://www.ida.org/DIVISIONS/sctr/cpof/

[22] Drillings, M. and Serfaty, D. (1997). Naturalistic Decision Making in Command and Control in: *Naturalistic Decision Making*. C.E. Zsambok and G. Klein (eds.). Erlbaum: Mahwah, NJ. pp. 71-80.

[23] Klein, G. (1999). *Sources of Power: How People Make Decisions*. MIT Press.

[24] Tambe, M. (1997). Towards Flexible Teamwork. *Journal of Artificial Intelligence Research*, 7:83-124.

[25] Grosz, B. and Kraus, S. (1996). Collaborative Plans for Complex Group Action. *Artificial Intelligence*, 86:269-357.

[26] Rao, A.S. and Georgeff, M.P. (1995). BDI Agents: From Theory to Practice. *Proceedings of the First International Conference on Multi-Agent Systems*, 312-319.

[27] Kitano, H., Kuniyoshi, Y., Noda, I., Asada, M., Matsubara, H. and Osawa, E. (1997). RoboCup: A Challenge Problem for AI. *AI Magazine*, 18(1):73-85.

[28] Jones, R.M., Laird, J.E., Nielsen, P.E., Coulter, K.J., Kenny, P., and Koss, F.V. (1999). Automated Intelligent Pilots for Combat Flight Simulation. *AI Magazine*, 20(1):27-41.

[29] Paolucci, M., Kalp, D., Pannu, A., Shehory, O., and Sycara, K. (1999). A Planning Component for RETSINA Agents. *Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages*, 147-161.

[30] Tidhar, G., Heinze, C., and Selvestrel, M.C. (1998). Flying Together: Modeling Air Mission

[31] Georgeff, M.P.; Lansky, A.L. (1986) "Procedural knowledge", *Proceedings of the IEEE special issue on knowledge representation*, 74:1383--1398.

[32]. M. E. Bratman, Intentions, Plans, and Practical Reason, Harvard University Press, Cambridge,MA, 1987.

[33]. M. P. Georgeff, A. L. Lansky, Procedural knowledge, Proceedings of the IEEE Special Issue on Knowledge Representation 74 (1986) 1383–1398.

[34]. M. J. Huber, JAM: a BDI-theoretic mobile agent architecture, in: Proceedings of the Third International Conference on Autonomous Agents (Agents'99), Seattle, USA, 1999, pp.236–243.

[35] M. d'Inverno, D. Kinny, M. Luck, M. Wooldridge, A formal specification of dMARS,in: M. P. Singh, A. S. Rao, M. Wooldridge (Eds.), Intelligent Agents IV: Proceedings of the Fourth InternationalWorkshop on Agent Theories, Architectures, and Languages, Springer-Verlag LNAI 1365, 1998, pp. 155–176.

[36] L. Braubach, A. Pokahr, W. Lamersdorf, Jadex: A short overview, in: Proceedings of Net.ObjectDays: AgentExpo, 2004, pp. 195–207.

[37] Bratman, M.E., Isreal, D.J., Pollack, M.E.: Plans and resource-bounded practical reasoning.Computational Intelligence 4 (1988)

[38] G. Weiss, (ed.), Multiagent System: A modern approach to Distributed AI, MIT Press, 1999.

[39] M. Wooldridge, P. Ciancarini, and G. Weiss (eds.), Proceedings of the 2nd International Workshop on Agent-Oriented Software Engineering (AOSE-2001), Montreal, CA, May 2001.

[40] M. Wooldridge and N. R. Jeanings, ''Intelligent agents: Theory and practice,'' Knowl. Eng. Rev., vol.10, no. 2, 1995.

[41]. A. Newell, ''The knowledge level,'' Artif. Intell., vol. 18, pp. 87–127, 1982.

[42] A. S. Rao and M. P. Georgeff, ''Modelling rational agents within a BDI-architecture,'' in Proceedings of Knowledge Representation and Reasoning (KRR-91) Conference, San Mateo CA, 1991.

