1-1-1995

Cooperative Behavior In ModSAF

Sumeet Rajput

INSTITUTE FOR SIMULATION AND TRAINING

Contract Number N61339-92-C-0045
U.S. Army STRICOM
20 November 1995

Cooperative Behavior in ModSAF

Sumeet Rajput
Clark R. Karr

Institute for Simulation and Training
3280 Progress Drive
Orlando FL 32826

University of Central Florida
Division of Sponsored Research

IST-CR-95-35

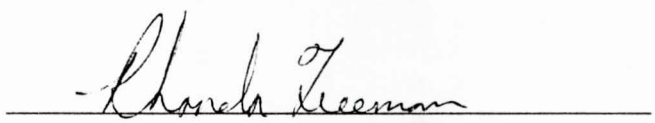
INSTITUTE FOR SIMULATION AND TRAINING

Cooperative Behavior in ModSAF

IST-CR-95-35
20 November 1995

Prepared For:
U.S. Army STRICOM
N61339-92-C-0045

Reviewed by:
Rhonda Freeman



Prepared by:
Sumeet Rajput
Clark R. Karr

Institute for Simulation and Training • 3280 Progress Drive • Orlando, Florida 32826

University of Central Florida • Division of Sponsored Research

Cooperative Behavior in ModSAF

IST-CR-95-35

Contract N61339-92-C-0045
November 20, 1995

Sumeet Rajput and Clark R. Karr

Table of Contents

List of Figures

List of Tables

1. Executive Summary

Success in representing individual vehicles within Computer Generated Forces (CGF) systems has allowed researchers to focus on representing CGF groups (units). Like vehicles in a real battlefield, CGF vehicles must cooperate with each other to achieve battlefield objectives. This document reports IST's research into mechanisms for cooperative behaviors in CGF.

An example of a cooperative behavior is *Bounding Overwatch*. A platoon uses this movement tactic when enemy contact is imminent. One section of the platoon moves (bounds) while the other watches over it. When the Bounding Section stops, the sections switch roles (bounding, overwatch) and the other section begins moving. The process repeats until enemy contact is made or an objective is reached.

Vehicles can cooperate with each other either *explicitly* or *implicitly*. In explicit cooperation, signals (e.g., radio messages, voice commands, hand signals) are used to synchronize actions. For example, in a platoon executing a Bounding Overwatch, the Bounding Section signals the Overwatch Section to switch roles. In implicit cooperation, vehicles cooperate by observing the actions of others. For example, in a platoon executing a Bounding Overwatch, the Overwatch Section would start moving after observing the Bounding Section stop.

Cooperative behavior of simulated entities is implemented using a *Control Architecture*. Traditional CGF systems, such as Modular Semi-Automated Forces (ModSAF), contain a Centralized Control Architecture (CCA) to control the behavior of simulated entities. In this approach, an unseen entity controls the behavior of other entities; this approach is sometimes referred to as a "puppet master" approach. CCAs are easy to implement but do not mirror cooperation of vehicles in the real world.

CCA's have several disadvantages. First, increasing the realism of the simulated entities' behaviors makes the software more complex. Second, because all behaviors are generated from a central point the controller is overworked. Third, modeling larger units, such as companies or battalions, necessitates an increase in complexity. Fourth, the entities' behavior is hardcoded into the software leading to predictable behaviors even in complex situations. Finally, CCA implementations combine the behaviors of different levels in a unit into one module; verifying and validating combined behaviors is difficult.

A second method of controlling the cooperative behavior of entities is through a Decentralized Control Architecture (DCA). In this approach entities cooperate with each other directly; there is no supervisory control. The advantages of DCA's are:

1. Unit or group cooperative behavior *emerges* as a result of direct cooperation between entities resulting in more realistic cooperative behavior in complex situations. Consequently, the software is more robust.
2. Because behavior generation is distributed across entities, which can be distributed across computers, limited hardware resources can be used efficiently.
3. DCAs give rise to *modular* implementations; for example, a Platoon Commander's cooperative behavior can be housed in a module separate from modules containing behaviors of other commanders. It is easier to *verify* and *validate* independent modules.

This report describes a DCA developed within the ModSAF CGF system. Although cooperative behaviors in ModSAF are typically implemented via centralized control, the underlying architecture supported the implemented of a DCA. The core of the DCA is a Finite State Machine (FSM) Engine. Cooperative behaviors are expressed as formal FSMs to obtain an unambiguous control process. Both explicit and implicit cooperation are supported. CGF entities can cooperate explicitly using simulated radio messages and implicitly by observing other entities.

2. Introduction

2.1 Purpose

This technical report is a deliverable under STRICOM contract N61339-92-C-0045, "Intelligent Autonomous Behavior by Semi-Automated Forces in Distributed Interactive Simulation." It satisfies CDRL A009 "Cooperative Behavior."

2.2 Background

This section provides a brief description of Distributed Interactive Simulation (DIS), Computer Generated Forces (CGF), and the Modular Semi-Automated Forces (ModSAF) CGF system. It may be skipped by readers familiar with these topics.

2.2.1 Distributive Interactive Simulation

Distributed Interactive Simulation (DIS) is an architecture for building large-scale simulation models from a set of independent simulator nodes (DIS[1993]). The simulator nodes are linked by a network and communicate via a common network protocol. (The term DIS is also sometimes used to designate a particular network protocol standard; in this document "DIS" refers to the simulation architecture; the DIS protocol standard will be so identified.) In DIS, the simulator nodes each independently simulate the activities of one or more entities in the simulated system and report their attributes and actions of interest to other simulator nodes over the network via the communication protocol. The simulated entities coexist in a common simulated environment (for example, a terrain database) and interact by exchanging network packets (Loper et. al. [1991]). Finally, an important characteristic of DIS simulations is that they are real-time; events in the simulation occur at the same rate as their real-world counterparts.

2.2.2 Computer Generated Forces

DIS environments are designed to provide a simulated battlefield which is used for training military personnel. In such a battlefield, the trainees need an opposing force against which to train. One technique is to use a computer system that generates and controls multiple simulation entities using software and possibly a human operator. This type of system is known as a Semi-Automated Force (SAF or SAFOR) or a Computer Generated Force (CGF).

A CGF system uses built-in behavior to react autonomously to the simulation situation or to carry out orders given by its operator. Its behavior may be encoded as algorithms, production rules, formal behavior specifications, or some other form. The intent is for the CGF system's behavior to be autonomous (i.e. not requiring human control) and realistic (i.e., true to doctrine, physics, and human responses) to the greatest possible extent.

2.2.2.1 The IST CGF Testbed

Under the sponsorship of the Army Research Projects Agency (ARPA) and the U. S. Army's Simulation Training and Instrumentation Command (STRICOM), the Institute for Simulation and Training (IST) has been conducting research in the area of CGF systems, seeking to increase the realism and autonomy of CGF behavior. A key product of that sponsorship is the IST CGF Testbed. The IST CGF Testbed is a CGF system that provides an environment for testing CGF behavioral control algorithms. It is documented in (Danisas et. al. [1990]), (Gonzalez et. al.[1990]), (Petty [1992]), (Smith et. al. [1992a]), and (Smith et. al. [1992b]).

Another CGF system used commonly in the research community is the Modular Semi-Automated Forces (ModSAF) system which has been used at IST since late 1994.

2.2.2.2 The Modular Semi Automated Forces CGF System

The Modular Semi Automated Forces (ModSAF) CGF system was developed by Loral Advanced Distributed Simulation for the Defense Advanced Research Projects Agency (DARPA) WISSARD (What If Simulation System for Advanced Research and Development) project and the STRICOM ADST (Advanced Distributed Simulation Technology) program. The ModSAF system is an extensible set of software modules which allows rapid development and testing of new agents (a simulation system, a simulation entity, or a simulation application) in a DIS environment (Loral [1993]) and (Loral [1994]). ModSAF's data driven execution and other features make it attractive as a testbed for testing and developing many ideas for behavior generation and command and control of automated DIS agents without extensive redevelopment of already available ModSAF supporting code.

3. Cooperative Behavior

3.1 Real World Cooperation

In a real battlefield, soldiers and vehicles (actually commanders inside the vehicles) cooperate in most, if not all, situations. They may cooperate:

- by coordinating movement and fire,
- by understanding the unit's plan and their role within it,
- by reacting to unexpected events in acceptable ways,
- through information passing, and
- by following commander's directives.

A unit in the battlefield has a hierarchy of command which reflects the information flow from the top to the bottom levels of the hierarchy. For example, a platoon has a Platoon Commander in charge of the platoon, Section Commanders in charge of individual sections (a section is typically made up of two vehicles), and Vehicle Commanders in charge of individual vehicles. In many cases an individual has several levels of responsibility; e.g., one human may be simultaneously Platoon, Section, and Vehicle Commanders.

Units also plan missions; for example, a *Bounding Overwatch*. A platoon uses this movement tactic when enemy contact is imminent. One section of the platoon moves (or bounds) while the other watches over it. When the Bounding Section stops it informs the other section, the Overwatch Section, to start moving. Now, the Overwatch Section moves and the process repeats until enemy contact is made or an objective is reached. The Platoon Commander, who is also a Section Commander, decides what each section does and communicates this information to the other Section Commander. The Section Commanders, who are also Vehicle Commanders, in turn communicate this information to Vehicle Commanders and the information flows down the hierarchy. In this way a plan is executed by breaking it down into simpler parts to be executed by lower levels in the hierarchy.

In a real battlefield, entities cooperate in a *decentralized* fashion as opposed to using a *centralized* approach. Decentralized means that entities cooperate with each other directly without being directly controlled by a supervisor. This does not mean they are unsupervised but rather the supervisor (commander) controls his subordinates through orders and signals and not through direct immediate control of the subordinate's behaviors.

Soldiers and vehicles cooperate either *explicitly* or *implicitly*. Explicit cooperation involves transmission of signals. Platoons transmit signals using: messenger, wire, visual, sound, and radio (US Army [1990]). Implicit cooperation does not involve any transmission of signals. Entities observe other entities and change their behavior accordingly; e.g., entities do formation-keeping by observing the behavior of other entities.

3.2 Background

To be effective as OPposing FORces (OPFOR) or adjunct friendly forces, CGF systems must model cooperation between entities in a way that is realistic and tactically correct. CGF systems employ a *Control Architecture* (CA) to control simulated entities. There are two CA approaches: *Centralized Control Architecture* (CCA) and *Decentralized Control Architecture* (DCA).

In a CCA, a controller (typically unseen) directs the actions of subordinate entities and makes decisions for them. Traditionally CGF systems, such as ModSAF, have used a CCA to model the cooperative behavior of vehicles. For example, a Bounding Overwatch task on a platoon is done by starting a centralized controller, the "Bounding Overwatch" task, on the platoon (Loral [1995]). This controller task divides the platoon into two groups: the Bounding Section and the Overwatch Section. The controller then plans routes to the next overwatch position for the Bounding Section and makes it move towards that position. All

formation keeping during movement is done in a centralized fashion. The controller puts the Overwatch Section in an occupy-position formation to overwatch the movement of the Bounding Section. When the Bounding Section reaches its destination, the controller switches the roles of the Bounding and Overwatch Sections and goes back to controlling the sections.

The CCA approach does not mirror real world cooperation. In the real world there is no direct control of a subordinate entity's behavior by a commander; rather, the entities control themselves in response to the commander's orders. For example, in Bounding Overwatch, real world entities compute their routes and maintain formation themselves.

In a DCA, entities follow the plan assigned to the group but control their own actions while giving orders to and receiving orders from others. This models the real world more closely. So far, no DCAs have been implemented in the CGF domain but they have been used for coordinating the actions of robots (Noreils [1993]), (Noreils [1992a]), (Noreils [1992b]), (Noreils [1992c]), (Parker[1994]), and (Shin and Epstein [1990]).

3.3 Methods for Cooperation

According to (Laird et. al [1994]), the key to coordination is knowledge. For a unit to coordinate its behavior, the individual entities must know:

1. The appropriate techniques and methods for maneuvering, sensing, and employing weapons.
2. The specific constraints under which the mission is being executed, such as rules of engagement, commit criteria, and so on.
3. During the mission, they must also build up their situational awareness, from their own sensors and through communication with others.
4. Finally, they must coordinate their actions in the face of the world around them.

During mission execution these different types of knowledge are acquired at different times using the types of methods listed below.

3.3.1 Common Doctrine and Tactics

This method addresses point 1 in Section 3.3. Common doctrine and tactics is "long-term" knowledge contained in the entity. This is similar to *social contracts*, where independent entities can create coordinated behavior by agreeing to behave in certain ways under certain circumstances (Shoham and Tennenholtz [1992]). For example, drivers in the United States coordinate their behavior (and thus avoid accidents) by always driving on the right side of a street.

Entities that are cooperating using common doctrine do not need to communicate (two cars passing each other do not need to negotiate which side they will pass). It allows an entity to predict the behavior of other entities without knowing the other entity's identity, and it reduces the cognitive load on the entity because the entity does not need to plan its behavior from first principles.

3.3.2 Mission Briefing

This method addresses point 2 in Section 3.3. Before a mission, the participants are briefed on the tactical situation, their responsibilities, and often, the responsibilities of others. The briefing helps establish specific operational parameters required for coordination, such as the specific partners of a section, their formations, the methods of communication (radio frequency, call sign), and so on.

3.3.3 Explicit Cooperation

This method addresses points 3 and 4 in Section 3.3. This is the most flexible way of cooperation between entities. It involves the transmission of signals between entities. Some ways of transmitting signals are

through radio communication and visual signals. Explicit cooperation is least susceptible to misinterpretation because a clear transmission of signals takes place.

There are many factors that prevent a unit from using explicit cooperation. For example, the possibility of radio communication being intercepted by enemy units may hinder its use. Also, obstructions, such as hills, prevent entities to communicate via visual signals.

3.3.4 Implicit Cooperation

This method addresses points 3 and 4 in Section 3.3. Implicit cooperation does not involve the transmission of signals between entities. Implicit cooperation is based on observing other entities' behavior and on modeling the behavior of other entities. An important capability of this approach is that the modeling entity can interpret not only the modeled entities' current actions but also predict the entities' future actions. (Tambe and Rosenbloom [1995]) call this *agent tracking*, where an entity monitors the observable actions of other entities as well as their unobserved actions or high-level plans, goals, and behaviors, and adjusts its behavior accordingly.

3.4 Statement of the problem

The goal of the research described in this report is to implement a CA architecture that:

- mirrors real life cooperation between vehicles,
- uses explicit and implicit cooperation between vehicles,
- allows new cooperative behaviors to be created easily and with little coding, and
- can be verified and validated easily.

4. Cooperative Behavior Control Architectures

The cooperative behavior control architecture controls the behavior of subordinate entities. There are two ways to control subordinate entities: Centralized control and Decentralized control.

4.1 Centralized Control Architectures (CCA)

In the real world, a commander controls a unit by giving orders. Subordinate entities act on these orders and change their behaviors accordingly. In a Centralized Control Architecture (CCA), a centralized controller makes behavioral decisions for subordinate entities and conveys these decisions to the subordinates. CCAs resemble the real world because, like the real world, the unit is controlled from a centralized location. However, there are important distinctions. The first distinction is in the granularity of control relative to that of the real world. CCAs exercise unrealistically fine control. For example, CCAs may do formation-keeping for a platoon by monitoring each entity and making sure that entities maintain appropriate distances between them. In the real world, formation-keeping is done by entities; proper entity-to-entity distances are computed and maintained by entities themselves. The second distinction is in reasoning and decision making. In CCAs, the centralized controller reasons and makes decisions on the entities' behalf whereas in the real world entities reason and make decisions themselves. For example, CCAs plan routes for the entities whereas real world entities plan their own routes.

The entity exercising centralized control may be either a simulated entity or an unsimulated "ghost" entity. Furthermore, the centralized controller may control subordinate entities either *explicitly* or *implicitly*. Explicit control requires the transmission of messages from the centralized controller to the subordinates; these messages are often orders to the subordinates. These orders, unlike real world orders, contain specific information which otherwise would have been computed by the subordinates themselves. For example, an order to move may *contain* the route information. In the real world, a subordinate will only be told to move to a destination and it will compute the route itself. Implicit control is more direct. In this case, the centralized controller executes code on or on behalf of subordinate entities. Code execution directly affects a subordinate's behavior.

