1994-09

Autonomous agent interactions in a real-time simulation system

McAndrews, Gary Michael

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AUTONOMOUS AGENT INTERACTIONS IN A REAL-TIME SIMULATION SYSTEM

by

Gary M. McAndrews

September 1994

Thesis Advisor: David Pratt

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1994	Master's Thesis

4. TITLE AND SUBTITLE
Autonomous Agent Interactions in a Real-Time Simulation System (U)

5. FUNDING NUMBERS

6. AUTHOR(S)
McAndrews, Gary Michael

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT *(Maximum 200 words)*

The major problem addressed by this research is the design and implementation of a command and control architecture to add company-level missions to an existing real-time combat-simulation system. The US Army is using Modular Semi-Autonomous Forces (ModSAF) to conduct research in simulation training. ModSAF only provides platoon and vehicle missions. Adding company level missions to ModSAF will allow a single operator to effectively control a greater number of forces and retain realistic behaviors.

The approach taken was to utilize ModSAF's finite-state machine architecture, and NPSNET -- a three dimensional combat-simulation system, to develop, test, and implement a company-level combat simulation mission. Simplistic terrain reasoning algorithms and a command and control finite state machine architecture were added to the ModSAF system.

The result is a prototype company-level mission "Occupy an Assembly Area," providing a successful proof-of-concept implementation of company level mission development using ModSAF's current finite state machine architecture. This research provides the groundwork for further development of company-level combat simulations in ModSAF.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Autonomous agents, decision making, combat, combat-simulation, Modsaf, NPSNET, real-time systems	110
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

Approved for public release; distribution is unlimited

AUTONOMOUS AGENT INTERACTIONS IN A REAL-TIME SIMULATION SYSTEM

Gary Michael McAndrews
Major, United States Army
B.S., United States Military Academy, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1994

Author: [redacted]
Gary Michael McAndrews

Approved By: [redacted]
David R. Pratt, Thesis Advisor

[redacted]
Robert McGhee, Second Reader

[redacted]
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The major problem addressed by this research is the design and implementation of a command and control architecture to add company-level missions to an existing real-time combat-simulation system. The US Army is using the Modular Semi-Autonomous Forces (ModSAF) simulator to conduct research in simulation training. ModSAF only provides platoon and vehicle missions. Adding company level missions to ModSAF will allow a single operator to effectively control a greater number of forces and retain realistic behaviors.

The approach taken was to utilize ModSAF's finite-state machine architecture, and NPSNET -- a three dimensional combat-simulation system, to develop, test, and implement a company-level combat simulation mission. Simplistic terrain reasoning algorithms and a command and control finite state machine architecture were added to the ModSAF system.

The result is a prototype company-level mission "Occupy an Assembly Area," providing a successful proof-of-concept implementation of company level mission development using ModSAF's current finite state machine architecture. This research provides the groundwork for further development of company-level combat simulations in ModSAF.

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
B. PROBLEM STATEMENT 2
C. APPROACH 2
D. ORGANIZATION 3

II. BACKGROUND AND PREVIOUS WORK 5
A. LOCAL THESIS WORK 5
B. ModSAF 5
C. AUTOMATED FORCES DEVELOPMENT 6
D. SUMMARY 8

III. ModSAF DESCRIPTION 9
A. DESCRIPTION 9
B. ModSAF SOFTWARE ARCHITECTURE 12
C. ModSAF COMMAND AND CONTROL 13
D. FINITE STATE MACHINES 18
E. INTERACTION WITH ModSAF 18

IV. ASSEMBLY AREA 21
A. COMPANY TASK ORGANIZATION 21
B. OVERVIEW OF ASSEMBLY AREA MISSION 22
C. SUBORDINATE LEVEL MISSIONS 29

V. DESIGN STRATEGY 33
A. DESIGN GOALS 33
B. TASK ORGANIZATION 34
C. ASSEMBLY AREA MISSION STAGES 34
D. ModSAF VEHICLE, UNIT, AND REACTIONARY TASKS 41
E. FINITE STATE MACHINE ARCHITECTURE 42
F. ASSEMBLY AREA LIBRARY MODULE 42
G. COMMUNICATION BETWEEN AUTONOMOUS AGENTS 43
H. SUMMARY 43

VI. DESIGN IMPLEMENTATION 45
A. TASK ORGANIZATION 45
B. CREATING A NEW TASK LIBRARY 46
C. DEVELOPMENT OF THE FINITE STATE MACHINE CODE 47
D. FINITE STATE MACHINE SUPPORT ROUTINES 48
E. ASSEMBLY AREA LIBRARY MODULE 64
F. TASKS AND TASK FRAME MANAGEMENT 65

G. SUMMARY 66

VII. RESULTS AND CONCLUSIONS 67
A. SUMMARY 67
B. ASSESSMENT OF ASSEMBLY AREA MISSION 67
C. CONCLUSIONS 69
D. RECOMMENDATIONS FOR FUTURE WORK 70

APPENDIX A 71

APPENDIX B 79

APPENDIX C 81

APPENDIX D 89

LIST OF REFERENCES 97

INITIAL DISTRIBUTION LIST 99

LIST OF FIGURES

Figure 1: NPSNET -- Three Dimensional Simulation System....6
Figure 2: ModSAF GUI SAFStation....10
Figure 3: From Ref. [Robasky, 94-1] Shared Databases....11
Figure 4: ModSAF Library Modules....12
Figure 5: ModSAF Actions on Contact....15
Figure 6: Company Task Organization....21
Figure 7: Occupy Assembly Area....23
Figure 8: Assembly Area Task List....24
Figure 9: No Cover, No Concealment....25
Figure 10: Limited Cover, Limited Concealment....26
Figure 11: Covered, Limited Concealment....27
Figure 12: Ft. Hunter Ligget, 1:50,000 Scale....28
Figure 13: Ft. Hunter Ligget, 1:5,000 Scale....29
Figure 14: NTC, Ft. Irwin, 1:50,000 Scale....30
Figure 15: Steps for Company Assembly Area....34
Figure 16: Assembly Area Search Space....35
Figure 17: Unit Travel Route Selection....36
Figure 18: Occupation Positions for Assembly Area....38
Figure 19: 2nd Platoon's Assembly Area....39
Figure 20: Road Route for Assembly Area....40
Figure 21: ModSAF's Occupy Position Task....41
Figure 22: Assembly Area Finite State Machine....42
Figure 23: Steps to Create a Task....47
Figure 24: Assembly Area Support Routines....49
Figure 25: Assembly Area Bounding Box....51
Figure 26: State Plan Route to Recon....52
Figure 27: Moving to the Reconnaissance Area....53
Figure 28: Canopy Center of Mass Selection....55
Figure 29: Graphic Entry for Occupy Position....56
Figure 30: Allocating a Route From Roads....58
Figure 31: Unit Travel Road Route Frame 1....59
Figure 32: Unit Travel Road Route Frame 2....60
Figure 33: Conduct Roadmarch in Multiple States....62
Figure 34: Occupy Position Finding Cover....63

I. INTRODUCTION

A. BACKGROUND

For the last two years, the *Army Times* newspaper, a weekly, unofficial report of Army related issues and news, has contained articles relating two common topics which greatly affect the Army; the first is the tightening of the defense budget, and the second is the drawdown of the services. In an article dated 15 February 1993, Defense Secretary Les Aspin ordered the services to cut their operating and training budgets, with the Army taking a four percent cut, about $2.5 billion dollars, from its spending plan. Mr. Aspin recommends "more effective use of simulation and other proven techniques." [Mathews, 93]. As the Army continues to drawdown its forces, it will continue to search for effective ways to train its soldiers with less training dollars and with less trainers. In this search for more cost-efficient training methods, the Army will increasingly look at the potential for using computer simulations to train its soldiers.

The Army extensively uses simulators to train its soldiers. One example is the The Unit Conduct of Fire Trainer (UCOFT). Every armor battalion now utilizes the UCOFT for required pre-gunnery training. The UCOFT trains M2 Bradley Fighting Vehicle crews and M1 crews in crew gunnery skills. This author's use of the UCOFT in training gunnery skills to soldiers resulted in documented higher unit gunnery scores. The UCOFT computer simulation closely replicates the actual behavior of an M1 Tank. It is the realism of this simulator that is the key to its success. As the Army looks at utilizing trainers for tactical training, the issue of realism must be addressed.