[43] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopolous, J., & Perini, A. "Tropos: An agent-oriented software development methodology" *Autonomous Agents and Multi-Agent Systems* , (2004)., *8*(3), 203-236

[44] The Tropos Metamodel and its Use ,Angelo Susi and Anna Perini and John Mylopoulos ITC-irst, Via Sommarive, 18, I-38050 Trento-Povo, Italy , Paolo Giorgini Department of Information and Communication Technology,University of Trento, via Sommarive 14, I-38050 Trento-Povo, Italy

[45] Agent-Oriented Methodologies: An Introduction, Paolo Giorgini University of Trento, Italy Brian Henderson-Sellers University of Technology, Sydney, Australia, Copyright © 2005, Idea Group Inc.

[46] H. Nwana, ''Software agents: An overview,'' Knowl. Eng. Rev. J., vol. 11, no. 3, 1996.

[47] L. K. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, Non-Functional Requirements in Software,Engineering, Kluwer Publishing, 2000.

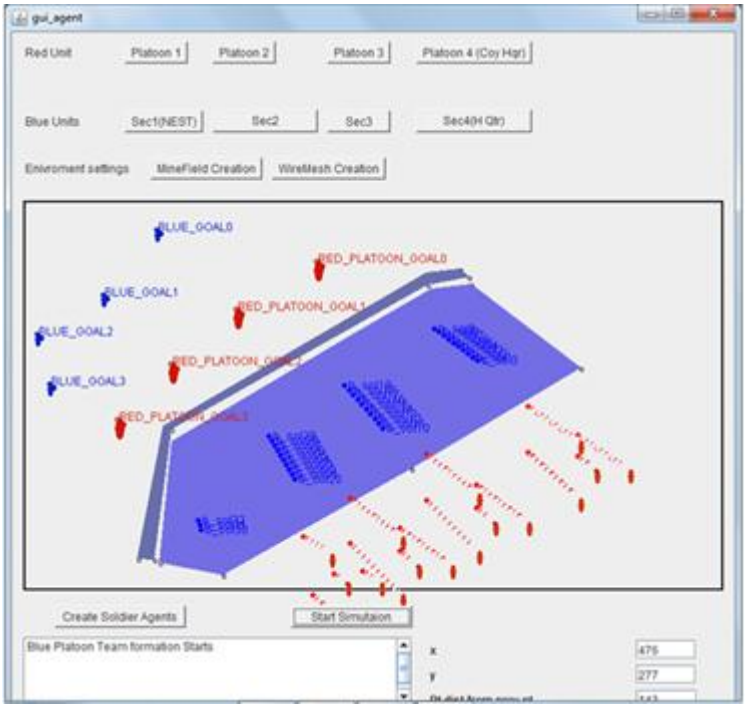
[48] Bernhard Bauer, Jrg P. Mller, and James Odell. Agent UML: A formalism for specifying multiagent interaction. In Paolo Ciancarini and Michael Wooldridge,

editors, Agent-Oriented Software Engineering, pages 91–103, Berlin, 2001. Springer-Verlag.