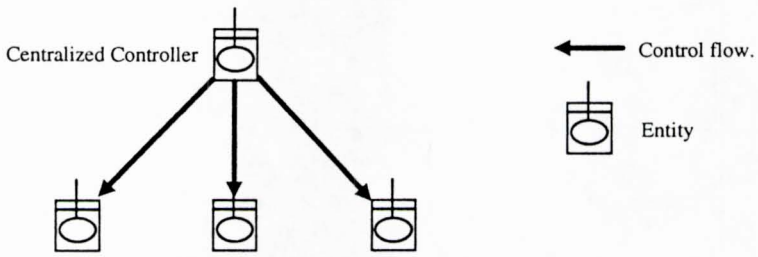


Figure 1: Centralized Control Architecture (simulated entity controller).

Figure 1 shows a CCA in which the centralized controller is a simulated entity. The arrows indicate control flow from the centralized controller to the subordinates. This may be either explicit (messages are sent) or implicit (code is executed by the centralized controller on or on behalf of the subordinates).

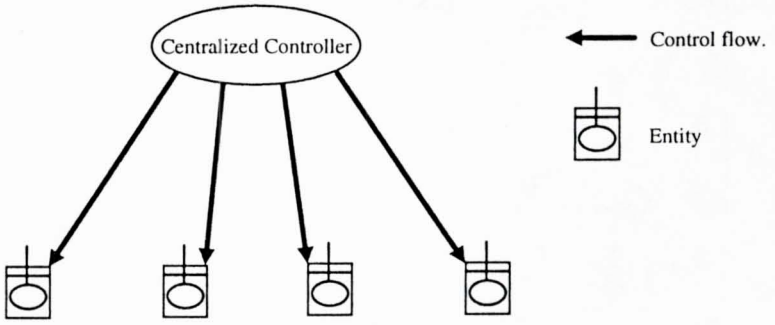


Figure 2: Centralized Control Architecture (unsimulated entity controller).

Figure 2 shows a CCA where the centralized controller is an unsimulated entity (code). As in Figure 1, the arrows indicate control flow from the centralized controller to the subordinates. Again, the control may be either explicit (messages are sent) or implicit (code is executed by the centralized controller on or on behalf of subordinates).

ModSAF implements cooperative behavior by combining the two approaches. The centralized controller in ModSAF is an unsimulated entity which "knows" the identity of the vehicle responsible for the unit, e.g., a Platoon Commander. When the Platoon Commander is disabled, ModSAF restarts the cooperative behavior on the platoon. The responsible entity is updated, i.e., another entity becomes the Platoon Commander.

The centralized ModSAF controller controls the subordinates implicitly by executing code on their behalf. For example, the centralized controller in ModSAF executes a platoon Bounding Overwatch by dividing the platoon into sections, each containing two vehicles. It then computes a destination and makes one section (Bounding Section) move to it while it puts the other section (the Overwatch Section) in an occupy-position formation. The centralized controller monitors the Bounding Section's location. When it reaches the destination, the centralized controller switches the roles of the two sections and the process repeats.

CCAs are suitable for implementing simple cooperative behaviors but have several disadvantages. First, implementing a CCA results in loss of realism. For example, with a "ghost" centralized controller, the unit's collective behavior can be unaffected by the loss of the simulated commander. On the other hand, if a simulated centralized controller is destroyed, the collective behavior of the unit is disrupted. Of course, both problems can be addressed by introducing provisions in the software for transfer of command. But the complexity required to centrally resolve all the conflicts between centrally controlling a real world decentralized control process forces compromises and simplifications. To make up for these losses would entail increasing the complexity of the software. Second, generating the behaviors of all entities from a single source results in inefficient use of resources; more time is spent in the controller causing it to be overworked. Finally, modeling larger units, such as companies or battalions, becomes increasingly complex because the centralized controller has to control more vehicles.

4.2 Decentralized Control Architectures (DCA)

In a Decentralized Control Architecture (DCA), subordinate entities follow the unit's plan and commander's orders but make their own behavior decisions. Unlike a CCA, there is no unseen controller that makes decisions on their behalf; this approach mirrors cooperation in the real world. A DCA commander functions like a real world commander by giving and receiving orders from other entities. For example, a DCA commander may order an entity to move to a destination. Like the real world, the commander may only supply the entity with the location of the destination and not a precise route. In this case, the entity computes its route to reach the destination.

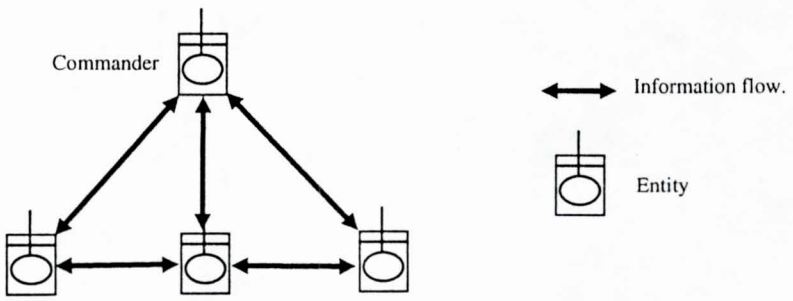


Figure 3: Decentralized Control Architecture.

Figure 3 shows a DCA. Notice that in contrast to a CCA (Figure 1 and 2), entities send information (shown by the double-headed arrows) to each other. Each entity contains knowledge to process incoming information and modify its behavior.

DCA's have several advantages. First, unit or group cooperative behavior *emerges* as a result of direct cooperation between entities resulting in more realistic cooperative behavior in complex situations. Second, because behavior generation is distributed across entities, which can be distributed across computers, limited hardware resources can be used efficiently. Finally, DCAs give rise to *modular* implementations; e.g., a Platoon Commander's cooperative behavior can be housed in a module separate from modules containing behaviors of other commanders. Behaviors can be verified and validated independently of independent modules. CCA implementations combine the behaviors of different levels in a unit into one module making verification and validation more difficult.

When discussing DCAs two questions need to be answered. First, how do entities cooperate with each other. Second, how do entities know what task to do and when to do it. These two topics are discussed in Section 4.2.1 and 4.2.2 respectively.

4.2.1 Entity cooperation in DCAs

Entities can cooperate in a number of ways:

- Message Passing.
- Shared Memory.
- Combination of Message Passing and Shared Memory.
- Implicit Cooperation.

Entities can cooperate explicitly (Section 3.3.3) by passing messages, such as radio messages, to each other (Noreils [1993]), (Noreils [1992a]), (Noreils [1992b]), (Noreils [1992c]), (Parker [1994]), (Shin and Epstein [1990]), (Lefebvre and Saridis [1992]), (Smith and Davis [1981]), (Fisher and Woodridge [1994]), (Decker [1987]), and (Ohko et. al. [1993]). Messages may be *broadcast* (Parker [1994]) to all entities or sent *point-to-point*.

Entities can also cooperate explicitly by placing information into memory which is shared by other entities. This shared memory is commonly called a *Blackboard*. Blackboard and its variants have been used in a number of DCAs (Laengle and Lueth [1994a]), (Laengle and Lueth [1994b]), (Corkill [1991]), (Occello and Demazeau [1994]), and (Dai et. al. [1993]).

Some DCAs implement a combination of the message passing approach and the Blackboard to allow entities to cooperate (Lun and Macleod [1992]), (Wang [1994]), and (Harmon et. al. [1986]).

In some DCAs entities can cooperate with each other implicitly (Section 3.3.4) by observing what the others are doing (Payton and Dolan [1991]).

4.2.2 Allocation of subtasks in DCAs

In DCAs, entities contain knowledge for executing tasks allocated to them. Before they begin using this knowledge they must be told what task they need to work on. The processing of letting entities know their tasks is called *task allocation*. There are two ways to allocate tasks in DCAs: Negotiation and Self-Contribution.

When tasks are allocated using Negotiation, (Noreils [1993]), (Noreils [1992a]), (Noreils [1992b]), (Noreils [1992c]), (Lun and Macleod [1992]), (Smith and Davis [1981]), (Fisher and Woodridge [1994]), (Decker [1987]), and (Ohko et. al. [1993]), one entity assumes the role of a *mediator* or *manager*. The manager subdivides the mission into tasks, advertises for the tasks, and receives *bids* from prospective *contractors* (entities who are able and willing to do the task). The manager selects the most appropriate contractor and awards the *contract*. A contractor receives task announcements from different managers and selects the one that best fits the skills/knowledge that it has. After the contract has been awarded, the manager and the contractor become linked by the contract and communicate privately the progress of the task being executed.

In contrast to being told what tasks are available for execution, entities use the Self-Contribution approach when they want to start executing tasks themselves. There are two approaches. In one approach (Corkill [1991]), each entity knows the condition under which it can execute the task and, at appropriate times, it attempts to do so. In the other approach (Parker [1994]), an entity executes a task when its motivation to do the task, measured by a *motivational behavior function*, exceeds a certain threshold.

4.2.3 Survey of DCAs

There is a variety of DCAs in the literature. Sections 4.2.3.1 through 4.2.3.4 is a survey of these DCAs.

4.2.3.1 Net-based DCAs

Net-based DCAs have been used commonly for implementing cooperative behavior between robots. Net-based DCAs encode the cooperative behavior of robots in Petri Nets (Peterson [1981]) or Petri Net modifications such as Predicate/Transition nets (Noreils [1993]), (Noreils [1992a]), (Noreils [1992b]), (Noreils [1992c]), (Lefebvre and Saridis [1992]), (Wang et. al. [1992]), (Zhou et. al. [1994]), and (Bachatene and Seghrouchni [1993]).

Petri Nets (PN) are tools for modeling the dynamic behavior of discrete event systems. Ordinary PN are directed graphs with two types of nodes called *places* and *transitions*, which are connected by *arcs* (Peterson [1981]). Places may contain *tokens* that indicate the state of the PN. A place is referred to as *marked* if there is at least one token in it. A transition is *sensitive* if all input places are marked. A transition is *fired* if it is sensitive. Transitions are *atomic*, meaning that if a transition is fired and code is executed as part of the transition, it remains fired until code execution is complete. Tokens are moved between places by the firing of a transition. PN are valuable for simulating concurrent systems because the PN structures can be analyzed for desirable properties such as boundedness and deadlock-free operation (Murata [1989]).

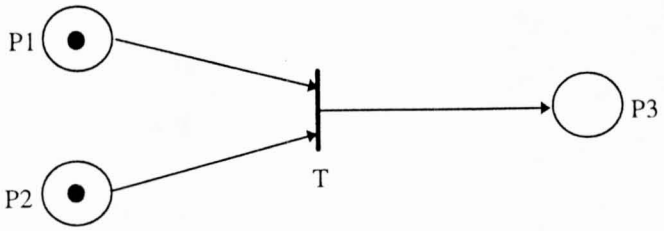


Figure 4: A Petri Net (all input places are marked).

Figure 4 shows a PN. The places are shown by circles P1, P2, and P3, a transition is shown by the vertical line T, and arrows denote arcs. Places P1 and P2 contain tokens, shown by dots, indicating that they are marked. Because all the input places for transition T are marked, the transition is sensitive and fires. The firing of a transition causes code to be executed.

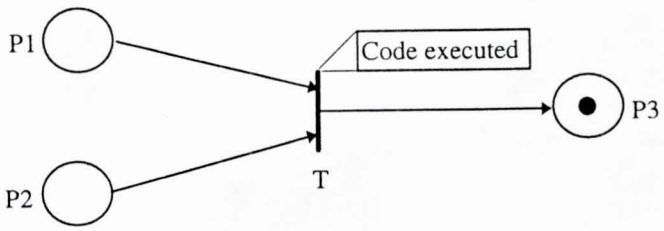


Figure 5: A Petri Net (a transition has fired).

Figure 5 shows the state of the PN after transition T has fired. Places P1 and P2 are *unmarked* and place P3 is marked. The code associated with transition T, shown symbolically in the box, is executed when T fires.

The cooperative behavior of a robot is described by a PN. When a robot "executes" its PN, it may wait for *internal events* or *messages* from other robots. When these arrive, certain places become marked resulting in transitions being fired. Code associated with the transition is executed; a robot uses this code to modify its behavior in response to internal events or messages from others. In this way, a robot executes a cooperative behavior by modifying its behavior in response to external or internal stimuli.

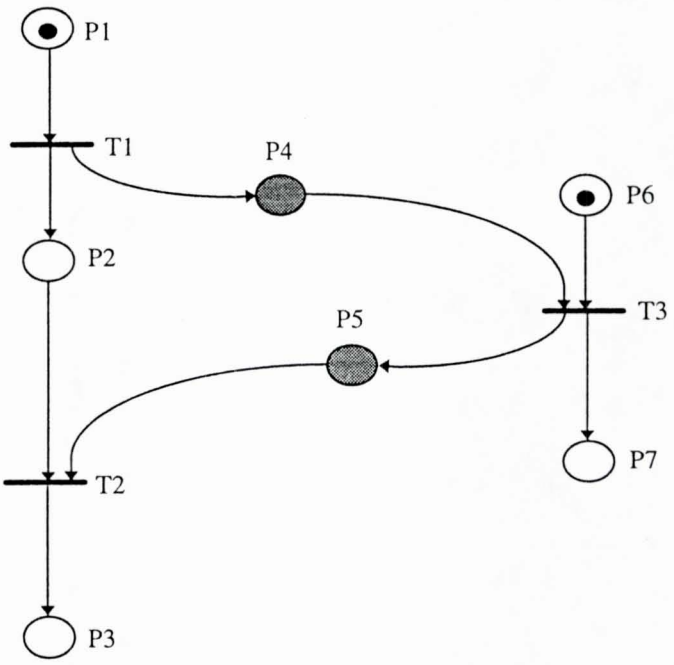


Figure 6: Two robots cooperating using a coordinated protocol.

Figure 6 shows two robots cooperating. The figure is used to illustrate how robots cooperate and is not intended to show a specific cooperative behavior. Places P4 and P5 are *shared* by the two robots. These places are associated with messages and become marked when messages arrive. For example, P4 becomes marked when a message, sent by Robot 1, is received by Robot 2.

Initially, place P1 and P6 are marked and T1 is sensitive. T1 fires, P2 gets marked, and a message is sent. The receipt of the message by Robot 2 causes P4 to be marked. T3 becomes sensitive and fires, P7 becomes marked, and a message is sent. The receipt of the message by Robot 1 causes P5 to be marked. T2 becomes sensitive and fires. This mechanism shows how sharing places between the robots' PNs can help them cooperate with each other.

An advantage of Net-based DCAs is that complex behaviors can be modeled by combining simple nets; each net models a simple behavior. In this way it is possible to build a library of behaviors and model complex behaviors by combining behaviors contained in the library.

4.2.3.2 Blackboard DCAs

The Blackboard (BB) (Corkill [1991]) is the oldest data structure that has been used for modeling cooperative problem solving. A BB system consists of three components (Corkill [1991]):

- Knowledge Sources (KSs),
- the Blackboard, and
- a Control Component (Figure 7).

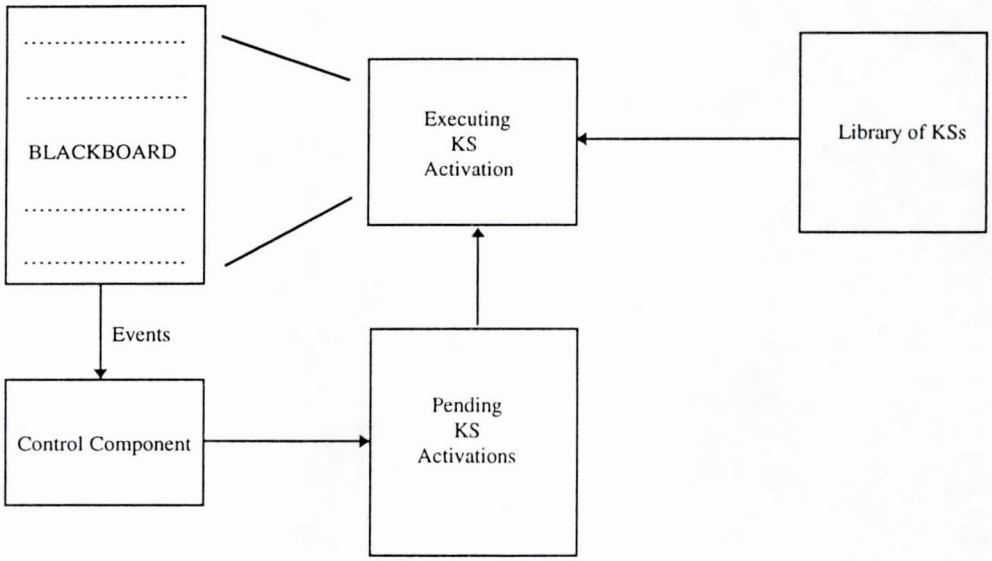


Figure 7: Basic Blackboard System.