Some early development in the use of tactical simulation trainers began with SIMNET (SIMulator NETworking). In 1984 the U.S. Army Armor School had a SIMNET tactical trainer site, which included a human operator control station, and 14 M1 Abrams Tank simulation stations. SIMNET, started by ARPA in 1983, was still in its early development as a computer combat simulation system that allowed the integration of manned tank and aircraft simulators with computer semi-automated forces (SAFs) [Ceranowicz, 93-1]. The

computer-generated semi-automated forces were controlled by an operator at a single workstation. In this manner, the Army could replicate large battles while only manning a few vehicles. The SAF operator could replicate both the adjacent friendly forces, and the opposing forces on his workstation. The system was designed to provide realistic behaviors from the computer forces engaging in combat with the manned simulators [Stanzione, 89].

ModSAF (Modular Semi-Automated Forces) is a continuation of the development of semi-automated forces. Its modular design provides a platform for SAF and DIS (Distributed Interactive Simulation) research. The goal of the SAF system is the same, to allow a single operator to exhibit realistic behaviors of several vehicles and larger units to fight against human opponents [Ceranowicz, 93-1]. The Naval Postgraduate School recently acquired ModSAF, version 1.0.

B. PROBLEM STATEMENT

The number of entities a human operator can manage in a combat simulation system is greatly affected by the level of autonomous decision making performed by the computer generated forces. The greater the automation of the lower-level behaviors and decision processes, the more entities the human operator can manage and still retain realistic unit and vehicle behaviors. [Ceranowicz, 93-2]. ModSAF, version 1.0, only provides platoon and vehicle level missions. The human operator must individually control each of these platoons to portray company-level behaviors. The addition of company-level missions to ModSAF, incorporating the existing platoon level missions and vehicle reactive behaviors, will enhance both the types of possible operations, and the realism of the computer forces the human operator is controlling.

C. APPROACH

Perhaps the most challenging part of the research for the implementation of company level missions in ModSAF was gaining a detailed understanding of the existing code. The release information file for ModSAF 1.0 (Loral Advanced Distributed Simulation, Inc., Copyright 1993) provides some insight into the size of this program, which includes 150

source libraries with 326,335 lines of code, approximately 215,000 lines being actual C code. This required extensive backward engineering and detailed analysis to determine the coupling and dependencies of the libraries used in ModSAF.

The incorporation of higher-level company missions is premised on utilizing the existing finite state machine architecture of ModSAF 1.0. After a detailed analysis of the design and implementation of finite state machines, and the construction of an independent library module, a prototype company-level mission, "Occupy an Assembly Area" was designed and implemented, requiring only minor changes to the current functionality of ModSAF.

D. ORGANIZATION

Chapter II introduces some of the current research work being done in the area of autonomous agents, specifically in the areas which utilize ModSAF. This chapter provides a starting point with which to discuss further development of the use of autonomous agents in combat simulations. Chapter III provides an overview of the ModSAF system and its architecture. The development of company level missions is premised to utilize the existing architecture of ModSAF, so an introduction to the finite state machine architecture and use of tasks is reviewed to provide a general understanding of ModSAF's autonomous agent control. Chapter IV describes the general concepts and procedures of an Assembly Area mission. Included in this chapter is a discussion of the overall mission and the individual and unit actions performed during the accomplishment of the mission. Chapter V is a step-by-step breakdown of the design strategy. Chapter VI is the implementation of the Assembly Area mission in ModSAF. These chapters outline the existing ModSAF functionality incorporated into the Assembly Area mission, and the specific code which was reconfigured, or designed, to attain the desired unit and vehicle behaviors. Chapter VII summarizes the accomplishments of the research in this area, and provides insight into further research which is needed in the area of company level missions using autonomous agents in ModSAF.

II. BACKGROUND AND PREVIOUS WORK

A. LOCAL THESIS WORK

The Computer Science Department at the Naval Postgraduate School developed and implemented a three-dimensional simulation system -- NPSNET -- using the Distributed Interactive Simulation (DIS) protocol (Figure 1). Soon after its development, research work began on how to incorporate autonomous agents into this three-dimensional world. Two local theses, NPSNET: Physically Based, Autonomous, Naval Surface Agents written by LT John Hearne, and Tactical Decision Making in Intelligent Agents: Developing Autonomous Forces in NPSNET by CPT Michael Culpepper, provided groundwork in the area of using an external planning agent to provide realistic behaviors to autonomous agents in NPSNET. Both Hearne and Culpepper used an expert system tool -- CLIPS -- to develop expert system shells to replicate behaviors of computer agents in a three dimensional world. LT Hearne added intelligent, autonomous naval surface ships that incorporated the complexities of actual ship turning and propulsion dynamics. CPT Culpepper added autonomous armor units that planned target selection, subordinate missions, and the cooperative efforts of platoon sized elements. Both developments provided autonomous agents that react to a changing environment by using expert system rule sets. [Hearne, 93] [Culpepper, 92]

B. ModSAF

In December 1993, the Naval Postgraduate School received ModSAF version 1.0. ModSAF provides a modular framework with which to develop and research autonomous agent design and implementation. It provides automated behaviors at the platoon and vehicle level, including simulation of platoon level tank and mechanized infantry forces [Ceranowicz, 93-1]. Whereas Culpepper and Hearne were required to handle each individual agent's behaviors, i.e. targeting, movement, line of sight, fire distribution, etc., ModSAF provides these low-level agent behaviors and the simulation platform to build and

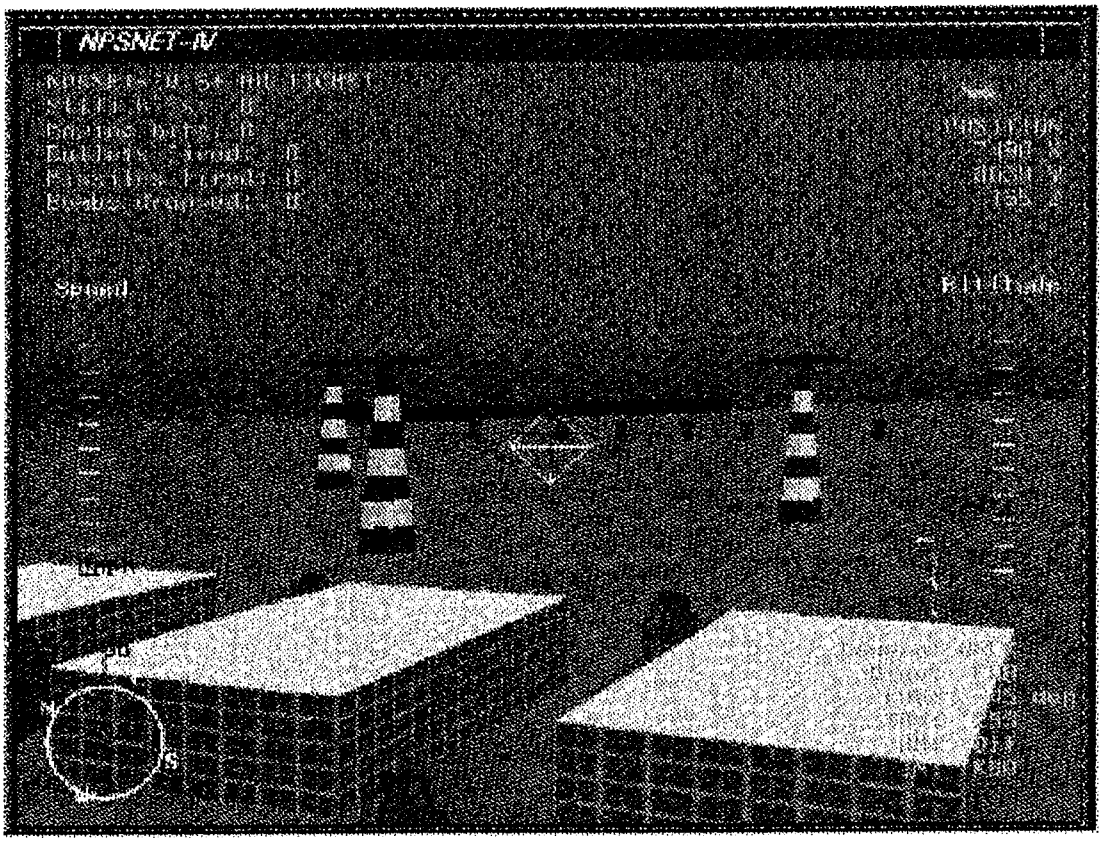


Figure 1: NPSNET -- Three Dimensional Simulation System

research new behaviors for autonomous agents. A more detailed description of the ModSAF system is included in Chapter III.