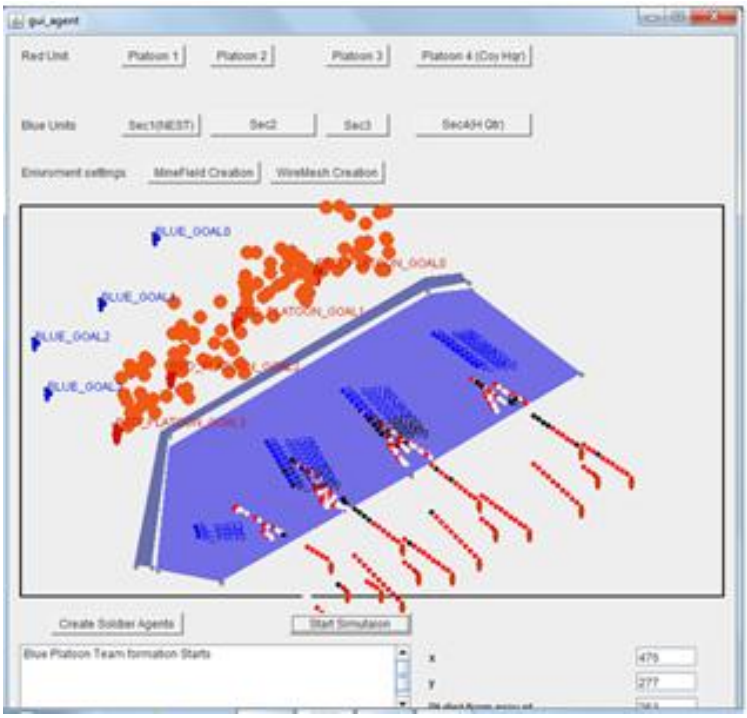
[49] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Extending UML for Agents.In Proceedings of AOIS Workshop at AAAI, 2000.

APPENDIX A: SCREEN SHOTS

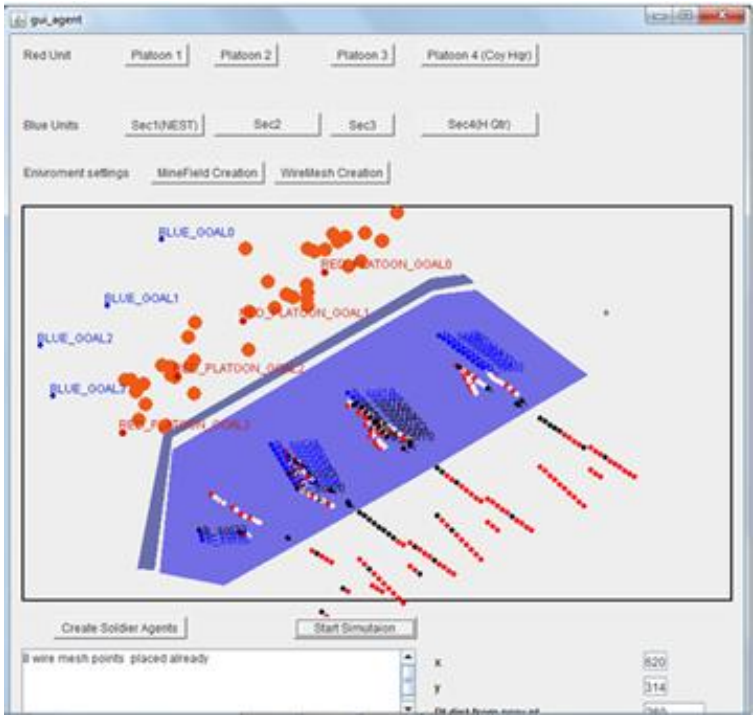
(Red Company attacking a blue Platoon)



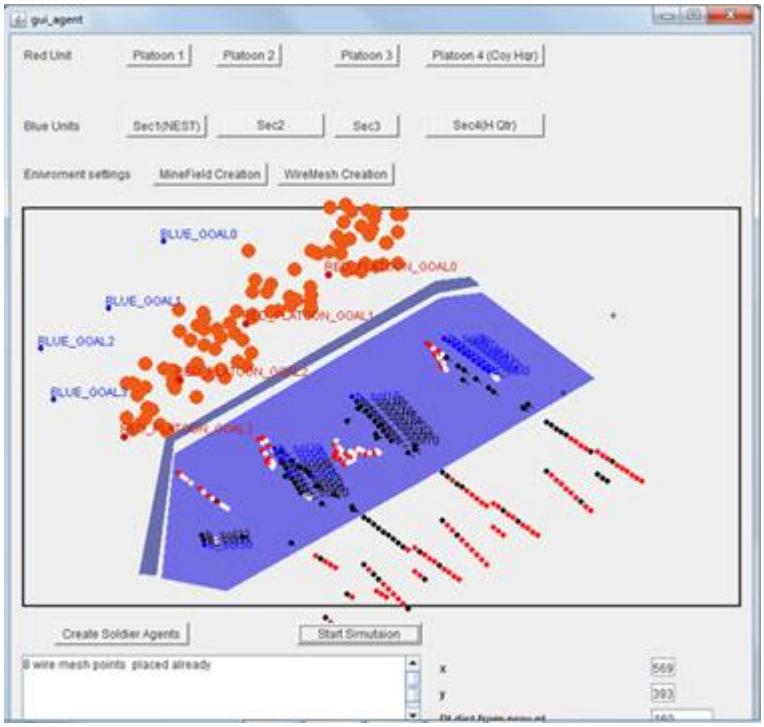
Red Company consisting of four platoon (Two Up formation) is moving towards their respective objective.