Knowledge sources are independent modules that contain the knowledge needed to solve a problem. KSs can be widely diverse in representation and inference techniques. Each KS is separate and independent of all other KSs. A KS needs no knowledge of the expertise, or even the existence, of the others; however, it must be able to understand the state of the problem-solving process and the representation of relevant information on the BB. Each KS knows the condition under which it can contribute to the solution and, at appropriate times, it attempts to contribute information toward solving the problem (Self-Contribution, Section 4.2.2). When this happens, KSs are triggered and KS *activations* (or *instances*) are created. KS activations are active processes that compete for execution resources.

The BB is a global database containing input data, partial solutions, and other data that are in various problem-solving states. The BB serves as:

- A community memory of raw input data; partial solutions, alternatives, and final solutions; and control information,
- A communication medium and buffer, and
- A KS trigger mechanism.

The Control Component makes runtime decisions about the course of problem solving and the expenditure of problem-solving resources. The control component is separate from the individual KSs.

The BB system uses an incremental reasoning style; the solution to the problem is built one step at a time. At each step, the system can:

- Execute any triggered KS.
- Choose a different focus of attention on the basis of the state of the solution.

Under a typical control approach, the currently executing KS activation generates events as it makes contributions to the BB. These events are maintained (and possibly ranked) until the executing KS activation is completed. At that point, the Control Component uses the events to trigger and activate other

KSs. The KS activations are ranked, and the most appropriate KS activation is selected for execution. This cycle continues until the problem is solved.

Other variations of DCAs using BBs are discussed in (Lun and Macleod [1992]), (Occello and Demazeau [1994]), (Wang [1994]), and (Dai et. al. [1993]).

BB DCAs are suitable when:

1. Many diverse, specialized knowledge representations are needed. KSs can be developed in the most appropriate representation for the data they are to handle. For example, one KS might be most naturally written as a rule-based system while another might be written as a neural-net or fuzzy-logic routine.
2. An integration framework is needed that allows for heterogeneous problem-solving representations and expertise. For example, a BB is an excellent framework for combining several separately established diagnostic systems.
3. The development of an application involves numerous developers. The modularity and independence provided by large-grained KSs in BB systems allow each KS to be developed and tested separately. The software engineering benefits of this approach apply during design, implementation, testing, and maintenance of the application.
4. Uncertain knowledge or limited data inhibits absolute determination of a solution. The incremental approach of the BB system will still allow progress to be made.
5. Multilevel reasoning or flexible, dynamic control of problem-solving activities is required in an application.

BB DCAs are not very common:

1. The advantages of BB systems do not scale down to simple problems; they are only worth pursuing for complex applications.
2. A BB system is useful for prototyping an application, but, once developed and understood, the application can be reimplemented without the BB structure.

4.2.3.3 Ad-hoc DCAs

Researchers in robotics have developed a variety of DCAs for coordinating the actions of a group of robots. These architectures have been developed as build-to-suit approaches.

(Yuta et. al. [1992]) and (Taipale and Hirai [1993]) discuss a combination of centralized and decentralized control. In this approach, when a group of robots are given a task, one becomes a *leader*, solves the problem, and conveys the result to the other robots. This approach is also called a *master-slave* control architecture because one robot, the master, controls the others, the slaves.

Some researchers have looked at the level of cooperative control. Entities can be controlled in a purely top-down *plan-based* manner and a purely bottom-up *behavior-based* manner; these two levels represent two extremes of cooperative control. In plan-based approaches, all entity actions and interactions are planned in advance. The plan is a script for action that all cooperating entities must follow faithfully. Assuming that the entities can each execute their plan correctly, then an overall coherent activity will take place. In behavior-based approaches, group behavior *emerges* from local entity interactions. By giving each entity the same set of procedures for how to behave in response to the actions of others, a variety of interesting and useful group behaviors can emerge (Arai et. al. [1989]). The work done by (Payton and Dolan [1991]) seeks to establish a bridge between the plan-based approach and the behavior-based approach. Plans are used as resources for action (Suchman [1987]). As resources, plans serve as sources of information and advice to entities that are already competent at dealing with the immediate concerns of their environment. Consequently, plans are used to bias the natural actions of entities so that they conform more closely to achieving some global objective.

Some other ad-hoc architectures are described in (Laengle and Lueth [1994a]), (Laengle and Lueth [1994b]), (Parker [1994]), and (Shin and Epstein [1990]).

4.2.3.4 Finite State Machines (FSMs)

FSMs have been used widely in CGF systems. The IST CGF Testbed (Smith et. al. [1992a]), ModSAF, and the CCTT SAF (Petty [1995]) use FSMs to implement cooperate behavior. FSMs are attractive because they are a well understood process control mechanism and have been defined formally in Computer Science (Sudkamp [1988]).

A formal FSM is defined as:

1. A set of *states*: An FSM is in one of its states. The state of an FSM is also the state of the process being controlled by the FSM.
2. *Events*: *Only* events cause an FSM to change states.
3. *State Transition Procedures (STPs)*: These are procedures (i.e., code) which are called to do work. STPs are used *only* during state transitions. Many informal FSM implementations associate code with states so that code is executed when an FSM goes to a new state and not *during* the transition.

5. A Decentralized Control Architecture using Finite State Machines

In Section 4.2.3 different DCA's were discussed. IST chose to implement the FSM-based DCA. Section 5.1 describes the approach taken by IST. Section 5.2 describes formal FSMs; it may be skipped by readers who are familiar with the topic. To cooperate, entities need to communicate. They do this via FSM communication which is described in Section 5.3. Section 5.4 describes the FSM Engine which is a mechanism for implementing formal FSMs.

5.1 Approach

To model cooperative behavior IST chose to implement a DCA within the ModSAF CGF system. DCAs have several advantages which are described in Section 4.2. Traditional CGF systems have used CCAs for controlling cooperative behavior. The work described in this report is the first time a DCA has been implemented within a CGF system for controlling cooperative behavior.

The DCA chosen is based on Finite State Machines (FSMs). FSMs were used as building blocks for the architecture; FSMs are a well understood process control mechanism and are defined formally in Computer Science (Sudkamp [1988]).

Entities cooperate explicitly by exchanging simulated radio messages (Signal PDUs) and implicitly by observing other entities. Observation is implemented by having an Observation Module, attached to an entity, send observation messages to the entity.

The implementation is data driven and allows new behaviors to be defined quickly and easily through data files. In current CGF systems, considerable coding effort is required to create new cooperative behaviors. This increases development and prototyping time for new cooperative behaviors.

5.2 Formal FSMs

FSMs have been defined formally in Computer Science (Sudkamp [1988]) but implemented to various degrees of formality. FSMs are one of the best techniques for process control. FSMs, as their name implies, track the *state* of a process, handling *events* which may cause the process to change state (Section 4.2.3.4).

Formal FSMs are often represented as diagrams. Consider the example of a coin-operated candy dispenser. whose FSM is shown in Figure 8.

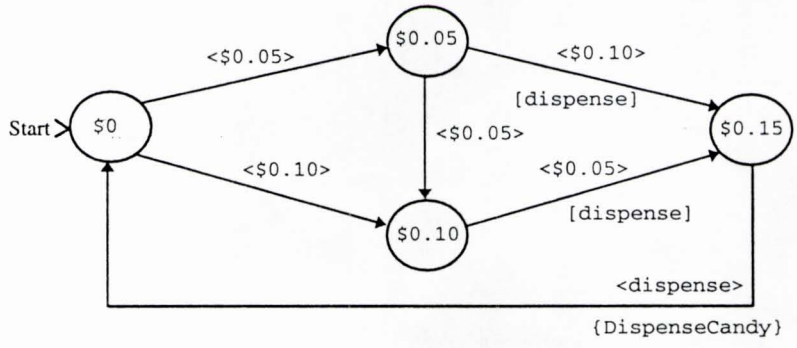


Figure 8: FSM for a coin-operated candy dispenser.

This machine accepts only nickels and dimes and dispenses candy worth $0.15. In Figure 8, the circles represent states and arrows represent state transitions. Above the state transition line, in angle brackets

(< >), is the event causing the transition. Square brackets ([ ]) represent any events that are generated as part of the transition. Braces ({ }) represent calls to STPs.

When a person walks up to the machine it is in state "$0" because no money has been deposited so far. Depending on whether a nickel or dime is deposited, the machine transitions to the "$0.05" or "$0.10" state. Assuming that a nickel was deposited, the state of the machine is "$0.05." Then, if a dime is deposited two things happen. First, an event (dispense) is generated, and second, the machine goes to the state "$0.15." When the dispense event arrives, an STP, DispenseCandy, is called to dispense the candy. Note that an STP does work as part of the transition back to the start state, "$0"; no code is executed in the state "$0.15" to dispense candy.

This example shows the workings of a simple formal FSM. A formal FSM does not poll for events to change states; rather, events are generated and their arrival causes state transitions. This feature of formal FSMs is especially attractive because it eliminates inefficiencies introduced in polling.

IST considered implementing the DCA using ModSAF FSMs. In ModSAF, FSMs are not implemented formally: code is executed within states rather than by STPs during state transitions, and many state transitions are not event driven.

Because formal FSMs provide an unambiguous way to control a process, they were used for the implementation. To reiterate, in formal FSMs:

1. only events cause state transitions,
2. states do not contain code; they are merely place-holders for the process' current state, and
3. code is only executed by State Transition Procedures (STPs) during state transitions.

5.3 FSM communication

An entity's cooperative behavior is implemented as an FSM. To cooperate, entities need to communicate and they do so via FSM communication. FSMs may communicate with each other (inter-FSM communication) or an FSM may communicate with itself (intra-FSM communication).

5.3.1 Inter-FSM communication

In inter-FSM communication, FSMs send events to each other. These events, called *external* events, often take the form of simulated "Radio Messages." For example, when the Bounding Section, in a platoon executing a Bounding Overwatch, reaches a destination, the Section Commander sends a radio message announcing the completion of its bound.

An entity may generate external events to itself. These events are generated from an Observation Module and are called Observation Events. Observation Events are generated in response to battlefield conditions. For example, the Observation Module for a Wingman sends an Observation Event informing him when his Section Commander starts moving. This observation event triggers the Wingman's FSM to start following the Section Commander.

5.3.2 Intra-FSM communication

FSMs communicate with themselves by sending *internal* events to themselves. Consider the FSM for the coin-operated candy dispenser shown in Figure 8. Assume that the FSM is in state "$0.05." When a dime is deposited, the machine generates an internal event, dispense, to itself and transitions to the state "$0.15." The receipt of the dispense internal event signals the FSM to transition to another state and execute an STP (DispenseCandy).

5.3.3 Event Queues

FSMs communicate by generating external events between themselves. When external events arrive, they are first mapped to internal events and then put into an event queue for processing. There are two approaches for handling external and internal events: use one or two event queues.

Using a common queue for external and internal events can lead to synchronization problems. State machine actions are non-preemptive; processing an internal event is done completely and, possibly, new internal events are generated in one execution thread. While internal events are being processed new external events may continue to arrive. If the external and internal events are processed in an interleaved manner unexpected situations can develop. Handling all possible interleaving of internal and external events is needlessly complicated.

The solution is to queue external and internal events in separate queues. External events are put into one or more *external event queues* while internal events are put into *an internal event queue*. No external event is dispatched until the internal event queue is empty. This allows all intra-machine communication (spawned by an external event) to complete without interference from new external events. The approach also allows a single external event to be re-mapped into several internal events. This reduces machine complexity and breaks complex external events into simpler requests.

5.4 FSM Engine

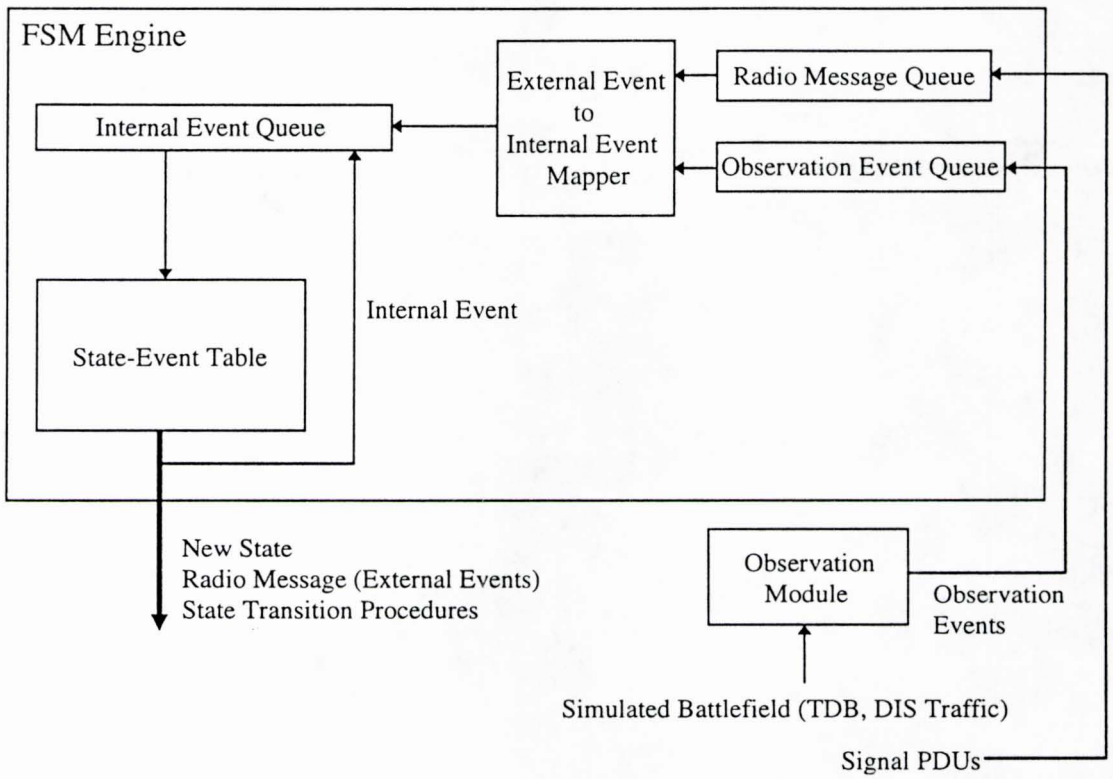


Figure 9: FSM Engine.

Because formal FSMs do not exist in ModSAF, an FSM Engine (Figure 9) was developed to run formal FSMs. The FSM Engine contains an FSM's description in a State-Event Table. The State-Event Table is created by reading a data file FSM description (Section 6.3). The table is indexed by a state/event pair that determines the new state of the FSM; the indices are the current state of the FSM and the internal event to be processed. During the transition, external and internal events may be generated and STPs called.

The FSM Engine receives input from two sources: Signal PDUs and Observation Events. Signal PDUs contain radio messages and simulate radio communication. Observation Events are generated by the Observation Module in response to battlefield situations. Radio messages and Observation Events are external events which are queued on two separate queues: Radio Message Queue and Observation Event Queue.

Periodically, the external event queues are checked to see if any external event is waiting to be processed. The external event is removed from the queue, *mapped* into an internal event, and queued on the internal event queue. Then, internal events from the internal event queue are removed and processed. To process events, the FSM Engine needs to be called periodically. This is done by calling the FSM Engine from a non-transitioning ModSAF FSM. Each time the ModSAF FSM becomes active, it calls the FSM Engine; the FSM Engine can be thought of as embedded within the ModSAF FSM. Note that the ModSAF FSM does not do anything. Its sole purpose is to ensure that the FSM Engine is called periodically; all the work required in processing events and changing the behaviors of entities is done by the FSM Engine.

6. Implementation

This section describes implementation details. Cooperative behavior occurs at different levels in a unit. To mimic the flow of information in a real unit a hierarchy of commanders is created (Section 6.1). Section 6.2 describes Bounding Overwatch, a cooperative behavior commonly used by platoons. The sub-sections describe the FSMs for the Platoon, Section, and Vehicle Commanders. Section 6.3 discusses the specification of cooperative behavior in data files.