C. AUTOMATED FORCES DEVELOPMENT

Common goals for developers of a semi-automated force include: providing a larger number of computer generated forces and types, with realistic behaviors, more realistic environments, and accomplishing this in a more economical way. [Ceranowicz, 94] Current technologies allow only the development of "semi-automated" not autonomous forces. "Semi-automated" forces implies that there is an external agent that completes the automation of the behaviors of these computer generated forces. ModSAF is built with this "man-in-the-loop" concept, having the operator of the computer generated forces add some

of the realism of their behaviors by performing the complex tasks of situational awareness, learning and adaptation to the environment. "By definition, a SAF operator is going to intervene to augment the automated decision logic of the forces he is commanding because they are not capable of a human level of operation."[Ceranowicz et al, 94]

Another problematic area of computer generated force design is whether the computer generated forces behave in a "military" manner. To a large degree, whether the computer generated forces perform in accordance to a military doctrine is based solely on the operator. In Operator Control of Behavior in ModSAF the authors state,

> "Military training is required to operate ModSAF in a tactically sound manner since it does not constrain the operator's commands to match any doctrine.... Verification, validation, and accreditation efforts for ModSAF must first be done on physical models, and automated behaviors.... Then the problem of calibrating the user interface for combatant operation can start to be addressed." [Ceranowicz et al, 94]

The ability for a computer generated unit to perform some degree of terrain analysis is imperative to modeling the realistic behaviors with respect to that terrain. In Terrain Reasoning for Reconnaissance Planning in Polygonal Terrain [Van Brackle et al, 93], the researchers developed a systematic algorithm to conduct a thorough reconnaissance of an area, with the underlying goal of identifying all enemy vehicles within that area. It included testing the path-planning algorithm against military experts. The military experts devised a path plan for a single vehicle to traverse an area of terrain, attempting to identify all enemy units within that terrain. The algorithms, likewise, conducted path planning and selected an optimized visitation route that would cover all "dead-space" within the area. The concept for their algorithm uses "important points" -- ridges, endpoints of treelines, and tree canopies -- to focus the terrain reasoning algorithm on these points and reduce the complexity of the search.

The ability to utilize cover and concealment, and include terrain analysis into the planning of a mission are essential elements of a terrain reasoning system for military mission planning [Stanzione, 89]. ModSAF provides automatic road route generation to the operator, and a degree of terrain analysis at the vehicle level. Vehicle behaviors including

finding cover and concealment have been incorporated into ModSAF unit level tasks. [Stanzione et al, 93] Currently, ModSAF offers only limited terrain analyzing algorithms, and only at the lowest vehicle levels, to accomplish the computer force missions. The idea of performing a reconnaissance of an area of terrain prior to conducting a mission in that terrain helped to formulate the assembly area mission researched in this thesis.

D. SUMMARY

The development of computer generated forces that exhibit realistic behaviors is a current and expanding field of research and design. The development of higher level command and control of autonomous agents is in its infancy. Several shortcomings of the currently fielded SAF systems have been analyzed by the Defense Modeling and Simulation Office (DMSO). In their 1993 DMSO Survey of Semi-Automated Forces it is stated:

> "While SAFOR provides a convenient and perhaps cost-effective way to play many vehicles on the virtual battlefield, the experiments cited in this chapter highlight several immediate deficiencies. The operator must now control the units at a low level using time-consuming mouse and menu interfaces. The automated rules governing the units' behaviors, and especially reactions to enemy actions, are not sophisticated enough to support totally automated execution.... The lack of higher level control effectively limits the size of exercise which one operator may run, or requires more SAFOR operators and equipment." [Booker et al, 93]

The purpose of this thesis was to research, design, and implement a company level mission that includes degrees of terrain analysis and computer generated force mission planning. It uses ModSAF as the simulation platform and establishes, by proof-of-concept, that higher level missions can be incorporated into ModSAF.

III. ModSAF DESCRIPTION

A. DESCRIPTION

An excellent overview of the ModSAF system can be found in the papers by Calder, Smith, Courtemanche, Mar and Ceranowicz, ModSAF Behavior Simulation and Control [Calder et al, 93], and Andrew Z. Ceranowicz, ModSAF and Command and Control [Ceranowicz, 93-2]. A basic overview of the ModSAF system and some of the particular ModSAF terms are defined in this chapter to aid in understanding the development of the company level mission "Assembly Area."

1. Introduction

The Modular Semi-Automated Forces (ModSAF) system is a Distributed Interactive Simulation (DIS) system that portrays Computer Generated Forces (CGF) with realistic individual and unit behaviors. The system is sponsored by the Defense Advanced Research Projects Agency (DARPA) WISSARD (What If Simulation System for Advanced Research and Development) project. ModSAF has become the standardized simulation platform for continued research in the use of Computer Generated Forces by the U.S. Army Simulation, Training, and Instrumentation Command's (STRICOM) Advanced Distributed Simulation Technology (ADST) program. ModSAF is a modular implementation of previous work conducted in SIMNET under DARPA and ODIN. [Calder et al, 93].

2. Overview

ModSAF is an object oriented design implemented in standard Kernigan & Ritchie C code. It consists of over 150 library modules and 215,000 lines of code. It runs on SGI, Sun, MIPS, and IBM RISC 6000 Hardware systems [Robasky, 94-1].

The architecture for the system includes three components: The ModSAF Command Workstation (SAFstation), the ModSAF Simulator (SAFsim), and the ModSAF Logger. The SAFstation provides the graphical user interface to the operator. The operator can create and place units, assign missions, and observe the execution of the units from this

station. The operator is given a two dimensional view of the terrain database on which the exercise is being simulated (Figure 2). The SAFsim simulates the vehicles and units created by the operator. It provides the realistic behaviors to the entities. The Logger is a recorder that records the states of the Persistent Object Database and the DIS database, and can replay exercises [Ceranowicz, 93-2]..