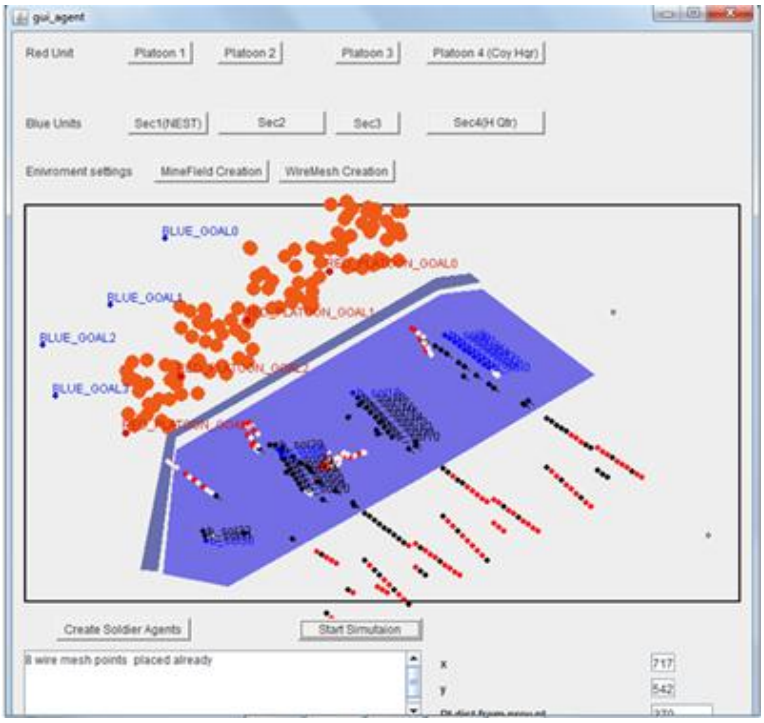


Assault sections of all platoons on encountering minefield request platoon commander for arty fire. Platoon commander gives order to sections to adjust in rod formation as soon as Assault section encounters minefield.