6.1 Hierarchy of Commanders

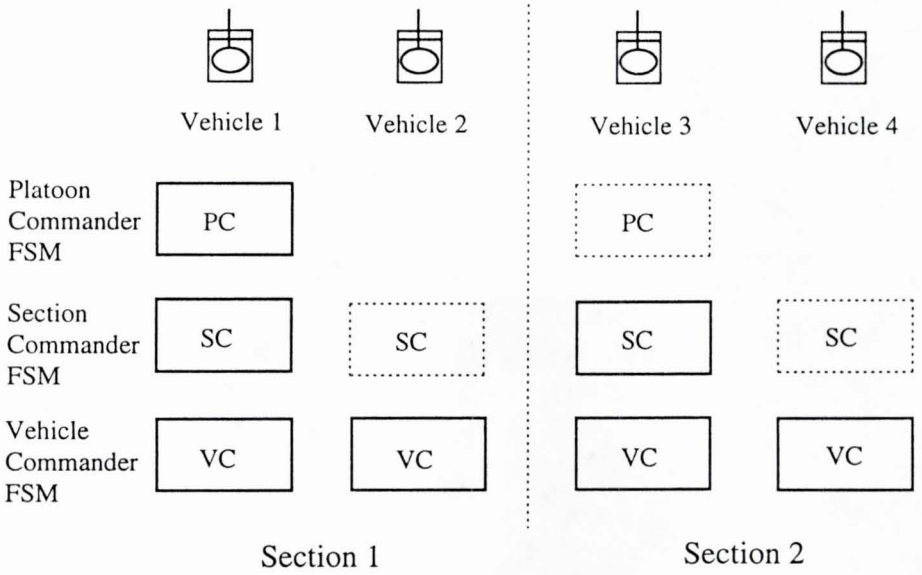


Figure 10: Hierarchy of commanders.

A vehicle can execute behaviors on many levels. Consider Vehicle 1 in Figure 10. The commander of this vehicle has three responsibilities, those of the Platoon Commander (PC), Section Commander (SC), and Vehicle Commander (VC). One way to represent the cooperative behavior of this commander would be to create a large and complex FSM that merges the platoon, section, and vehicle commander behaviors. This process can become arbitrarily complex as the hierarchy grows and commanders with more responsibilities are modeled. For example, for the hierarchy shown in Figure 10, Platoon-Section-Vehicle Commander, Section-Vehicle Commander, and Vehicle Commander FSMs would be needed to encapsulate all classes of cooperative behaviors.

Instead, IST established a hierarchy of commanders like the one shown in Figure 10. Each box in the figure represents a ModSAF FSM. Embedded inside each ModSAF FSM is the FSM Engine (Section 5.4). This approach allows complex behaviors to be split into fundamental behaviors that are implemented as separate FSMs; complex FSMs containing merged behaviors are thus avoided. For example, Vehicle 1 (Figure 10) has three FSMs (Platoon, Section, and Vehicle Commander FSMs) controlling its behavior. Each FSM communicates with others. The command hierarchy is created by higher level commander FSMs *spawning* lower level commander FSMs; for example, the Platoon Commander FSM spawns the Section Commander FSMs which in turn spawn Vehicle Commander FSMs.

In addition, there are next in command (*deputy*) commanders, shown by dotted boxes in Figure 10. Deputy commanders assume command when the original commanders are disabled so that the unit's mission can continue unhindered (Section 6.4). They model the behavior of the original commander but do not communicate with other entities. This allows them to continuously track the original commanders behavior and assume command in case the original commander is disabled. In Figure 10, Vehicle 3 is a deputy

Platoon Commander (i.e., Platoon Sergeant), and Vehicles 2 and 4 are deputy Section Commanders for Vehicles 1 and 3 respectively.

To start the whole process, a user assigns a mission to the unit. As part of initialization, a data structure known as a **Role Matrix** is created. The Role Matrix is a two dimensional array of vehicle IDs and roles such as Platoon Commander, Section Commander, Vehicle Commander, and deputy commanders.

Vehicle ID	1	2	3	4
Platoon Commander	1	0	0	0
Platoon Commander deputy	0	0	1	0
Section Commander	1	0	1	0
Section Commander deputy	0	1	0	1
Vehicle Commander	1	1	1	1

Figure 11: Simplified Role Matrix.

Figure 11 shows a simplified Role Matrix for the commander hierarchy in Figure 10. The vehicles in the unit have IDs from 1 to 4. A "1" in a cell at the intersection of a vehicle ID column and role row means the vehicle is playing that role; for example, Vehicle 1 is the Platoon Commander, Section Commander, and Vehicle Commander. A "0" in a cell at the intersection of a vehicle ID column and role row means that the vehicle is not playing that role; for example, Vehicle 3 is not the Platoon Commander. Note that Vehicle 3 is a deputy Platoon Commander and Vehicles 2 and 4 are deputy Section Commanders. Because a Vehicle Commander's responsibility is limited to his vehicle's domain and another Vehicle Commander cannot assume his functions, there are no deputy Vehicle Commanders. This is represented by the absence of a deputy Vehicle Commanders row in the Role Matrix.

A vehicle's Role Matrix is accessible from the vehicle's FSMs. Using the Role Matrix a vehicle can easily determine the role of other vehicles. For example, Vehicle 2 knows that Vehicle 1 is a Platoon and Section Commander. In a real battlefield, a vehicle is designated roles before an exercise begins. The Role Matrix is a manifestation of this information in the computer.

6.2 Bounding Overwatch

The FSM architecture was tested on a platoon executing a Bounding Overwatch. Bounding Overwatch provides a simple and elegant way to test the architecture. In this behavior, a platoon advances by having its sections alternately move and overwatch the movement of the other section. The sections move until the platoon reaches an objective or enemy contact is made. By moving in this fashion, the platoon reduces the risk of being ambushed by enemy forces.

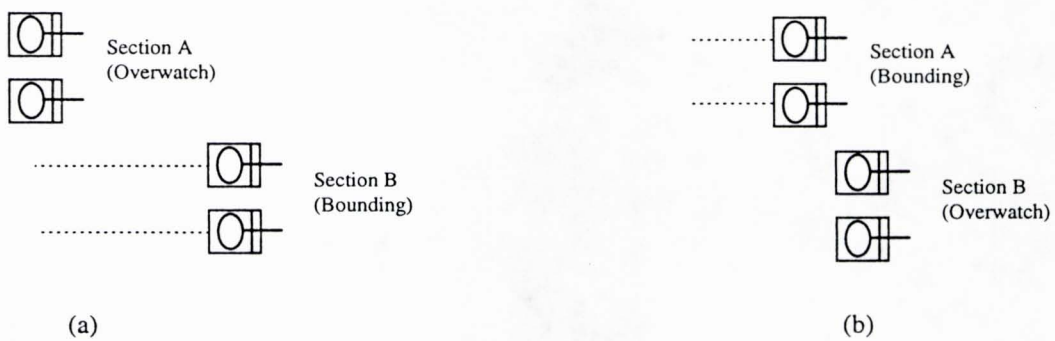


Figure 12: Bounding Overwatch.

The sections are in *bounding* or *overwatch* roles before the orders to start Bounding Overwatch are given. The Bounding Section moves towards an intermediate destination always maintaining line of sight with the Overwatch section, which is covering its movement (Figure 12(a)). Once the Bounding Section reaches an intermediate destination, the Section Commander sends a radio message that the section is ready to overwatch the movement of the other section. Alternatively, the Overwatch Section may observe that the Bounding Section has stopped. Section Commanders use radio to communicate when enemy contact is not imminent while relying on observation if the chance of running into an enemy is high.

In any case, there is a reversal of roles when the Bounding Section stops. The Overwatch Section switches roles with the Bounding Section and overwatches its movement (Figure 12(b)). This process continues until the objective is reached or enemy contact is made.

Sections 6.2.1 through 6.2.3 show FSMs for the Platoon, Section, and Vehicle Commanders for Bounding Overwatch. These FSMs communicate via radio messages (explicit cooperation). Note that in the following discussion overwatch is also called "Cover."

6.2.1 Platoon Commander FSM

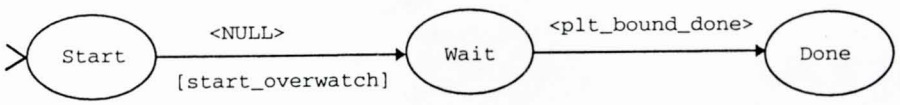


Figure 13: Bounding Overwatch Platoon Commander FSM.

To start the process, the Platoon Commander FSM sends a radio message, `[start_overwatch]`, and transitions to the `Wait` state (Figure 13). It then stays there until it is informed (via event `<plt_bound_done>`) the platoon is at the objective, when it goes to the `Done` state.

6.2.2 *Section Commander FSM*

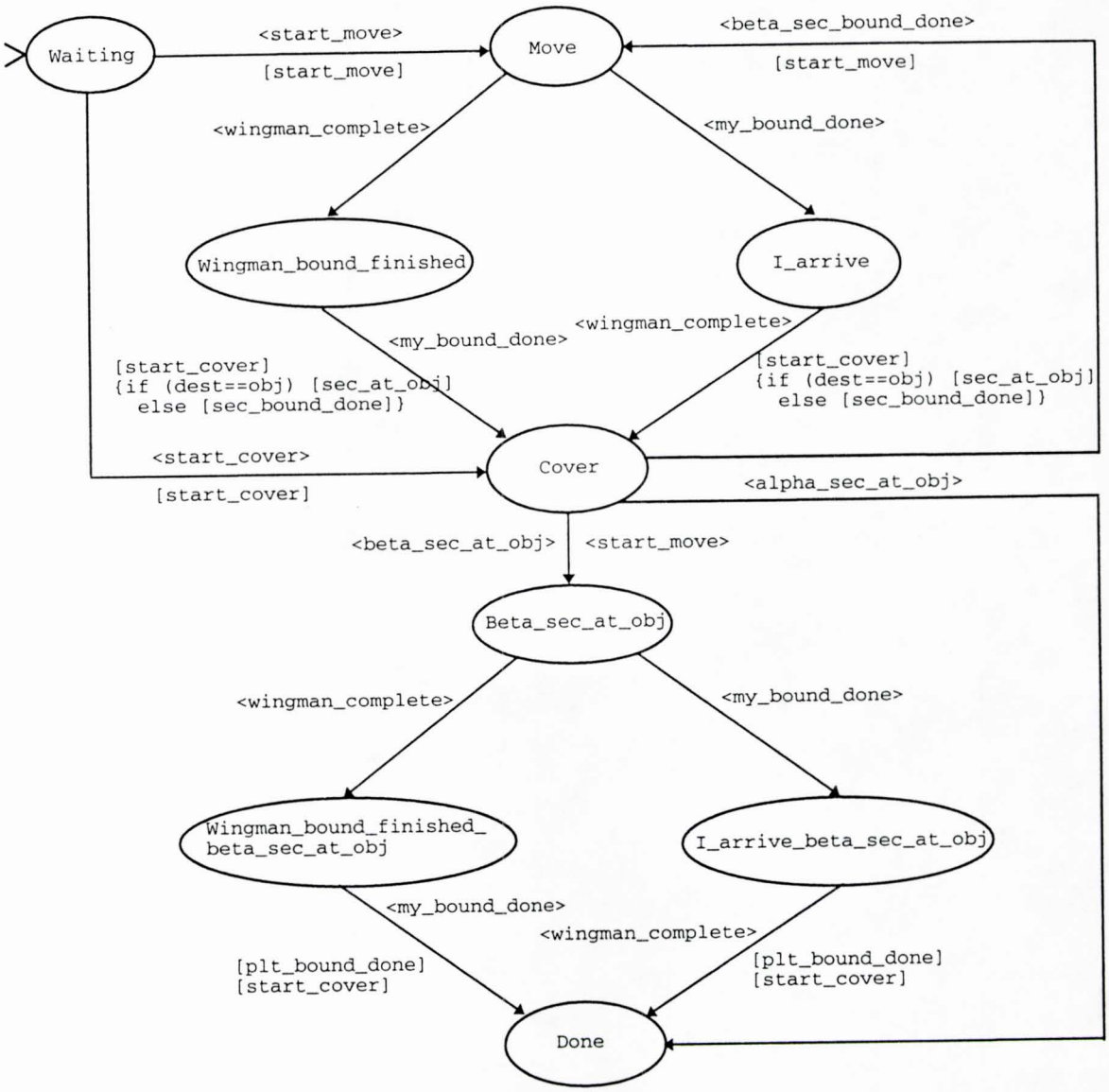


Figure 14: Bounding Overwatch Section Commander FSM.

Figure 14 shows the Section Commander FSM for Bounding Overwatch. Initially the FSM is in the `Waiting` state. When a Section Commander receives the order to move (via event `<start_move>`), it sends a radio message, `[start_move]`, to its Vehicle Commanders and transitions to the `Move` state. In this state the section moves toward an intermediate destination. When a Section Commander receives the order to cover (via event `<start_cover>`), it sends a radio message, `[start_cover]`, to its Vehicle Commanders and transitions to the `Cover` state where it overwatches the moving section.

The moving section may arrive at the intermediate destination in two ways. Either the Section Commander arrives first (event `<my_bound_done>` arrives) followed by the Wingman (event `<wingman_complete>` arrives) or vice versa. If the Section Commander arrives first, the FSM transitions to the `I_arrive` state. When the Wingman arrives (event `<wingman_complete>` arrives) the FSM transitions to the `Cover` state and as part of the transition does this: First, the Section Commander

issues a radio message to its Vehicle Commanders to start cover and second, checks if the intermediate destination is the objective. If the section is at the objective it sends a radio message, `[sec_at_obj]` (section at objective), otherwise the section is at an intermediate destination and a radio message, `[sec_bound_done]` (section bound done), is sent. The section now overwatches the movement of the other section, which has transitioned from overwatch to move.

6.2.3 Vehicle Commander FSM

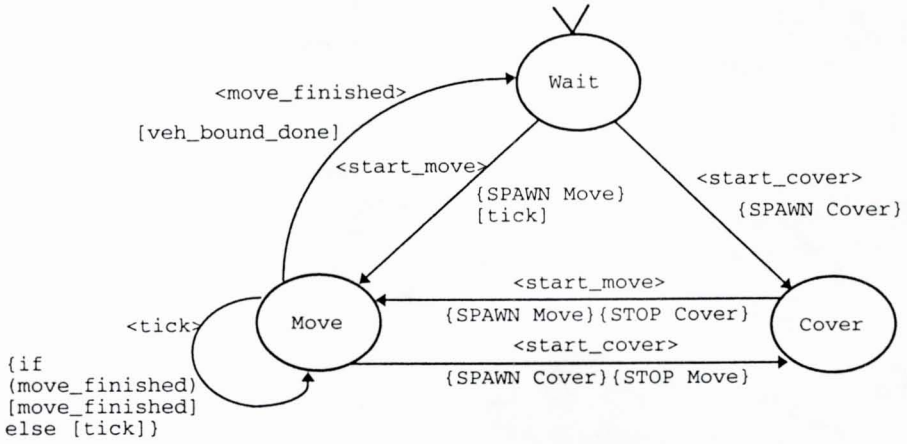


Figure 15: Bounding Overwatch Vehicle Commander FSM.

Figure 15 shows the Bounding Overwatch Vehicle Commander FSM. Initially, the FSM is in the `Wait` state. The order to start a move (via event `<start_move>`) takes the FSM to the `Move` state. As part of the transition the FSM spawns a ModSAF Move task (via STP `{SPAWN Move}`). This is a low level ModSAF behavior that a vehicle uses to travel.

Periodically, the Vehicle Commander checks if it has finished traveling. This check is made every time the FSM receives a `<tick>` event. When the move is finished the Vehicle Commander sends a radio message, `[veh_bound_done]` (vehicle bound done), and transitions to the `Wait` state to receive further orders from its Section Commander.

6.3 Describing Commander FSMs in Data Files

FSMs describing the cooperative behavior of commanders are written in data files. This approach allows quick behavior specification; a user only needs to change a data file to create a new behavior, code changes are not required.

6.3.1 FSM Grammar Production Rules

To describe FSMs production rules were developed. These production rules specify the structure of an FSM description. FSM descriptions are "parsed" based on production rules and a representation of the FSM is created inside the computer.