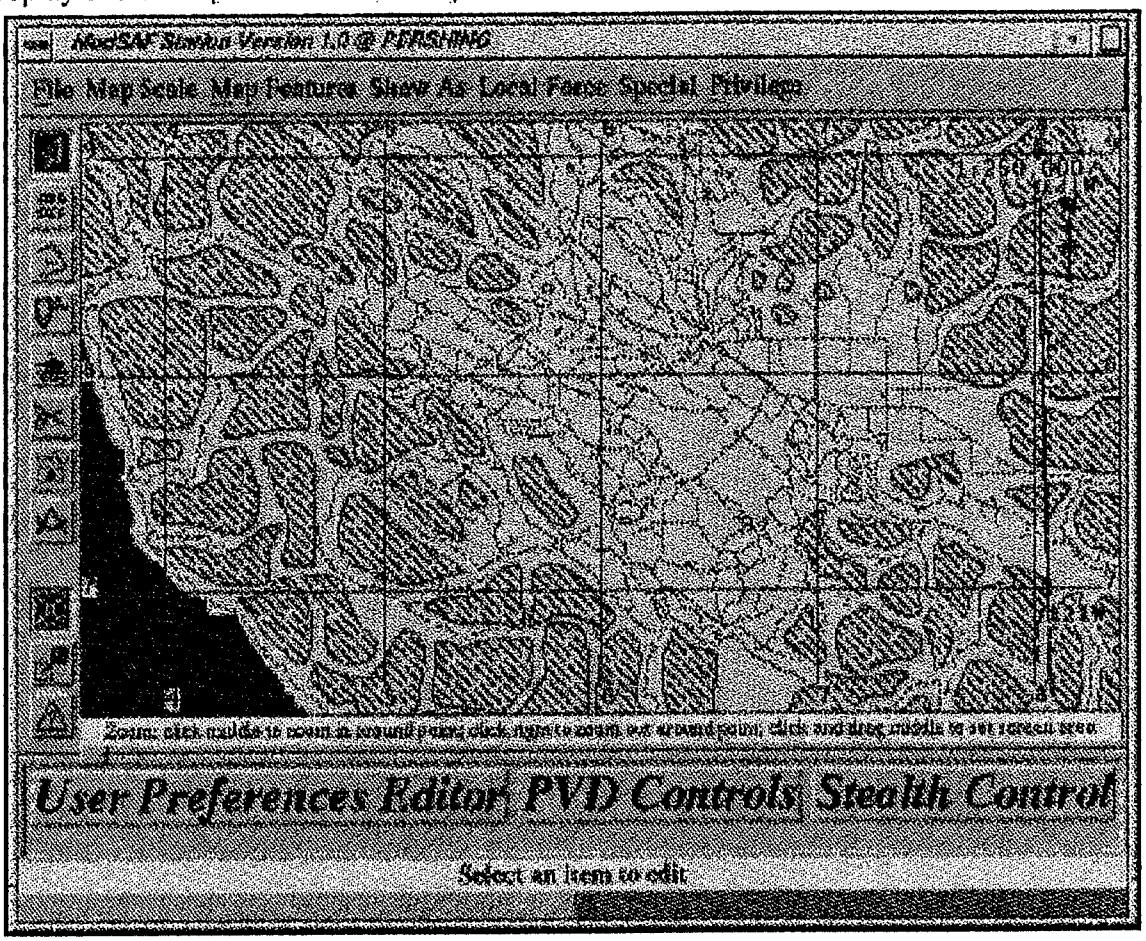


Figure 2: ModSAF GUI SAFStation

3. Communications

The Distributed Interactive System concept is defined in STRICOM's Proposed IEEE Standard Draft, Standard for Information Technology - Protocols for Distributed Interactive Simulation Applications [Standard, 93]. "DIS is a time and space coherent synthetic representation of world environments designed for linking the interactive, free play activities of people in operational exercises." [Standard, 93]. In a very general sense,

computers can share information about the world (or terrain) and the positions and activities of the entities in the world, with each computer portraying the world and all its entities simultaneously. The DIS Protocol is a standardized format to package the necessary information about the entities being displayed in a DIS environment. ModSAF communicates with other computers on the simulation by sending and receiving DIS Protocol Data Units (PDUs).

Whereas the DIS protocol provides a way to share the physical state of the world between computers, ModSAF has its own Persistent Object (PO) Database to retain information about the entities it is simulating. The PO Database keeps track of information about a unit including the unit's current mission and status, the unit's organization, and the individual vehicle information for each vehicle in the unit. The ModSAF computers share command and control and system information via the Persistent Object (PO) Protocol [Ceranowicz, 93-2].

ModSAF maintains two databases; the DIS Database, and the PO Database. The DIS Database is a conceptual database, used to share information between ModSAF and external DIS simulation systems. The PO Database stores internal information about the world and its entities. (Figure 3).

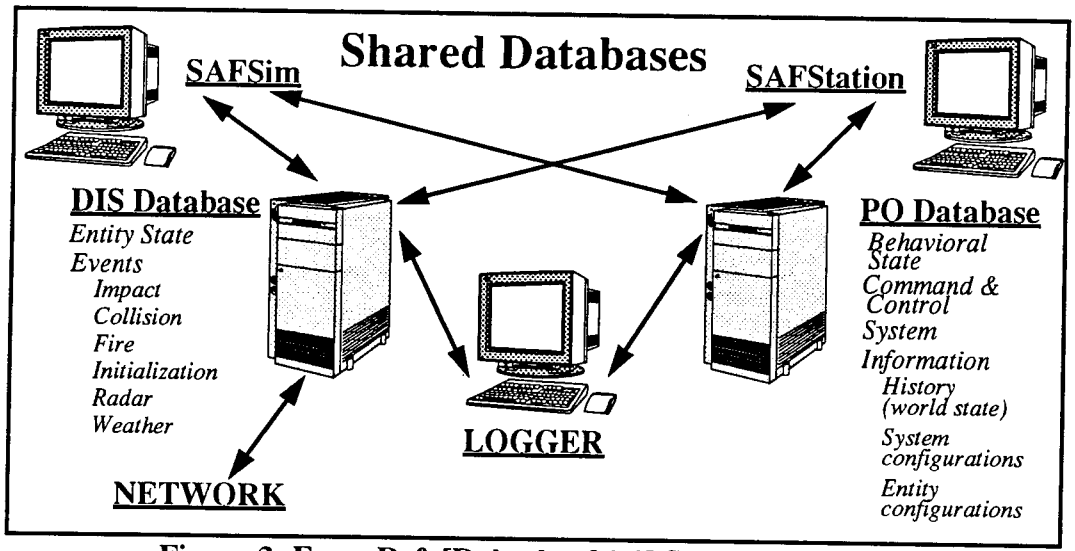


Figure 3: From Ref. [Robasky, 94-1] Shared Databases

B. ModSAF SOFTWARE ARCHITECTURE

Since ModSAF was written in Kernigan & Ritchie C code, it is not object-oriented in the purest sense of the definition. The ModSAF system replicates the behavior of classes and methods offered in C++ by utilizing layering and object-based programming techniques.

> "Layering is a design methodology in which software modules are grouped into layers, and software in one layer is restricted to use only functions and services available in lower layers.... Object-based programming techniques are used to cleanly separate the subsystems or modules into classes of objects. Each object class is defined by a data structure and a family of functions which operate on that data structure." [Calder et al, 93].

The category of software modules that the company assembly mission utilizes falls into the set of simulation modules. This set of modules provides the behaviors for the ModSAF entitities. Figure 4 lists some of the several different ModSAF library modules. The first set of modules are behavioral modules. These library modules provide the behaviors for a vehicle or the components of a vehicle. The turret, hull, and guns of a tank are each controlled by an individual behavior module. Libraries prefixed with "libv" are vehicle simulation modules. These modules perform task actions for individual vehicles. Libraries prefixed with "libu" are unit simulation modules. These modules provide the task

libgenturret libmissile libturrets	libguns libmlauncher	libhulls libradar	libifdam libtracked
libuactcontact libubingofuel libuenemy libumount libutargeter	libuassault libucap libuflwrte libuoccpos libutraveling	libuataint libucommit libuflyrte libuoverwatchmove	libuatgrndtrgt libudsmnt libuhalt libupoccpos
libvassess libvcollide libvflygrndavoid libvmount libvspotter libvterrain	libvataint libvecmat libview libvmove libvtab	libvatgrndtrgt libvenemy libvisual libvorbit libvtakeoff	libvcap libvflwrte libvland libvsearch libvtargeter

Figure 4: ModSAF Library Modules

actions for entire units. Of particular interest are the libraries used to formulate the company assembly area mission: libupoccpos (**lib**rary **u**nit **p**repare to **occ**upy **pos**ition) and libutravel (**lib**rary **u**nit **travel**ling). These two unit task libraries utilize vehicle task libraries, and the vehicle task libraries, in turn, use behavioral libraries. This is accomplished by the layering and object-based programming techniques of the ModSAF system.

C. ModSAF COMMAND AND CONTROL

The architecture selected to replicate command and control of the computer generated forces provides capabilities, identified as command and control goals, to the developer. These goals, as discussed in [Calder et al, 93], include:

- The capability to create complex missions including preplanned contingency operations,
- The altering of a mission after assignment,
- An operator's ability to override the simulation at any time for any unit,
- A defined architecture for unit and individual behaviors,
- A graphical user interface with available missions,
- A defined structure to explain unit and individual behaviors to the user.

The company assembly area mission was designed to exploit the capabilities of ModSAF to fulfill these command and control goals.

1. ModSAF Objects and Entities

An object in ModSAF is stored in the PO Database. Objects may include graphical control measures for a particular unit, individual vehicles, or entire units. Objects which are simulated by the SAFsim are termed ModSAF entities.