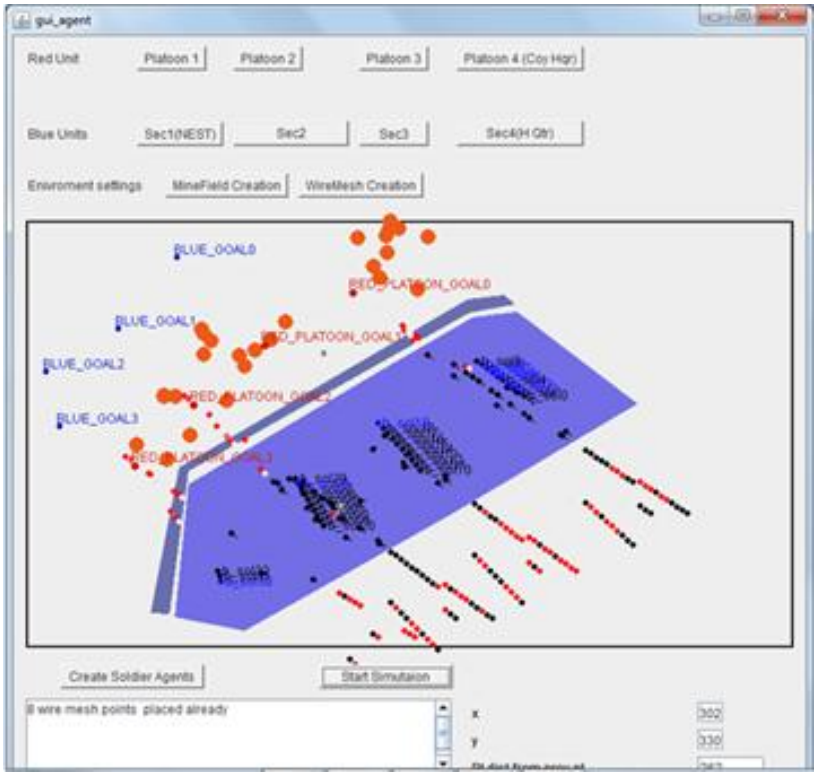


Assault Platoon suffers blue forces ambush fire, It request company commander for fire support from other platoons.

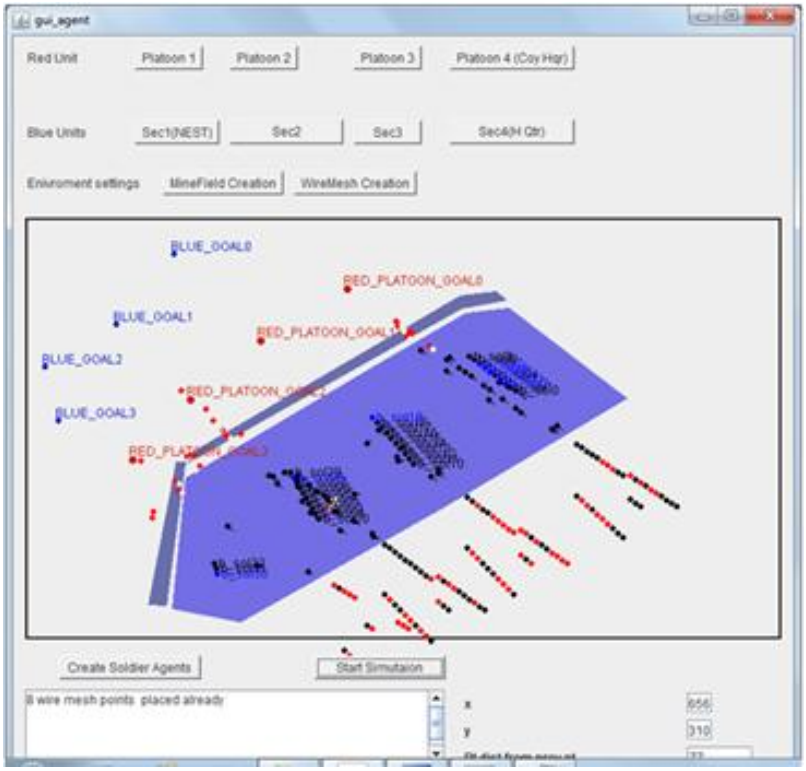




Support Platoons with high morale moves towards the enemy position of assault Platoon for engagement.



Assault Platoons is in the process of crossing mine field completely. It request platoon commander to lift Arty fire and fire support.



Assault section crosses mine field completely. Platoon commander lifts Arty fire support. Fire support section on receiving platoon order stops enemy engagement & moves towards actual platoon objective.

APPENDIX B : ACRONYMS

DoD : Department of Defense

CAEN : Close Action Environment

ISAAC : Irreducible Semi-Autonomous Adaptive Combat

SAI : Simulation Agent Infrastructure

ABS : Agent Based Simulation

CGF : Computer generated Forces

C&C: Command & Control

C3: Command, Control & Communication