The production rules for the FSM grammar are:

FSM	⇒	(State)
State	⇒	(state_name (Event)) || (state_name (Event)) State
Event	⇒	(event_name next_state (STP)) || (event_name next_state (STP)) Event
STP	⇒	(TRUE (Actions)) STP || (PRED (Actions) (Actions)) STP || (FUNC) || ε
Actions	⇒	MSG string Actions || EVENT string Actions || SPAWN string Actions || STOP string Actions || ε
PRED	⇒	string
FUNC	⇒	string

where:

ε is the symbol for a NULL string.

An operator (TRUE, PRED, and FUNC) specifies how Actions are to be treated. TRUE means execute unconditionally the Actions that follow. PRED is a user specified *predicate function*. Based on the result of the predicate function, true or false, the first or the second list of Actions is executed. FUNC is a user defined function. The Actions specify what is to be done. MSG means to broadcast the string that follows as a radio message. EVENT means to put an internal event, string, on the internal event queue for processing. SPAWN means to spawn a ModSAF task specified by string. STOP means to stop a ModSAF task, specified by string, which was spawned earlier.

6.4 Change in Command

In the real world, when a commander becomes disabled, the next in command (deputy) commander takes charge. Shifting command enable units to continue their missions with minimal disruption. This important real world feature was implemented in this project.

In the simulated battlefield, a deputy commander models his commander's cooperative behavior via an FSM similar to the commander's FSM. This model (FSM) is constantly updated through receipt of observation and radio messages. This information keeps the model synchronized with the original commander's cooperative behavior. A deputy commander "knows" what his commander is doing because the deputy's commander's FSM goes through the same transitions as his commander's FSM. (An important characteristic of the model is that information flow is *unidirectional*; i.e., information contained in observation and radio messages flows into the model but does not flow out, e.g., a deputy commander does not transmit radio messages which are intended for transmission by the original commander). If required, a deputy commander can assume command and continue the mission from the last executed command of the original commander.

In the simulated battlefield, all entities watch out for each other and respond when someone is disabled. When an entity is disabled, such as by a firepower kill, another Vehicle Commander is notified by a "vehicle destroyed" *observation message* from its Observation Module. The observation message contains the Vehicle ID of the disabled vehicle. Upon receipt of the Observation Message, the Vehicle Commander sends a "vehicle destroyed" *radio message* containing the disabled vehicle's Vehicle ID. This message is sent only once.

A deputy commander runs a ModSAF FSM, called a Monitor FSM, to process vehicle-destroyed messages. Because deputy commanders are present at different levels in the command hierarchy, such as deputy commanders for Platoon and Section Commanders (Section 6.1), Monitor FSMs are also present at different levels. When a Monitor FSM receives a vehicle-destroyed message it checks the vehicle ID in the message with the vehicle ID of the original commander. If they are different, the message is discarded. Otherwise, the Monitor FSM changes the Role Matrix (Section 6.1) to reflect the change of command.

When a commander is disabled, ModSAF designates another entity as the commander and restarts the unit's mission. ModSAF developers believe that restarting the mission reflects the change in command; another entity is "promoted" to the commander. ModSAF's internal architecture imposed a barrier to implementing this change of command process. The new commander plans and executes the task using the current vehicles and positions.

Because changes to the ModSAF software, to disable automatic mission restart, involve a fundamental change to the ModSAF architecture, IST did not pursue this approach. However, to test the transfer of command, IST designated an entity to be the Platoon Commander which is different than the ModSAF-designated Platoon Commander. When the IST-designated Platoon Commander is destroyed, control is transferred to the Platoon Sergeant.

7. Results

The formal FSM DCA was implemented in ModSAF version 1.5.1. Bounding Overwatch with explicit and implicit cooperation was implemented and tested. Section 7.1 shows a platoon in Bounding Overwatch cooperating using radio messages (explicit cooperation) while Section 7.2 shows a platoon in Bounding Overwatch using observed behavior (implicit cooperation).

7.1 Platoon Bounding Overwatch Using Explicit Cooperation

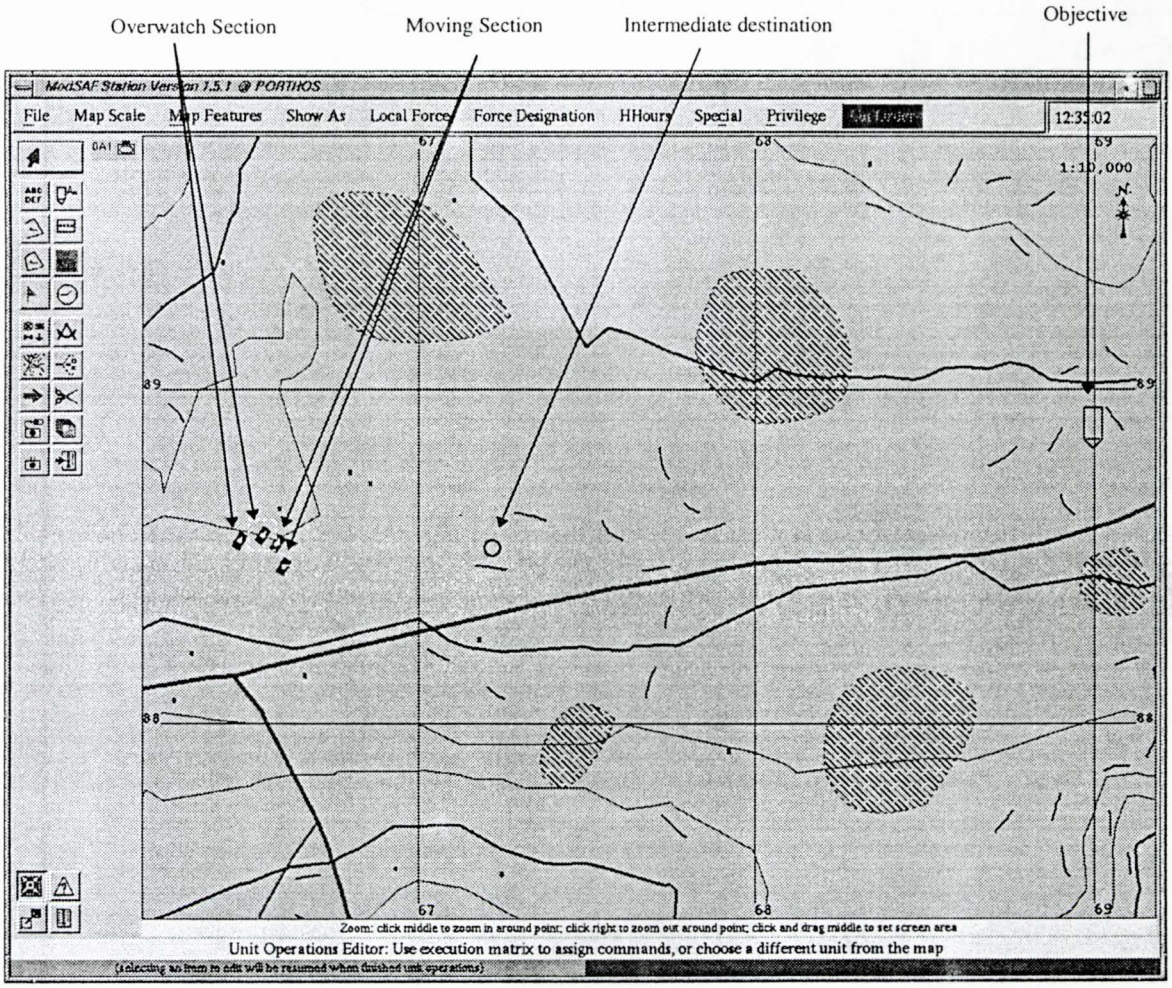


Figure 16: Platoon Bounding Overwatch: Initial position.

Figure 16 shows a platoon at its initial position. The remaining figures in this and the following section show portions of the map where significant actions take place. All activities take place within the context of the scenario that starts and ends at the locations shown in this figure. Note that vehicles are shown enlarged from their actual size.

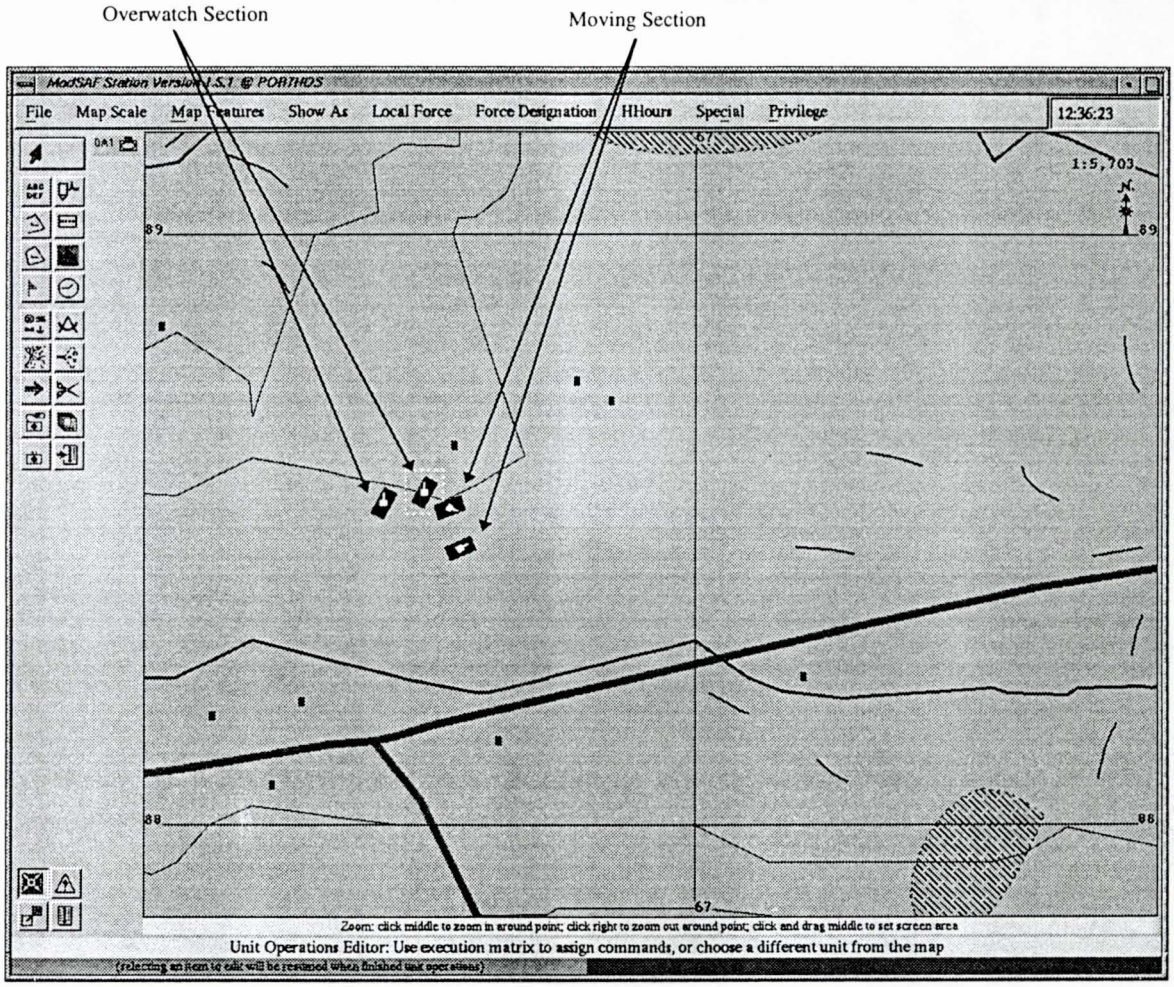


Figure 17: Start (explicit): One section starts moving.

Figure 17 shows a platoon starting a Bounding Overwatch. Both the vehicles in the Moving Section have started moving at the same time which is shown by their being abreast of each other. The Section Commander of the Moving Section gives the order to move via radio and both Vehicle Commanders respond simultaneously.

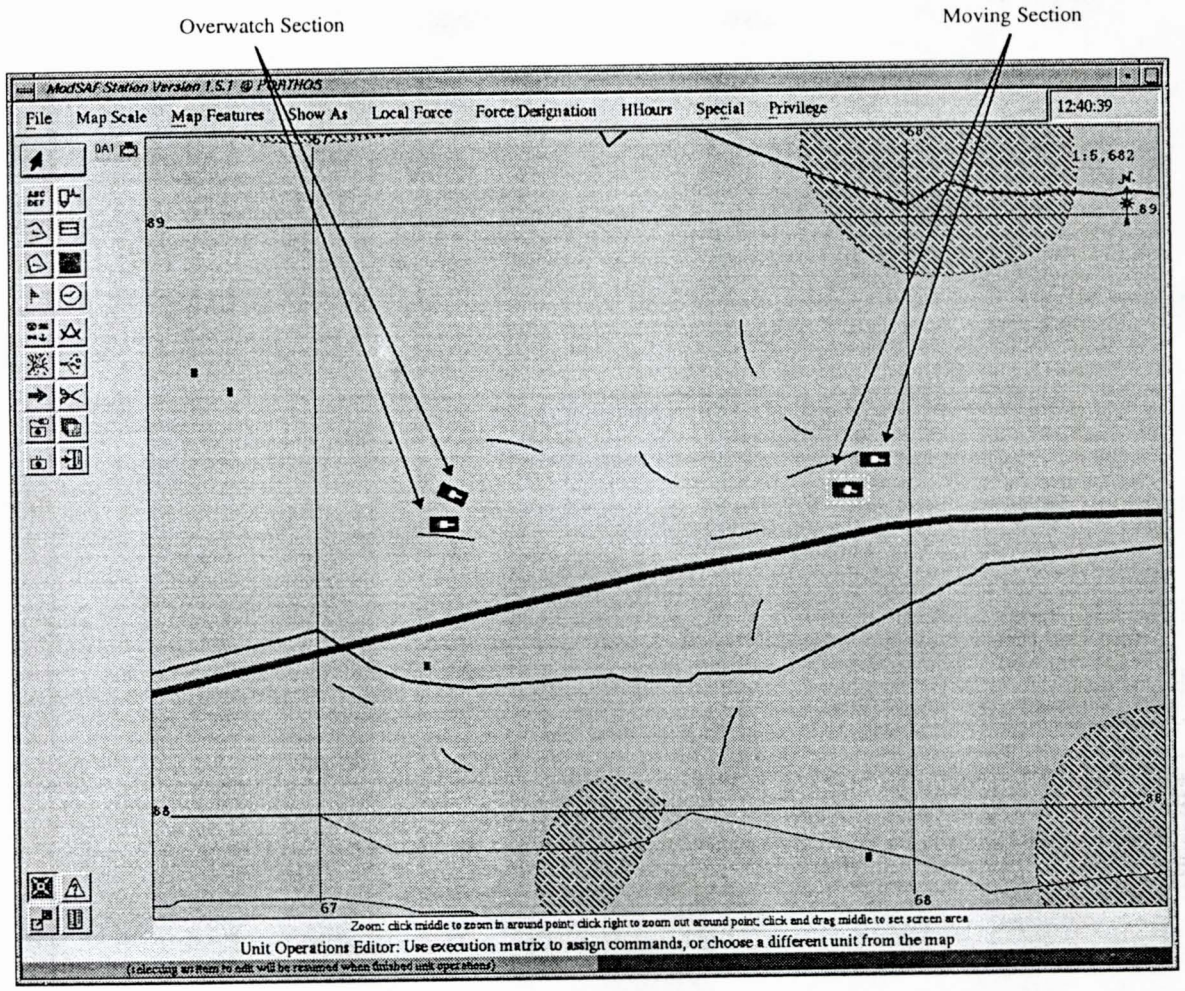


Figure 18: Intermediate stage (explicit): Vehicle move together.

Figure 18 shows Bounding Overwatch at the halfway mark. At this point, the original Bounding Section is in overwatch and the original Overwatch Section is moving.