> "When a SAFsim simulates a unit, the SAFsim not only creates the SAF entities (such as a plane) in a unit but also builds a structure corresponding to the unit hierarchy. The user can then issue commands to the top-level units or drop down the chain of command to give orders to subordinate units or vehicles. The SAFsim interprets these orders and then generates the appropriate unit and vehicle behavior and tactics without further action from the user. However, the user can override or interrupt any automated behavior." [ModSAF, 94]

The behavior of units and vehicles is controlled by "tasks" and "task frames". "A task is a behavior performed by a ModSAF entity or unit on the battlefield.... Task frames group a collection of related tasks that run at the same time." [ModSAF, 94] A more detailed description of tasks and task frames is given in the following sections.

2. Tasks

> "The foundation of the ModSAF command and control architecture is the concept of a task. Most tasks are behaviors performed by units or individuals on the battlefield, and are used by ModSAF to model the information processing done by its simulated entities.... There are five types of tasks which are implemented in the ModSAF system: unit tasks, individual vehicle tasks, reactive tasks, enabling tasks, and arbitration tasks." [Calder et al, 93]

An individual vehicle task controls the lowest-level actuators of a vehicle. These actuators control the simulated capabilities of checking intervisibility, target detection, target identification, target selection, fire planning, collision avoidance, and detection. [ModSAF, 94] The vehicle tasks take inputs from its sensors and other actuators and produce commands for the physical actuators. [Ceranowicz, 93-2].

A unit task encapsulates the behaviors of a unit and its individual vehicles for a particular task. For example, in the ModSAF unit task UTravel (a unit travelling task), the task issues individual vehicle tasks to each member of the platoon, and monitors the collective status of the unit's movement. Changes to the current situation or parameter changes by the operator are handled by the unit task, which may introduce new vehicle tasks, or terminate individual vehicle tasks that have ended. The concept of issuing lower-level tasks from the unit level task models the military command and control structure. [Ceranowicz, 93-2] For example, A platoon leader receives an order from his company commander to move to a location. The platoon leader develops a plan that will collectively get his unit to that location in a given formation. He then issues the orders to the individual vehicles to accomplish this mission. ModSAF replicates this command architecture through the use of unit level tasks. The Assembly Area mission is designed as a unit level task.

A reactive task is triggered in response to a change in the environment. Similar to a platoon drill, the reactive task is a pre-defined set of reactions that a unit or vehicle will implement in response to a specific environmental change. For example, when assigning a unit a move task, the operator can

set specific parameters of how to react to an enemy force. The reactive tasks for "Actions on Contact" can be set by the operator as parametric inputs. Figure 5 shows the graphical user interface for the Actions on Contact parametric entries. The operator can select the enemy vehicle thresholds and the resulting action by the unit.

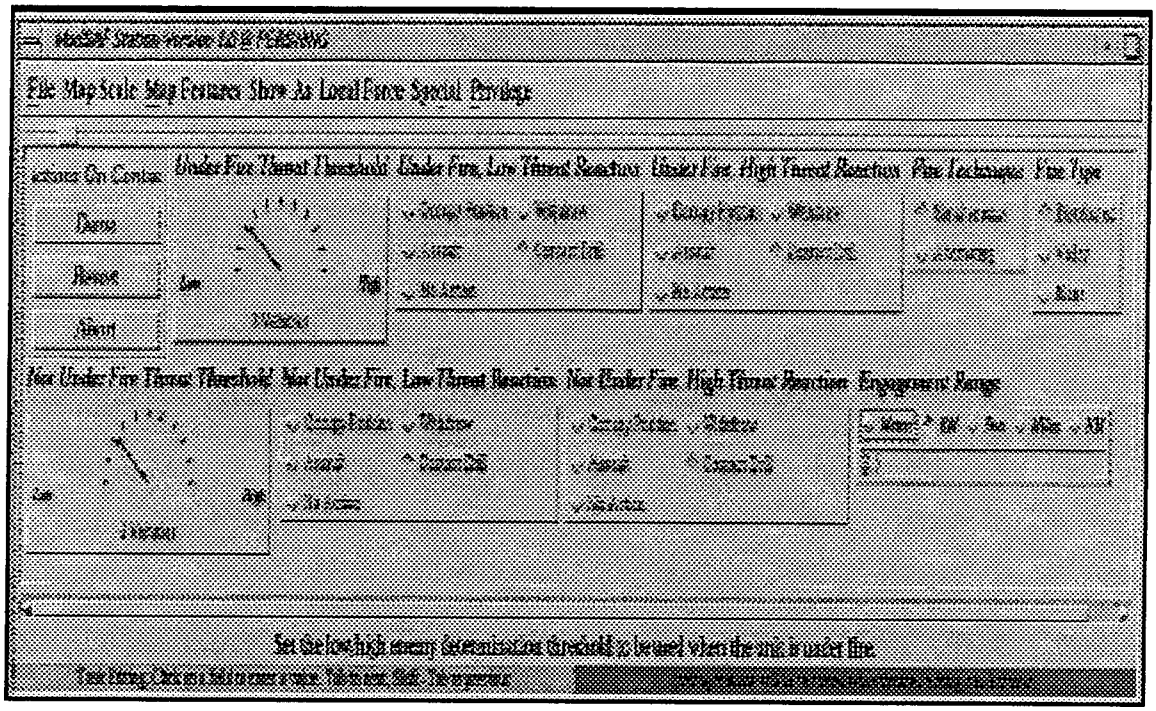


Figure 5: ModSAF Actions on Contact

Suppose the operator decides that if the unit comes under fire by less than three vehicles it should assault them, but if there are three vehicles or more it should occupy a defensive position. Also, if the unit is not being fired upon, and there are more than three enemy vehicles, conduct a contact drill. If there are less than three enemy vehicles do

nothing. These decisions can be input into the parametric entry graphical user interface shown above in Figure 5. The results of the operators inputs are shown in Table 1

Enemy Vehicles \ Situation	Under Fire	Not Under Fire
Less Than Three Vehicles	Assault	No Action
Three or More Vehicles	Occupy Position	Contact Drill

Table 1: Action on Contact

Enabling tasks link the execution of the task frames. They are defined within the mission. Example enabling tasks include continue (continue with the next task frame when the current one ends), on order (after the previous task frame ends, halt execution until the operator says to continue), control measure (when a unit hits a graphical control measure change task frames), or at a certain time execute a task frame. Enabling tasks give the unit alternative actions to take in response to events the mission developer has foreseen when developing the mission. [Calder et al, 93]

Arbitration tasks resolve multiple recommendations for what specific action a unit should execute. These tasks are discussed in more detail in the paragraph Task Arbitration, page 18.

3. Task Frames

Related tasks which run concurrently to accomplish an action are termed "task frames". A task frame represents a phase of a mission and are defined with associated parameters. Some of these parameters may be adjusted by the operator to modify the behavior during the frame. [Ceranowicz, 93-2] [Calder et al, 93]. For example, a platoon conducting a move operation is operating within a task frame. The task frame includes the movement task and the reactive tasks for "Actions on Contact". The reactive parameters may be set by the operator when assigning the move task frame. As the move task is operating, the platoon may encounter an enemy unit. The move task will be pushed on a

stack while the reactionary task, contact drill, assault, or occupy position, executes. The operator can stop, change, or override the reaction of the unit during the execution of the move task frame. "Task frames are typically composed of move, shoot, and react tasks." [ModSAF, 94]

4. Missions

A sequence of task frames collectively form a mission. Before the next frame begins, the previous frame must have ended. An example mission which includes several separate tasks would be a platoon to move along a route, attack an objective, and then move to another location and occupy a position. The attack on the objective will not begin until the movement along the route has ended. [Calder et al, 93]

ModSAF provides platoon level missions for ground vehicle platoons and dismounted infantry platoons. Some platoon missions may be assigned to a single vehicle. These missions include: "Move", "Follow a Vehicle", "Occupy Position", and "Assault". One mission written solely for a ground vehicle platoon is "Bounding Overwatch." In a Bounding Overwatch mission, the platoon is split into two sections. One section moves and the other section stops to cover its movement. Assigning this mission to a single vehicle would not be consistent with the mission parameters and logic. The dismounted platoon missions include "Mount" and "Dismount." A platoon must be a dismounted infantry platoon to perform these missions.