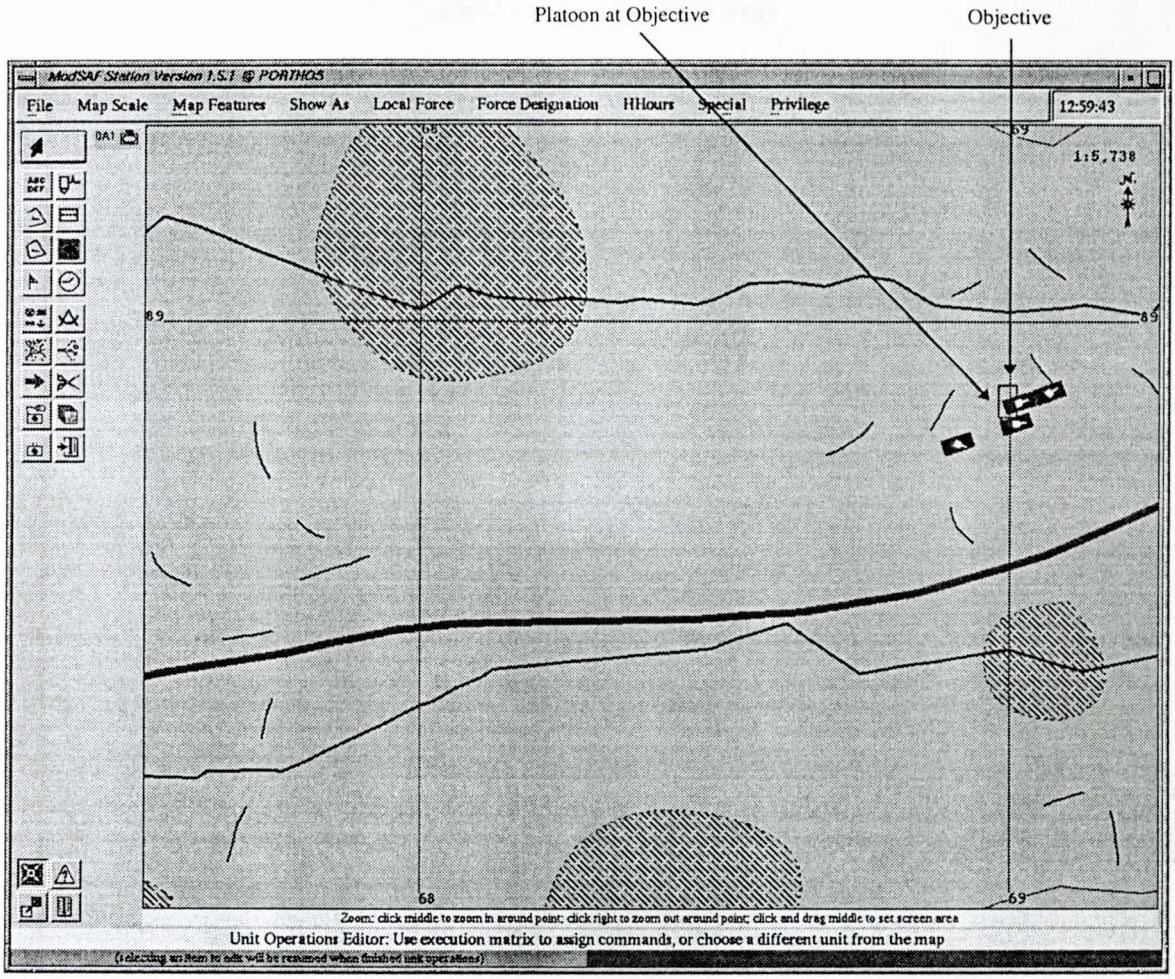


Figure 19: End (explicit): Platoon at Objective.

Figure 19 shows the platoon has arrived at the Objective. One section arrives at the Objective first and overwatches the movement of the other as it arrives.

7.2 Platoon Bounding Overwatch Using Implicit Cooperation

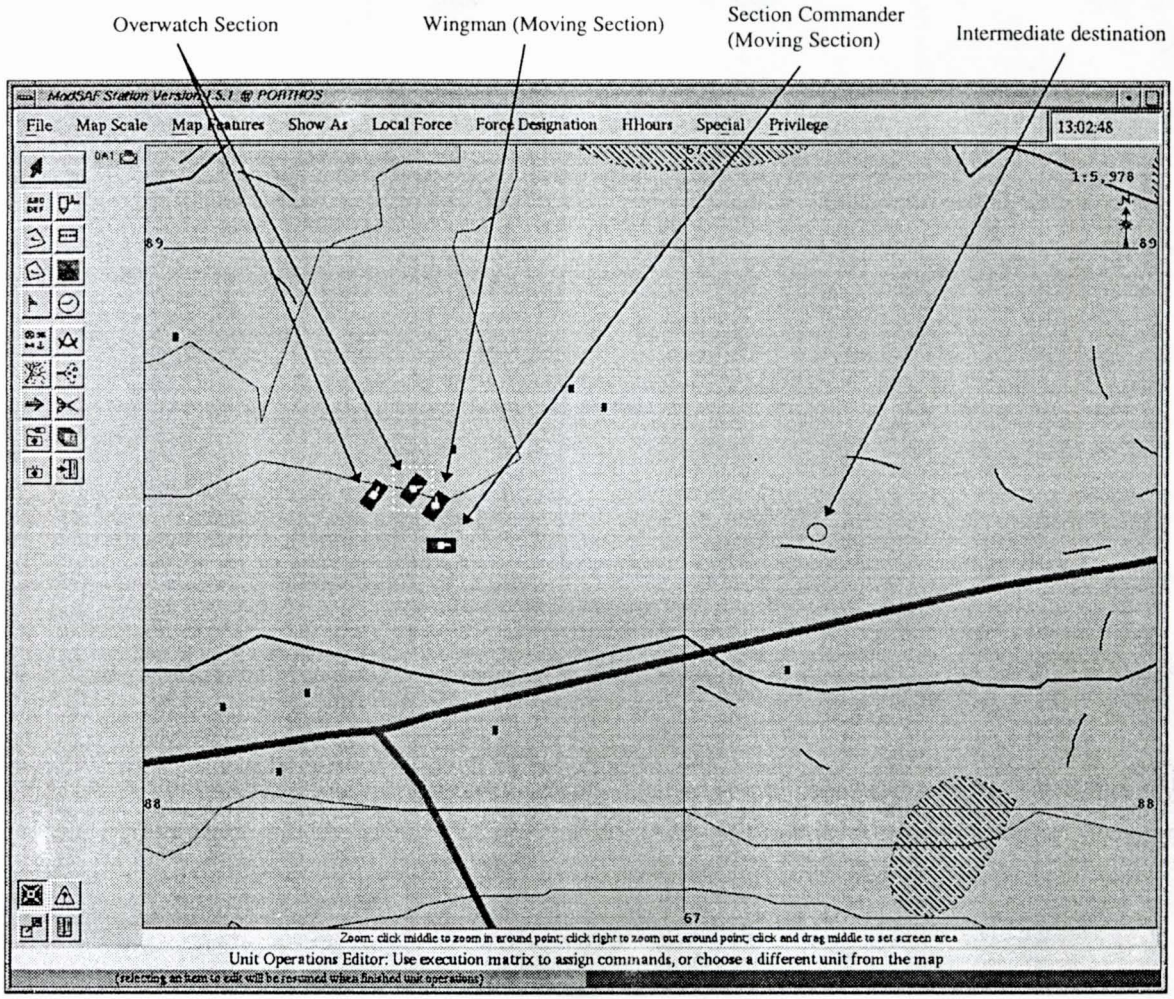


Figure 20: Start (implicit): Section Commander starts moving.

Figure 20 shows a platoon just starting a Bounding Overwatch using implicit cooperation. Note that the Section Commander has started moving (it is not facing where the other vehicles are facing) whereas the Wingman is still stationary. This is because the Wingman has not yet noticed that the Section Commander is moving. Soon, the Wingman notices the Section Commander is moving and starts following the Section Commander.

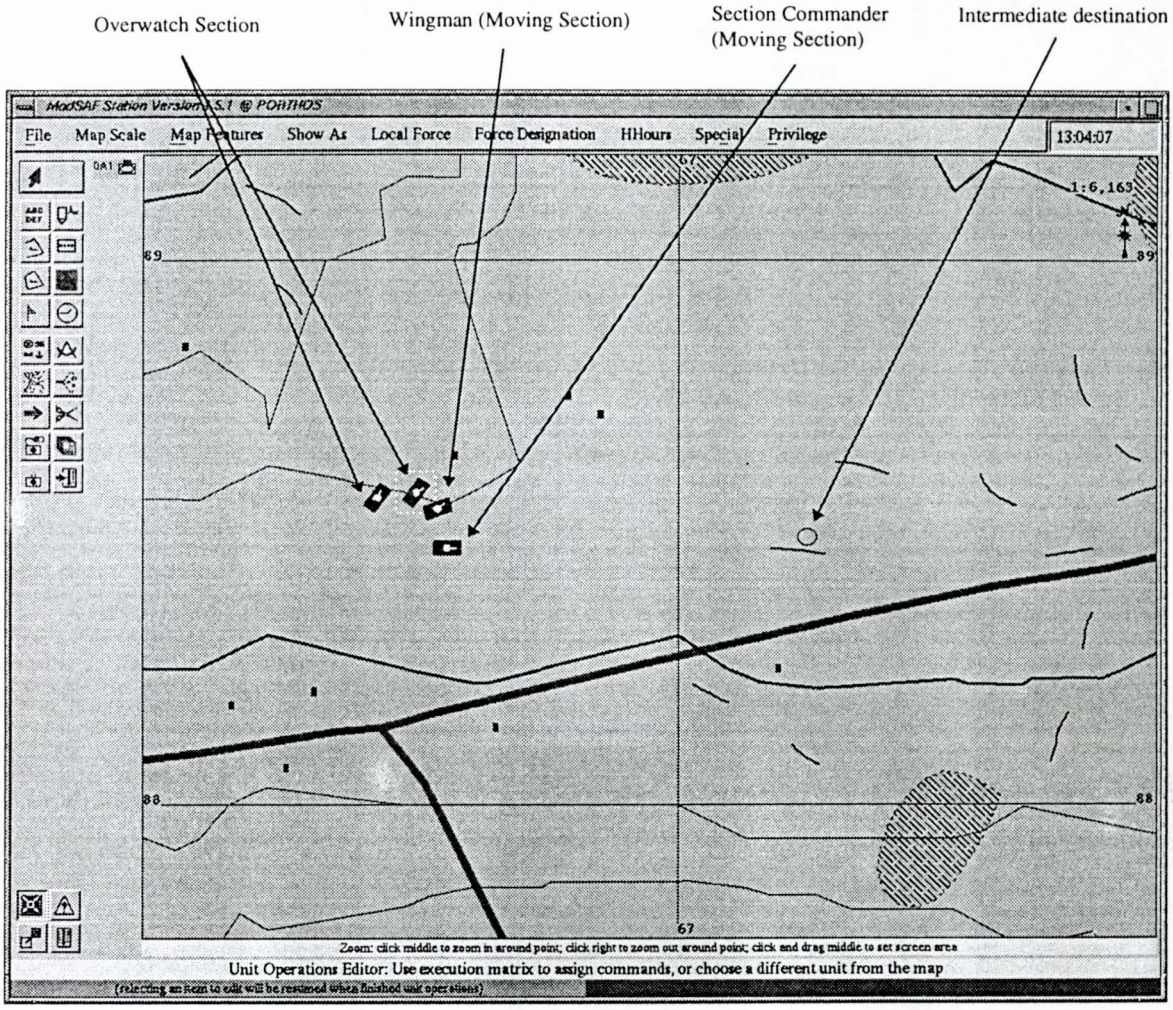


Figure 21: Start (implicit): Wingman starts moving.

In Figure 21, the Wingman is starting to move. After the short delay in observing the Section Commander's movement, the Wingman is now following the Section Commander.

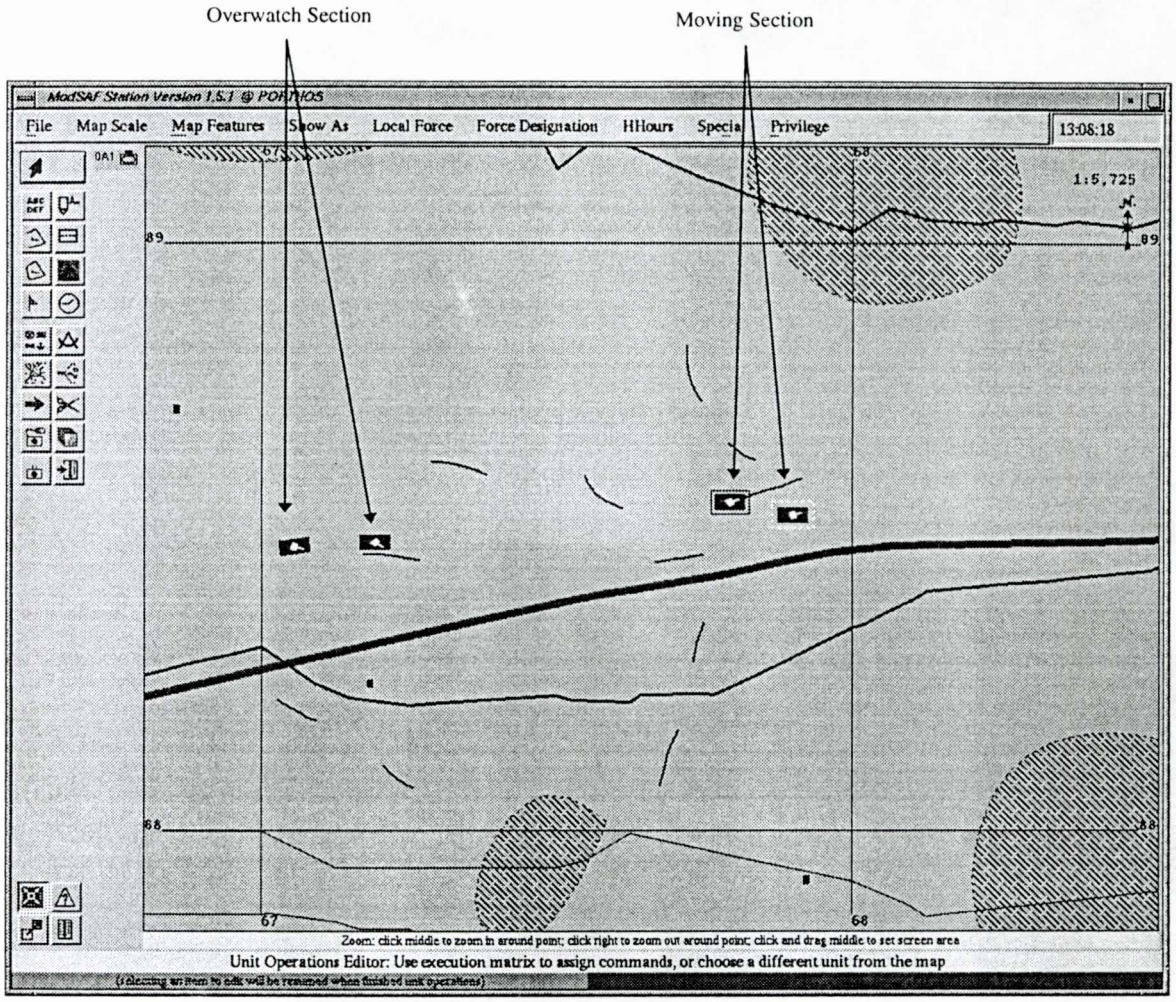


Figure 22: Intermediate stage (implicit): One section overwatches the movement of another.

Figure 22 shows the sections about halfway to the Objective. The vehicles in the Overwatch Section are not aligned as they are in the explicit case (Figure 18). There is a gap between the Section Commander's position and that of the Wingman's. This is because the Wingman's start is delayed and this delay shows up as the gap between Wingman and Section Commander.