5. Task Manager

The collection of tasks and task frames which combine to form missions are controlled by the ModSAF Task Manager. The task manager maintains information about prerequisite tasks and follow-on tasks for every mission. For example, a helicopter is given a move mission. A prerequisite task would be to "take-off" and a follow-on task may be to "land". The task manager maintains these task dependencies and develops a task execution list that considers the required before and after tasks along with the specified task. It then executes

the tasks in the task execution list order. The task manager also handles the task frame management for the unit and vehicle tasks. [Calder et al, 93]

6. Task Arbitration

Often, more than one task is operating at the same time. These tasks may send different commands to the vehicle. In the previous example of a platoon performing a movement that encounters an enemy force, the move task is setting the move path for each individual vehicle. Suddenly, a reactive "Action Drill" is initiated in response to making contact with the enemy. The "Action Drill" task will also give movement paths to each of the individual vehicles. The vehicles are given two possibly different paths to follow, and some method of deconfliction is needed. It is the responsibility of the Task Arbitrator to decide which movement path to utilize for each vehicle. The Task Arbitrator takes all recommendations for the control of a particular actuator (movement, fire control, sensors) and then decides, based on a priority scheme, which task will control that actuator. [Calder et al, 93]

D. FINITE STATE MACHINES

ModSAF's unit tasks, vehicle tasks, and some behavioral tasks are implemented using an "Augmented Asynchronous Finite State Machine" format. The tasks are developed by separating them into states of a finite state machine. Once the finite state machine (FSM) is coded, a "finite state machine to C code" conversion utility is called to convert the FSM to standard Kernigan & Ritchie C code. The format is asynchronous in that units may generate outputs in response to a particular event or group of events, and is augmented in that not only does it keep track of the state of a unit/entity, but maintains additional information (in the PO database) of other private variables besides just the state variables. [Robasky, 94-1]

E. INTERACTION WITH ModSAF

The ModSAF system can be easily modified and extended by others to support multiple behavioral representations and multiple levels of command and control. There are

several different ways to interact with ModSAF. A developer can change existing software modules or add new software modules, replace entire subsystems, or write separate programs that communicate with ModSAF through the Persistent Object Database. [Calder et al, 93] The Company Assembly Area mission was developed by changing existing software modules, and by creating a new software module, the *unit assembly* library.

IV. ASSEMBLY AREA

The purpose of this chapter is to provide an overview of the military mission "Assembly Area." It discusses the subtasks required for the mission, and a general overview of the company's actions during the execution of this mission.

A. COMPANY TASK ORGANIZATION

ModSAF provides several unit choices to utilize for a mission. Some of the standard sizes for unit selections include a single vehicle -- M1, a platoon of vehicles -- M1 platoon, or an entire company of vehicles -- M1 company. The standard M1 Company selection includes fourteen M1 Abrams tanks, task organized as three platoons of four M1's each, with two headquarters M1 tanks, the Company Commander's and the Executive Officer's tanks. The assembly area mission required the addition of one more vehicle. The addition of an M998 utility truck, utilized by the First Sergeant of an M1 Company, completes the company task organization for the unit. The task organization for the company is shown below in Figure 6. The company includes, from left to right: 7 - the First Sergeant's utility truck, 66 - the Company Commander, 65 - the Company Executive Officer, the first platoon (expanded to show the individual tanks 11, 12, 13, and 14), the second platoon, and the third platoon.

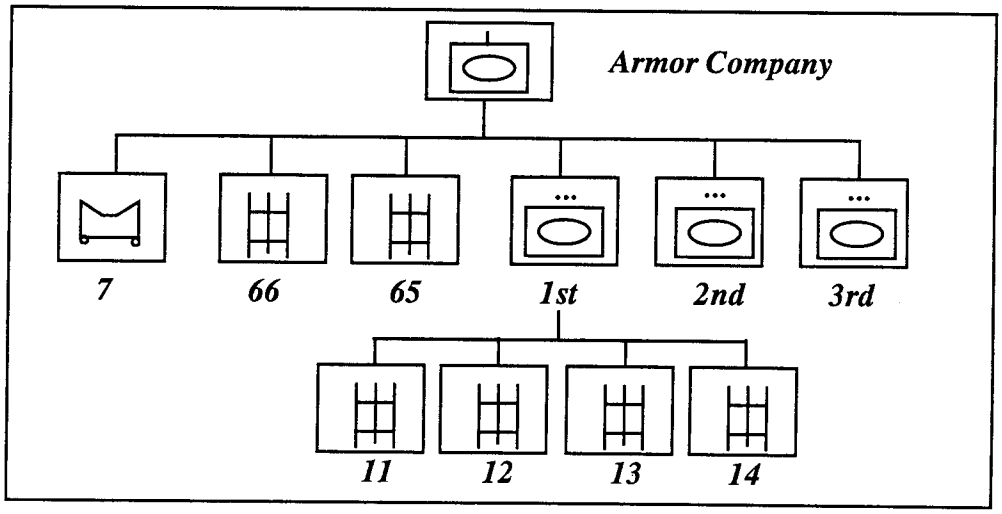


Figure 6: Company Task Organization

Adding the First Sergeant (1SG) into the task organization of the company allows the First Sergeant to be handled as an independent unit, and be assigned tasks while the remainder of the company conducts previously assigned missions. The First Sergeant, acting independently, can conduct his reconnaissance and identify the unit routes for the assembly area.

B. OVERVIEW OF ASSEMBLY AREA MISSION

1. Assembly Area Parameters

The assembly area mission requires the human operator to provide a minimum of two input parameters: the desired center of mass location for the assembly area, and the direction to the nearest enemy unit. The location determines the center of mass of a three kilometer by three kilometer search area in which the 1SG will conduct his reconnaissance. The direction to the enemy location provides a reference for the 1SG to select routes to the assembly area location, as well as routes into and out of the assembly area itself.

2. Overview of Mission

The tactical assembly area is utilized when a unit remains in one place for an extended time period, preparing for future operations. U.S. Army Field Manual, FM 71-1, *Tank and Mechanized Infantry Company Team*, outlines some of the tactical requirements for the assembly area mission. In accordance with FM 71-1,

> "An assembly area is used to prepare for future operations. While not a battle position, it should be located on easily defensible terrain, and should be planned like a defensive position. The company team may occupy part of a task force assembly area. Normally, the company team will be assigned a separate area.... If alone... the company team must establish 360-degree security. A well-planned assembly area will have concealment, good routes in and out, security from ground and air attack and observation, and good drainage." [USA FM 71-1, 88]

While occupying an assembly area, a company will conduct some of the following missions: issue orders for upcoming operations, organize for a mission, perform maintenance on its vehicles, rehearse the next mission, eat and rest [USA FM 71-1, 88].

3. Step-by-Step Assembly Area Procedures

A summarized listing of the specific tasks and subtasks for the assembly area mission are given in [USA ARTEP 71-1-MTP, 88] (Figure 7).

Task: Occupy an Assembly Area.

Condition: The company team is ordered to move and occupy an assembly area in preparation for combat operations. The enemy may attack with indirect fire, Close Air Support (CAS), and platoon-sized forces.

Task Standard: The quartering party completes the preparation of the assembly area and guides the company team into their positions not later than the time specified in the operations order. The company team main body is not surprised by the enemy. The company team completes its preparation for future combat operations.

Subtasks:

1. The company team organizes a quartering party.
2. Personnel get essential equipment.
3. The quartering party prepares for movement.
4. The Non-Commissioned Officer in Charge (NCOIC) briefs the quartering party on specific instructions for occupying the assembly area.
5. The quartering party moves along the route of march.
6. The quartering party moves into the company team assembly area and prepares the area for the company team's arrival.
7. The company team moves to and occupies the assembly area.
8. The company team performs assembly area actions.
9. The commander coordinates with adjacent units.
10. The company team performs perimeter defense.
11. On order, the company prepares to leave the assembly area.