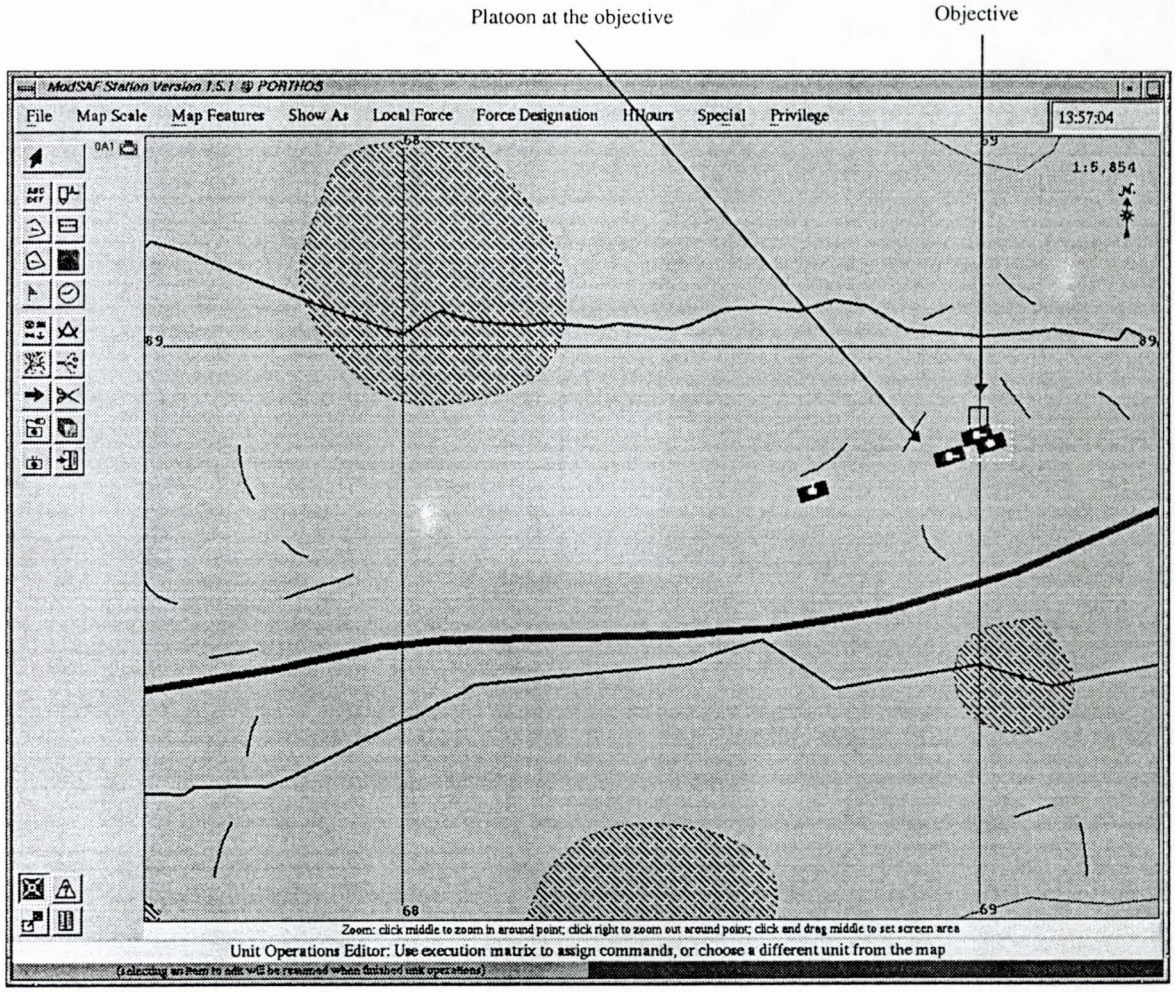


Figure 23: End (implicit): The platoon arrives at the Objective.

Figure 23 shows the platoon at the Objective.

8. Conclusions

To be effective enemy and adjunct friendly forces CGF systems must be able to portray cooperative behavior between entities realistically. There are two ways to implement cooperative behavior: Centralized and Decentralized control. Centralized control has been used in a variety of CGF systems. In this control an unseen "entity" controls and directs the behavior of subordinate entities. In addition to being unrealistic, this approach can become arbitrarily complex if larger echelons, such as battalions, are to be modeled (Section 4.1). Decentralized control involves entities cooperating with each other directly and mirrors cooperation in the real world. DCAs have several advantages as discussed in Section 4.2

This project has implemented a Decentralized Control Architecture (DCA) within the ModSAF CGF system. In addition to mirroring cooperation in the real world, DCAs allow cooperation within larger units (companies, battalions, etc.) to be modeled with little increase in complexity. Explicit and implicit cooperation between entities has been demonstrated within a platoon engaged in a Bounding Overwatch.

The cooperative behavior of an entity is implemented through FSMs. An entity's cooperative behavior is described in data files. These descriptions are read and converted into FSM representations inside the computer. Communication between entities is implemented by FSM communication. FSMs communicate by sending each other external events implemented as radio messages. An FSM communicates with itself by sending internal events. An FSM Engine, embedded within ModSAF, "reads" the description and executes the defined behavior. The FSM Engine is general purpose and can be used by other ModSAF code; ModSAF has been extended.

The FSMs use low level ModSAF behaviors. For example, a ModSAF task is used for vehicle travel as the underlying fundamental behavior. This attempts to reuse code as much as possible. Thus, in addition to being extendible, the approach is built on top of ModSAF.

Simpler implementations result as a consequence of the FSM approach. Instead of modeling various responsibilities of a commander as a large and complex FSM, responsibilities corresponding to different levels in the command hierarchy are modeled as separate FSMs which communicate with each other.

9. References

1. Danisas, K., Smith, S. H., and Wood, D. D. (1990). "Sequencer/Executive for Modular Simulator Design", *Technical Report IST-TR-90-1*, Institute for Simulation and Training, University of Central Florida, 16 pages.

2. DIS Steering Committee (1993). "The DIS Vision: A Map to the Future of Distributed Simulation", *IST Technical Report*, 47 pages.

3. Gonzalez, G., Mullally, D. E., Smith, S. H., Vanzant-Hodge, A. F., Watkins, J. E., and Wood, D. D. (1990). "A Testbed for Automated Entity Generation in Distributed Interactive Simulation", *Technical Report IST-TR-90-15*, Institute for Simulation and Training, University of Central Florida, 37 pages.

4. Loper, M. L., Thompson, J. R., and Williams, H. L. (1991). "Simulator Networking: What Can It Offer The Training Community?", *Military Simulation & Training*, Issue 4 1991, pp. 11-14.

5. Petty, M. D. (1992). "Computer Generated Forces in Battlefield Simulation", *Proceedings of the Southeastern Simulation Conference 1992*, The Society for Computer Simulation, Pensacola FL, October 22-23 1992, pp. 56-71.

6. Smith, S. H., Karr, C. R., Petty, M. D., Franceschini R. W., Wood, D. D., Watkins, J. E, and Campbell, C. E. (1992a). "The IST Computer Generated Forces Testbed", *Technical Report IST-TR-92-7*, Institute for Simulation and Training, University of Central Florida.

7. Smith, S. H., and Petty, M. D. (1992b). "Controlling Autonomous Behavior in Real-Time Simulation", *Proceedings of the Southeastern Simulation Conference 1992*, The Society for Computer Simulation, Pensacola FL, October 22-23 1992, pp. 27-40.

8. Loral (1993). "ModSAF Behavior Simulation and Control", *Proceedings of the Third Conference on Computer Generated Forces and Behavioral Representation*, University of Central Florida, Orlando, Florida, pp. 347-356.

9. Loral (1994). "ModSAF User Manual," Loral Advanced Distribution Simulation, Cambridge, Massachusetts, September 30, 1994.

10. US Army (1990). "FM 7-7J: The Mechanized Infantry Platoon And Squad (Bradley)", Coordinating Draft, Department of the Army, United States Army Infantry School, Fort Benning, Georgia 31905.

11. Loral (1995a). "Libuoverwatchmove Online Documentation", Loral Advanced Distribution Simulation, Cambridge, Massachusetts, April 28, 1995.

12. Laird, John E., Jones, Randolph M., and Nielsen, Paul E (1994). "Coordinated Behavior of Computer Generated Forces in TacAir-Soar", *Proceedings of the Fourth Conference on Computer Generated Forces and Behavioral Representation*, University of Central Florida, Orlando, Florida, pp. 325-332.

13. Shoham Y., and Tennenholtz M (1992). "On The Synthesis Of Useful Social Laws For Artificial Agents Societies", (preliminary report), *Proceedings of AAAI-92*, Morgan Kaufmann, 1992.

14. Tambe, Milind, and Rosenbloom, Paul S (1995). "Agent Tracking in Complex Multi-agent Environments: New Results", *Proceedings of the Fifth Conference on Computer Generated Forces and Behavioral Representation*, University of Central Florida, Orlando, Florida, pp. 125-133.

15. Noreils, Fabrice R. (1993). "Toward a Robot Architecture Integrating Cooperation between Mobile Robots". The International Journal of Robotics Research, vol. 12, no. 1, February 1993.

16. Noreils, Fabrice R. (1992a). "An Architecture for Cooperative and Autonomous Mobile Robots". Proceedings of the 1992 IEEE International Conference on Robotics and Automation, Nice, France, May 1992, pp. 2703-2710.

17. Noreils, Fabrice R. (1992b). "Multi-Robot Coordination for Battlefield Strategies". Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems, Raleigh, NC, July 7-10, 1992, pp. 1777-1784.

18. Noreils, Fabrice R. (1992c). "Coordinated Protocols: An Approach to Formalize Coordination Between Mobile Robots". Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems, Raleigh, NC, July 7-10, 1992, pp. 717-724.

19. Parker, Lynne E. (1994). "ALLIANCE: An Architecture for Fault Tolerant, Cooperative Control of Heterogeneous Mobile Robots". Proceedings of the IEEE/RSG/GI International Conference on Intelligent Robots and Systems (IROS '94) Vol. 2, 1994, pp. 776-783.

20. Shin, Kang G. and Epstein, Mark E. (1990). "Intertask Communications in an Integrated Multirobot System". Multirobot Systems, IEEE Computer Society Press, Los Alamitos, California, 1990.

21. Lefebvre, D. R. and Saridis, G. N. (1992). "A Computer Architecture for Intelligent Machines". Proceedings of the 1992 IEEE International Conference on Robotics and Automation.

22. Smith, R. G. and Davis, R. (1981). "Framework for Cooperation in Distributed Problem Solving". Proceedings of the IEEE Transactions on System, Man and Cybernetics, Vol. SMC-11, No. 1, January 1981, pp. 61-70.

23. Fisher, M. and Woodridge, M. (1994). "Specifying and Executing Protocols for Cooperative Action". CKBS-94, Proceedings of the Second International Working Conference on Cooperating Knowledge Systems, Springer-Verlag, 1994.

24. Decker, K. S. (1987). "Distributed Problem-Solving Techniques: A Survey". Proceedings of the IEEE Transactions on Systems, Man, and Cybernetics, Vol. SMC-17, No. 5, September/October 1987.

25. Ohko, T., Hiraki, K, and Anzai, Y. (1993). "LEMMING: A Learning System for Multi-Robot Environments". Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems, Vol. 2, July 1993, pp. 1141-1146.

26. Laengle, T. and Lueth, T.C. (1994a). "Decentralized Control of Distributed Intelligent Robots and Subsystems". Proceedings of the IFAC Symposium on Artificial Intelligence in Real Time Control (AIRTC '94).

27. Laengle, T. and Lueth, T.C. (1994b). "Task Description, Decomposition, and Allocation in a Distributed Autonomous Multi-Agent Robot System". Proceedings of the 1994 IEEE/RSJ International Conference on Intelligent Robots and Systems.

28. Corkill, Daniel (1991). "Blackboard Systems". AI Expert 6(9):40-47, September 1991.

29. Occello, M. and Demazeau, Y. (1994). "Building Real Time Agents using Parallel Blackboards and its use for Mobile Robotics". Proceedings of the 1994 IEEE International Conference on Systems, Man and Cybernetics, San Antonio, October 1994.

30. Dai, H., Hughes, J. G. and Bell, D. A. (1993). "A Distributed Real-Time Knowledge-Based System and its Implementation using Object-Oriented Techniques". Proceedings of the International Conference on Intelligent and Cooperative Information Systems, May 1993, pp. 23-30.

31. Lun, V. and MacLeod, I. M. (1992). "Strategies for Real-Time Dialogue and Interaction in Multiagent Systems". Proceedings of the IEEE Transactions on Systems, Man and Cybernetics, Vol. 22, No. 4, July/August 1992.

32. Wang, J. (1994). "On Sign-board Based Inter-Robot Communication in Distributed Robotic Systems". Proceedings of the 1994 IEEE International Conference on Robotics and Automation, Vol. 2, May 1994, pp. 1045-1050.

33. Harmon, S. Y., Aviles, W. A., and Gage, D. E. (1986). "A Technique for Coordinating Autonomous Robots." IEEE International Conference on Robotics and Animation, 1986, Vol. 1, page 666.

34. Payton, David W. and Dolan, Charles P. (1991). "Cooperative Control". Seminars on Robotics in the Air/Land Battlefield, NATO Defense Group, 1991.

35. Peterson, J. L. (1981). Petri Net Theory and the Modeling of Systems, Prentice-Hall, 1981.

36. Murata T. (1989). "Petri Nets: Properties, Analysis, and Applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541-580, 1989.

37. Genrich, H. (1986). "Predicate/Transition nets," Petri Nets: Central models and their properties - Lecture Notes in Computer Science, pages 208 - 247, Springer Verlag, 1986.

38. Wang, Fei-Yue, Mittman, Michael, and Saridis, G. N. (1992). "Coordination Specification for a Robotic Platform System." Proceedings of the 1992 IEEE/RSJ International Conference on Robots and Systems, Raleigh, NC, July 7-10, 1992.

39. Zhou, Mengchu, Wang, David T., and Chao, Daniel Y. (1994). "Design of Command and Control Systems Using Petri Nets and Object-Oriented Technology". Proceedings of the 1994 IEEE International Conference on Systems, Man, and Cybernetics, San Antonio, October 2-5, 1994.

40. Bachatene, H. and Seghrouchni, A. (1993). "A Multiformalism Approach to Formalize Intelligent Cooperative Information Systems". Proceedings of the International Conference on Intelligent and Cooperative Information Systems, May 1993, pp. 13-22.

41. Yuta, Shun'ichi and Premvuti, Suparerk (1992). "Coordinating Autonomous and Centralized Decision Making to Achieve Cooperative Behaviors Between Multiple Mobile Robots". Proceedings of the 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems, Raleigh, NC, July 7-10, 1992, pp. 1566-1574.

42. Taipale, Tapio and Hirai, Shigeoki (1993). "A Behavior-Based Control System Applied Over Multi-Robot System". Proceedings of the 1993 IEEE/RSJ International Conference on Intelligent Robots and Systems, Yokohama, Japan, July 26-30, 1993, pp. 1941-1943.

43. Arai, T., Ogata, H., and Suzuki, T. (1989). "Collision Avoidance Among Multiple Robots Using Virtual Impedence," in IEEE/RSG International Workshop on Intelligent Robots and Systems, Tsukuba, Japan, September 4-6, 1989, pp. 479-485.

44. Suchman, L. (1987). "Plans and Situated Actions," Cambridge University Press, 1987.

45. Sudkamp, Thomas A. (1988). Languages and Machines: An Introduction to the Theory of Computer Science. Addison-Wesley Publishing Company Inc., 1988.

46. Petty, Mikel D. (1995). "Computer Generated Forces in Distributed Interactive Simulation," in Proceedings of the SPIE conference on Distributed Interactive Simulation Systems for Simulation and Training in the Aerospace Environment, Orlando, FL, April 19-20, 1995, pp. 251-280.

47. Loral (1995b). "ModSAF 1.5.1 Developer's Course Exercise Workbook," LADS Document No. 94006, v. 1.5.1, July 24, 1995.

10. APPENDICES

10.1 Glossary

Term	Description
ADST	Advanced Distributed Simulation Technology
ARPA	Army Research Projects Agency
BB	Blackboard
CA	Control Architecture
CCA	Centralized Control Architecture
CGF	Computer Generated Forces
DARPA	Defense Army Research Projects Agency
DCA	Decentralized Control Architecture
DIS	Distributed Interactive Simulation
FSM	Finite State Machine
FUNC	Function
IST	Institute for Simulation and Training
KS	Knowledge Source
ModSAF	Modular Semi-Automated Forces
MSG	Message
OPFOR	Opposing Forces
PC	Platoon Commander
PDU	Protocol Data Unit
PN	Petri Net
Pr/Tr	Predicate/Transition
PRED	Predicate
SAF/SAFOR	Semi Automated Forces
SC	Section Commander
STP	State Transition Procedure
STRICOM	Simulation Training and Instrumentation Command
VC	Vehicle Commander
WISSARD	What If Simulation System for Advanced Research and Development

Table 1: Glossary

10.2 Writing a New Cooperative Behavior

The DCA, described in this report, is general enough to allow addition of new cooperative behaviors. Cooperative behaviors are described as FSMs in data files. To write a new cooperative behavior entails: Writing FSM descriptions and Code changes.

10.2.1 Writing FSM descriptions

Design FSM(s) for the new cooperative behavior. After FSMs have been designed and their operation verified, write FSM data files like the ones shown in Section 10.3 or refer to the files "uisfom_pltcmdr.rdr," "uisfom_seccmdr.rdr," and "uisfom_vehcmdr.rdr" in libraries "libpltcmdr," "libseccmdr," and "libvehcmdr" respectively. These data files contain Bounding Overwatch FSM descriptions for Platoon, Section, and Vehicle Commanders.

10.2.2 Code Changes

Code changes entail:

1. Create the behavior task: This is a user-assignable task which starts the process. For example, assigning the "UISFOverwatchMove" to a unit executes a Bounding Overwatch using explicit cooperation. Create this task (do the "advanced" exercise in creating new behaviors, (Loral [1995b])) and then modify the task FSM (refer to the file "uisfom_task.fsm" in library "libuisfoverwatchmove") to include the creation of the Role Matrix. Also, "spawn" the Platoon Commanders as done in "uisfom_task.fsm."

2. Changes to Platoon and Section Commander ModSAF FSMs: Refer to the files "pcmdr_task.fsm" and scmdr_task.fsm" in libraries "libpltcmdr" and "libseccmdr" respectively. These files contain Platoon and Section Commander ModSAF FSMs. In the ModSAF FSM state where subordinate commanders are spawned, "SpawningSectionCommanders" and "SpawningVehicleCommanders", introduce a "case" statement for the new behavior and spawn the subordinate commanders (refer to how subordinate commanders for Bounding Overwatch are spawned).

   After steps 1 and 2, the command hierarchy is established. Steps 3 to 7 pertain to the new behavior.

3. Reading FSM descriptions: Refer to the file "event_mgr.c" in library "libcoopbhv." Modify the function "InitFSMData" so that FSM files, created in Section 10.2.1, are read. The modification is adding a "case" statement for the new behavior.

   After this step, data file FSM descriptions are in the computer.

4. Mapping radio messages to internal events: Refer to files "plt_radio_map.c," "sec_radio_map.c," and "veh_radio_map.c" in library "libcoopbhv." The new behavior may entail the creation of new radio messages which must be mapped to internal events. Modify the "map" function in these files to include the new mapping.

5. Mapping observation messages to internal events: Refer to files "plt_obs_map.c," "sec_obs_map.c," and "veh_obs_map.c" in library "libcoopbhv." The new behavior may entail the creation of new Observation Events which must be mapped to internal events. Modify the "map" function in these files to include the new mapping.

   After steps 4 and 5, *external* events are mapped to *internal* events.

6. Predicate and other functions: The new behavior may call predicate and other functions. Write these function in an appropriate file and make sure that it is included in the system.

7. Modifications to event dispatcher to use predicate and other functions: Refer to the file "queue.c" in "libcoopbhv." Predicate and other functions added in Step 6 have to be called. Modify the event dispatcher function, "DispatchEvents," to include calls to them. See how another predicate function, "MoveFinished," is used and repeat the process for the new functions.

After following the procedure outlined above, a new cooperative behavior will be created. Compile and link the system. Run the ModSAF executable and assign the new behavior to a unit. Verify that the behavior is correct. For more information, contact the first author listed on this report.

10.3 Bounding Overwatch FSM descriptions

Sections 10.3.1 and 10.3.2 show data files describing the cooperative behavior of various commanders for Bounding Overwatch. Each data file has three sections. The first section, States, list the states of the FSM.

The second section, Events, lists valid events for the FSM. The third section, FSM description, contains the FSM description in the format specified by the production rules of the FSM Grammar (Section 6.3.1).

10.3.1 Explicit Cooperation

10.3.1.1 Platoon Commander Behavior

```
; Platoon Commander FSM - Explicit Cooperation
(
 ; States
 (
        S_WAIT
        S_DONE
 )

 ; Events
 (
        E_PLT_BOUND_DONE
 )

 ; FSM description
 (S_WAIT
        (E_PLT_BOUND_DONE S_DONE
                ()
  )
 )
)
```

10.3.1.2 Section Commander Behavior

```
; Section Commander FSM - Explicit Cooperation
(
 ; States
 (
        S_WAITING
        S_EXEC_MOVE
        S_WMAN_BOUND_FINISHED
        S_I_ARRIVE
        S_EXEC_COVER
        S_BETA_SEC_AT_OBJ
        S_WMAN_BOUND_FINISHED_WITH_BETA_SEC_AT_OBJ
        S_I_ARRIVE_WITH_BETA_SEC_AT_OBJ
        S_DONE
 )

 ; Events
 (
        E_START_MOVE
        E_START_COVER
        E_WINGMAN_COMPLETE
        E_MY_BOUND_DONE
        E_BETA_SEC_BOUND_DONE
        E_BETA_SEC_AT_OBJ
        E_ALPHA_SEC_AT_OBJ
 )
```

```
; FSM description
(S_WAITING
        (E_START_MOVE  S_EXEC_MOVE
                (TRUE (MSG START_MOVE))
        )
        (E_START_COVER S_EXEC_COVER
                (TRUE (MSG START_COVER))
        )
)

(S_EXEC_MOVE
        (E_WINGMAN_COMPLETE S_WMAN_BOUND_FINISHED
                ()
        )
        (E_MY_BOUND_DONE S_I_ARRIVE
                ()
        )
)

(S_WMAN_BOUND_FINISHED
        (E_MY_BOUND_DONE S_EXEC_COVER
                (TRUE                   (MSG START_COVER))
                (DEST_EQUAL_OBJ (MSG SEC_AT_OBJ) (MSG SEC_BOUND_DONE))
        )
)

(S_I_ARRIVE
        (E_WINGMAN_COMPLETE S_EXEC_COVER
                (TRUE                   (MSG START_COVER))
                (DEST_EQUAL_OBJ (MSG SEC_AT_OBJ) (MSG SEC_BOUND_DONE))
        )
)

(S_EXEC_COVER
        (E_BETA_SEC_BOUND_DONE S_EXEC_MOVE
                (TRUE (MSG START_MOVE))
 )
        (E_ALPHA_SEC_AT_OBJ S_DONE
                ()
 )
        (E_BETA_SEC_AT_OBJ S_BETA_SEC_AT_OBJ
                (TRUE (MSG START_MOVE))
 )
)

(S_BETA_SEC_AT_OBJ
        (E_WINGMAN_COMPLETE S_WMAN_BOUND_FINISHED_WITH_BETA_SEC_AT_OBJ
                ()
        )
        (E_MY_BOUND_DONE S_I_ARRIVE_WITH_BETA_SEC_AT_OBJ
                ()
        )
)
```

```
 (S_WMAN_BOUND_FINISHED_WITH_BETA_SEC_AT_OBJ
        (E_MY_BOUND_DONE S_DONE
                (TRUE (MSG PLT_BOUND_DONE MSG START_COVER))
  )
 )

 (S_I_ARRIVE_WITH_BETA_SEC_AT_OBJ
        (E_WINGMAN_COMPLETE S_DONE
                (TRUE (MSG PLT_BOUND_DONE MSG START_COVER))
  )
 )
)
```

10.3.1.3 Vehicle Commander Behavior

```
; Vehicle Commander FSM - Explicit Cooperation
(
 ; States
 (
        S_WAIT
        S_EXEC_MOVE
        S_EXEC_COVER
 )

 ; Events
 (
        E_START_MOVE
        E_START_COVER
        E_MOVE_FINISHED
        E_TICK
 )

 ; FSM description
 (S_WAIT
        (E_START_MOVE S_EXEC_MOVE
                (TRUE (SPAWN MOVE EVENT E_TICK))
  )
        (E_START_COVER S_EXEC_COVER
                (TRUE (SPAWN COVER))
  )
 )

 (S_EXEC_MOVE
        (E_MOVE_FINISHED S_WAIT
                (TRUE (MSG VEH_BOUND_DONE))
  )
        (E_TICK S_EXEC_MOVE
                (MOVE_FINISHED (EVENT E_MOVE_FINISHED) (EVENT E_TICK))
        )
        (E_START_COVER S_EXEC_COVER
                (TRUE (STOP MOVE SPAWN COVER))
        )
 )

 (S_EXEC_COVER
```

```
        (E_START_MOVE S_EXEC_MOVE
                (TRUE (STOP COVER SPAWN MOVE EVENT E_TICK))
   )
  )
)
```

10.3.2 Implicit Cooperation

10.3.2.1 Platoon Commander Behavior

```
; Platoon Commander FSM - Implicit Cooperation
(
 ; States
 (
        S_WAIT
        S_DONE
 )

 ; Events
 (
        E_PLT_BOUND_DONE
 )

 ; FSM description
 (S_WAIT
        (E_PLT_BOUND_DONE S_DONE
                ()
   )
  )
)
```

10.3.2.2 Section Commander Behavior

```
; Section Commander FSM - Implicit Cooperation
(
 ; States
 (
        S_WAITING
        S_EXEC_MOVE
        S_WMAN_BOUND_FINISHED
        S_I_ARRIVE
        S_EXEC_COVER
        S_BETA_SEC_AT_OBJ
        S_WMAN_BOUND_FINISHED_WITH_BETA_SEC_AT_OBJ
        S_I_ARRIVE_WITH_BETA_SEC_AT_OBJ
        S_DONE
 )

 ; Events
 (
        E_START_MOVE
        E_START_COVER
        E_WINGMAN_COMPLETE
        E_MY_BOUND_DONE
        E_BETA_SEC_BOUND_DONE
```

```
        E_BETA_SEC_AT_OBJ
        E_ALPHA_SEC_AT_OBJ
)

; FSM description
(S_WAITING
        (E_START_MOVE  S_EXEC_MOVE
                (TRUE (MSG              START_MOVE
                        INSTALL_FUNC CHECK_WINGMAN_COMPLETE
                        INSTALL_FUNC CHECK_MY_BOUND_DONE))
        )
        (E_START_COVER S_EXEC_COVER
                (TRUE (MSG              START_COVER
                        INSTALL_FUNC CHECK_BETA_SEC_BOUND_DONE
                        INSTALL_FUNC CHECK_BETA_SEC_AT_OBJ
                        INSTALL_FUNC CHECK_ALPHA_SEC_AT_OBJ))
        )
)

(S_EXEC_MOVE
        (E_WINGMAN_COMPLETE S_WMAN_BOUND_FINISHED
                (TRUE (REMOVE_FUNC CHECK_WINGMAN_COMPLETE))
        )
        (E_MY_BOUND_DONE S_I_ARRIVE
                (TRUE (REMOVE_FUNC CHECK_MY_BOUND_DONE))
        )
)

(S_WMAN_BOUND_FINISHED
        (E_MY_BOUND_DONE S_EXEC_COVER
                (TRUE (MSG              START_COVER
                        REMOVE_FUNC  CHECK_MY_BOUND_DONE
                        INSTALL_FUNC  CHECK_BETA_SEC_BOUND_DONE
                        INSTALL_FUNC  CHECK_BETA_SEC_AT_OBJ
                        INSTALL_FUNC  CHECK_ALPHA_SEC_AT_OBJ))
        )
)

(S_I_ARRIVE
        (E_WINGMAN_COMPLETE S_EXEC_COVER
                (TRUE (MSG              START_COVER
                        REMOVE_FUNC  CHECK_WINGMAN_COMPLETE
                        INSTALL_FUNC  CHECK_BETA_SEC_BOUND_DONE
                        INSTALL_FUNC  CHECK_BETA_SEC_AT_OBJ
                        INSTALL_FUNC  CHECK_ALPHA_SEC_AT_OBJ))
        )
)

(S_EXEC_COVER
        (E_BETA_SEC_BOUND_DONE S_EXEC_MOVE
                (TRUE (MSG              START_MOVE
                        REMOVE_FUNC  CHECK_BETA_SEC_BOUND_DONE
                        REMOVE_FUNC  CHECK_BETA_SEC_AT_OBJ
                        REMOVE_FUNC  CHECK_ALPHA_SEC_AT_OBJ
                        INSTALL_FUNC  CHECK_WINGMAN_COMPLETE
```

```
                    INSTALL_FUNC  CHECK_MY_BOUND_DONE))
 )
      (E_ALPHA_SEC_AT_OBJ S_DONE
            (TRUE (REMOVE_FUNC  CHECK_BETA_SEC_BOUND_DONE
                    REMOVE_FUNC  CHECK_BETA_SEC_AT_OBJ
                    REMOVE_FUNC  CHECK_ALPHA_SEC_AT_OBJ))
      )
      (E_BETA_SEC_AT_OBJ S_BETA_SEC_AT_OBJ
            (TRUE (MSG              START_MOVE
                    REMOVE_FUNC  CHECK_BETA_SEC_BOUND_DONE
                    REMOVE_FUNC  CHECK_BETA_SEC_AT_OBJ
                    REMOVE_FUNC  CHECK_ALPHA_SEC_AT_OBJ
                    INSTALL_FUNC  CHECK_WINGMAN_COMPLETE
                    INSTALL_FUNC  CHECK_MY_BOUND_DONE))
 )
)

(S_BETA_SEC_AT_OBJ
      (E_WINGMAN_COMPLETE S_WMAN_BOUND_FINISHED_WITH_BETA_SEC_AT_OBJ
            (TRUE (REMOVE_FUNC  CHECK_WINGMAN_COMPLETE))
      )
      (E_MY_BOUND_DONE S_I_ARRIVE_WITH_BETA_SEC_AT_OBJ
            (TRUE (REMOVE_FUNC  CHECK_MY_BOUND_DONE))
      )
)

(S_WMAN_BOUND_FINISHED_WITH_BETA_SEC_AT_OBJ
      (E_MY_BOUND_DONE S_DONE
            (TRUE (MSG              PLT_BOUND_DONE
                    MSG              START_COVER
                    REMOVE_FUNC CHECK_MY_BOUND_DONE))
 )
)

(S_I_ARRIVE_WITH_BETA_SEC_AT_OBJ
      (E_WINGMAN_COMPLETE S_DONE
            (TRUE (MSG              PLT_BOUND_DONE
                    MSG              START_COVER
                    REMOVE_FUNC CHECK_WINGMAN_COMPLETE))
 )
)
)
```

10.3.2.3 Vehicle Commander Behavior

```
; Vehicle Commander FSM - Implicit Cooperation
(
 ; States
 (
        S_WAIT
        S_EXEC_MOVE
        S_EXEC_FOLLOW_SCMDR
        S_EXEC_COVER
 )
```

```
; Events
(
        E_START_MOVE
        E_START_COVER
        E_FOLLOW_SCMDR
        E_FOLLOW_SCMDR_FINISHED
        E_MOVE_FINISHED
        E_TICK
)

; FSM description
(S_WAIT
        (E_START_MOVE S_EXEC_MOVE
                (TRUE (SPAWN MOVE EVENT E_TICK))
 )
        (E_START_COVER S_EXEC_COVER
                (TRUE (SPAWN COVER))
 )
        (E_FOLLOW_SCMDR S_EXEC_FOLLOW_SCMDR
                (TRUE (SPAWN            FOLLOW_SCMDR
                        REMOVE_FUNC  CHECK_SCMDR_MOVEMENT
                        EVENT           E_TICK))
        )
)

(S_EXEC_MOVE
        (E_MOVE_FINISHED S_WAIT
                ()
 )
        (E_TICK S_EXEC_MOVE
                (MOVE_FINISHED (EVENT E_MOVE_FINISHED) (EVENT E_TICK))
        )
        (E_START_COVER S_EXEC_COVER
                (TRUE (STOP MOVE SPAWN COVER))
        )
)

(S_EXEC_FOLLOW_SCMDR
        (E_FOLLOW_SCMDR_FINISHED S_EXEC_COVER
                (TRUE (INSTALL_FUNC CHECK_SCMDR_MOVEMENT
                        STOP            FOLLOW_SCMDR))
        )
        (E_TICK S_EXEC_FOLLOW_SCMDR
                (FOLLOW_SCMDR_FINISHED (EVENT E_FOLLOW_SCMDR_FINISHED)
                                        (EVENT E_TICK))
        )
)

(S_EXEC_COVER
        (E_START_MOVE S_EXEC_MOVE
                (TRUE (STOP COVER SPAWN MOVE EVENT E_TICK))
        )
        (E_FOLLOW_SCMDR S_EXEC_FOLLOW_SCMDR
                (TRUE (SPAWN            FOLLOW_SCMDR
                        REMOVE_FUNC  CHECK_SCMDR_MOVEMENT
```

```
                    EVENT           E_TICK))
            )
        )
    )
```