Figure 7: Occupy Assembly Area

The deliberate occupation of an assembly area includes a reconnaissance of the assembly area site, the establishment of quartering parties to assist in the occupation of the area, the movement to the assembly area location, and the establishing of the platoon and headquarters vehicle occupation areas. After the unit arrives in the assembly area, there are

several additional tasks that must be accomplished. FM 71-1 provides a listing that summarizes these tasks (Figure 8). [USA FM 71-1, 88]

Assembly Area Tasks

- Move vehicles from Release Point (RP) into assembly area without stopping.
- Guard and cover entrances and exits with weapon systems.
- Cover avenues of approach with weapon systems.
- Achieve mutual platoon and vehicle support.
- Establish company and platoon fire plans.
- Coordinate with flank units to interlock fires.
- Camouflage vehicles from ground and air detection.
- Send platoon status reports to company.
- Send company team status report to higher commander.
- Rearm, resupply, and refuel.
- Perform equipment maintenance.
- Establish hot loop for communications.
- Issue orders.
- Establish dismount points.
- Enforce noise and light discipline.
- Organize and conduct security patrols and observation points (OPs).
- If hot loop cannot be established, maintain radio watch.
- Secure company trains area in the company position.
- Establish a command post.
- Deploy chemical detection equipment.
- Reconnoiter exit routes in case of artillery or ground attack.
- Designate rally point.
- Designate possible alternate assembly area.
- Disseminate challenge and password.

Figure 8: Assembly Area Task List

4. Cover and Concealment

The assembly area mission will attempt to maximize a units cover and concealment from the enemy. A covered position provides security from the enemy's direct fire weapons. Positioning a vehicle behind a hill mass provides cover from the enemy's direct fire weapons. Concealment does not protect a vehicle from direct fire weapons, but

prevents enemy detection. A tree-line or tree canopy provides concealment from enemy ground and air vehicles, but may not provide adequate cover from direct fire weapons. Examples of cover and concealment are illustrated below in Figures 9 - 11.

Figure 9 depicts an M1 tank in the open. The vehicle is neither covered nor concealed from the enemy. A vehicle is most vulnerable to the enemy at this time.

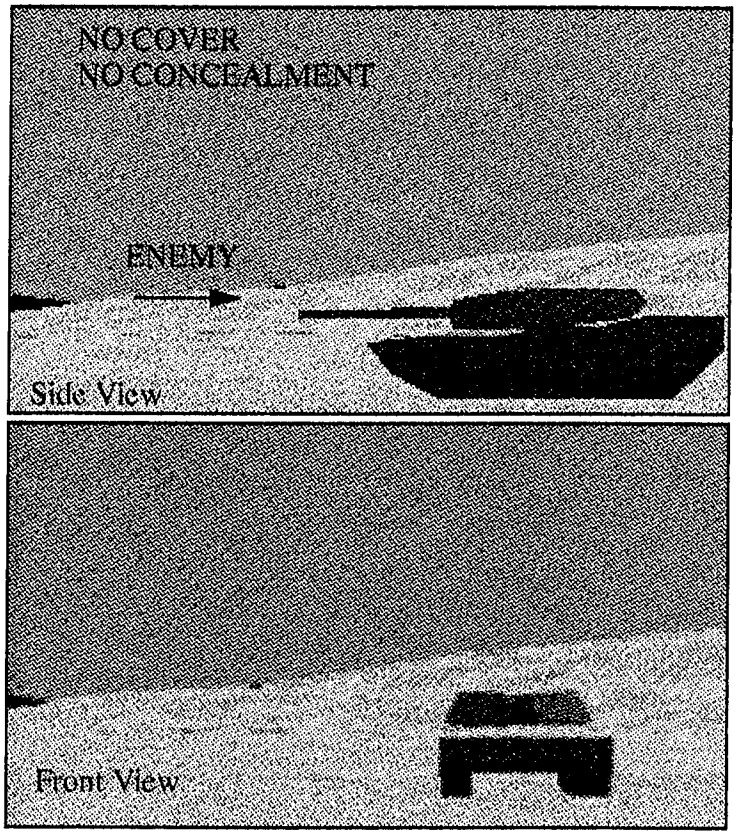


Figure 9: No Cover, No Concealment

Figure 10 depicts a tank with limited cover and limited concealment positioned behind a small berm. This position is referred to as a "Hull-Down" position. This is the position the tank must be in to have its gun tube clear the terrain and be capable of shooting at the enemy. The vehicle can still be detected by the enemy and can still be engaged with direct fire weapons, but since only the turret is exposed, this position provides limited cover and concealment. Concealment from air vehicles is still a problem.

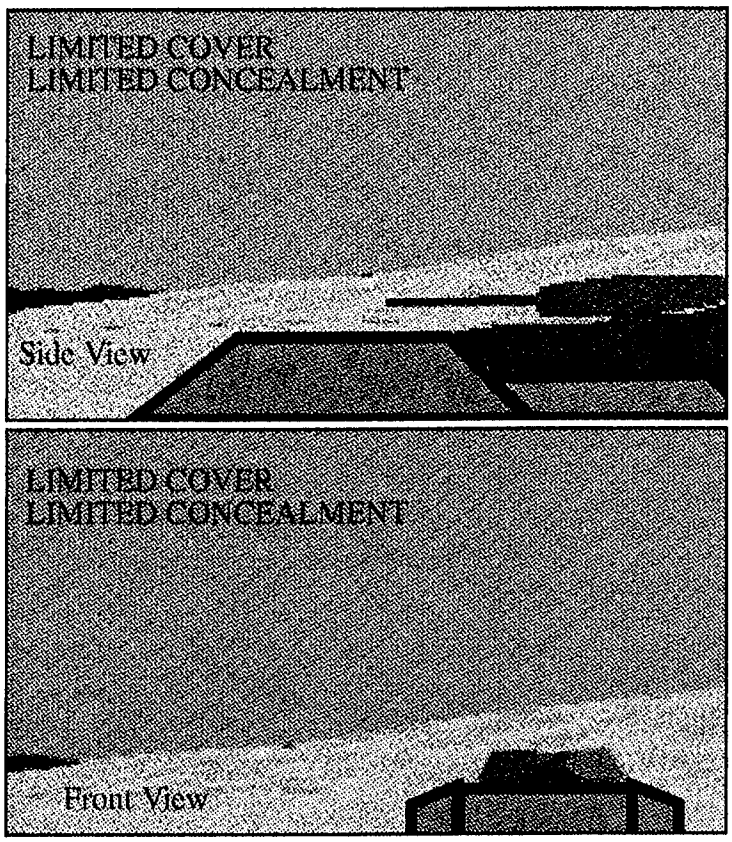


Figure 10: Limited Cover, Limited Concealment

Figure 11 depicts a tank in a "Turret-Down" position. From this position, a tank crew can observe the enemy with their optical sights, but cannot fire at the enemy. Cover is provided by the intervening terrain between our vehicle and the enemy vehicle. Enemy detection from the air is still a major concern.

5. Mission Simplifications

For the purposes of the computer generated assembly area mission, the responsibilities of the quartering party are given to one key player, the company first sergeant. The First Sergeant, once given a tentative location for setting-up an assembly area, must do several things. First, he performs a map reconnaissance. He looks for tentative locations that may provide cover and concealment from both enemy ground and air vehicles. Ideally, the company is provided both cover and concealment, but concealment is especially important

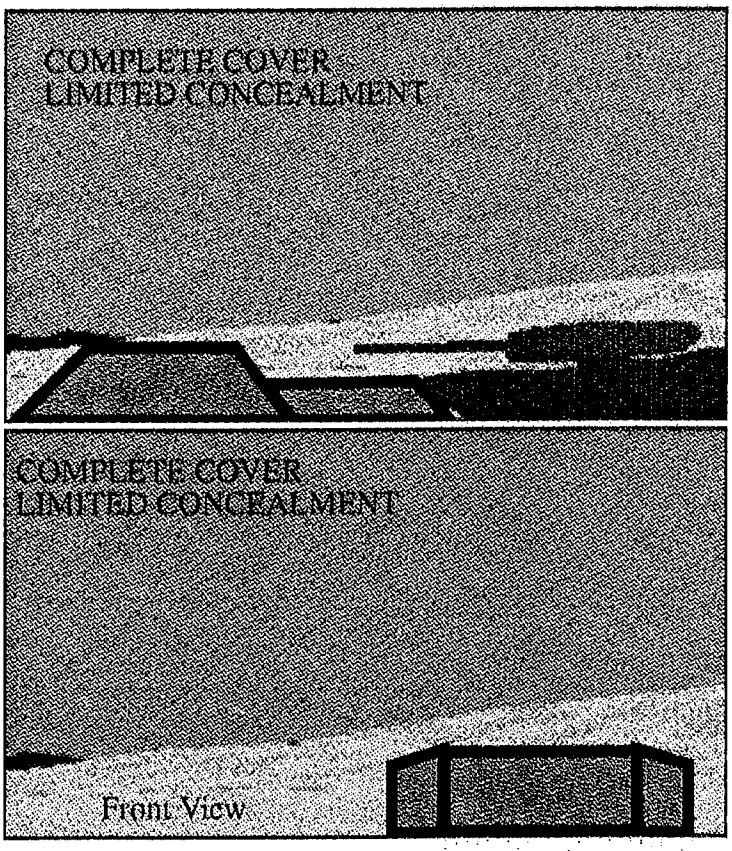


Figure 11: Covered, Limited Concealment

since the assembly area location is normally not within direct fire of the enemy. The First Sergeant can quickly identify some possible locations providing possible tree canopy concealment from a standard defense mapping agency military map. At a standard map scale of 1:50,000 (one inch on the map represents 50,000 inches on the ground), the first sergeant can identify some potential locations for the assembly area. Identifying tree canopies that provide adequate room for a company team assembly area can be accomplished. If there are no canopies providing adequate cover, he may look for a land feature that may provide cover from the direction of the enemy. At this scale, however, terrain irregularities and detailed gradient information may be lacking. It is not until the First Sergeant arrives at the tentative areas that he can adequately analyze the terrain.

For example, consider Figure 12 below. At a scale of 1:50,000, there are five canopy regions that may be considered for assembly area locations (numbered 1 to 5).

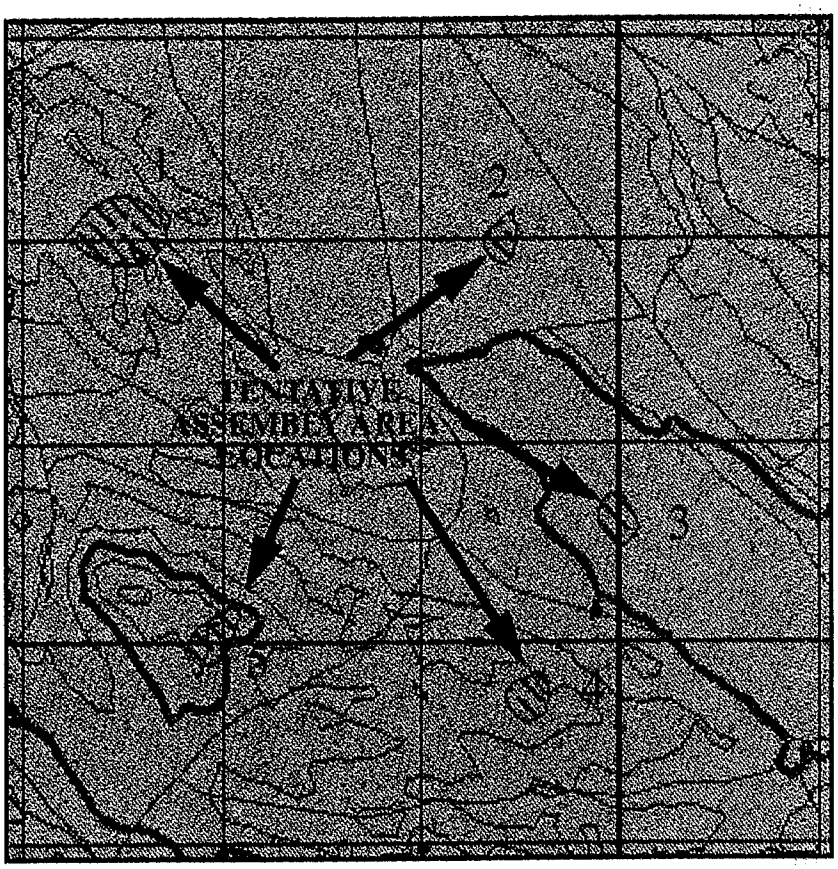


Figure 12: Ft. Hunter Ligget, 1:50,000

Canopy five appears to be on the side of a hill. It may be too steep in gradient to be considered as an acceptable location. Canopies one through four appear to be both large enough and not too steep. When we look at canopy one, at a scale of 1:5000, however, we see that there are a number of areas where the gradient exceeds a ten percent grade, and some areas that have a nineteen percent grade (Figure 13). This information will not be identified by the First Sergeant until he reaches the tentative assembly area location.

There are times when there are no canopies or trees available for concealment. At the National Training Center (NTC) at Fort Irwin, California, assembly areas must make use of terrain features other than trees and canopies. Figure 14 depicts a possible assembly area location that has mountainous terrain on three sides, providing cover and limited

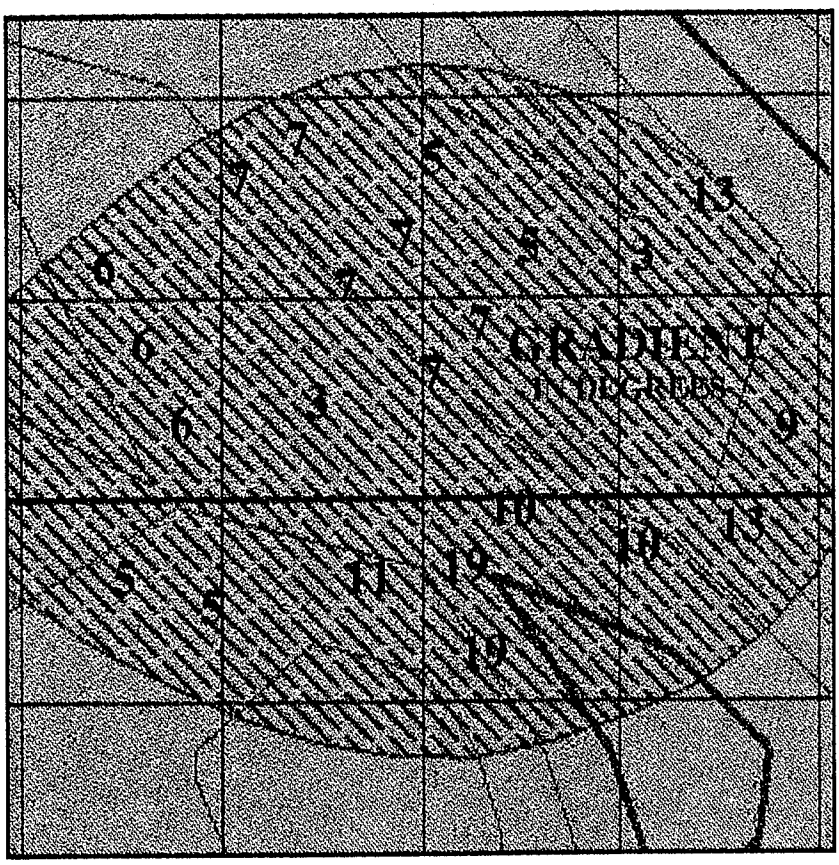


Figure 13: Ft. Hunter Ligget, 1:5,000 Scale

concealment. Here there are few regions where canopies of trees are available to provide concealment. The unit must use the terrain to provide cover and concealment from the enemy.

C. SUBORDINATE LEVEL MISSIONS

1. First Sergeant Tasks

The First Sergeant is given a location to establish an assembly area. He uses this location to establish the boundaries of a search area. The assembly area mission uses a two kilometer by two kilometer square area for this search area. Once the First Sergeant determines the search area, he selects tentative assembly area locations as if he were using a 1:50,000 scale map. He then plans a road route (if one is available) that will end in the search area. Once he arrives in the search area, he selects one of the tentative assembly area

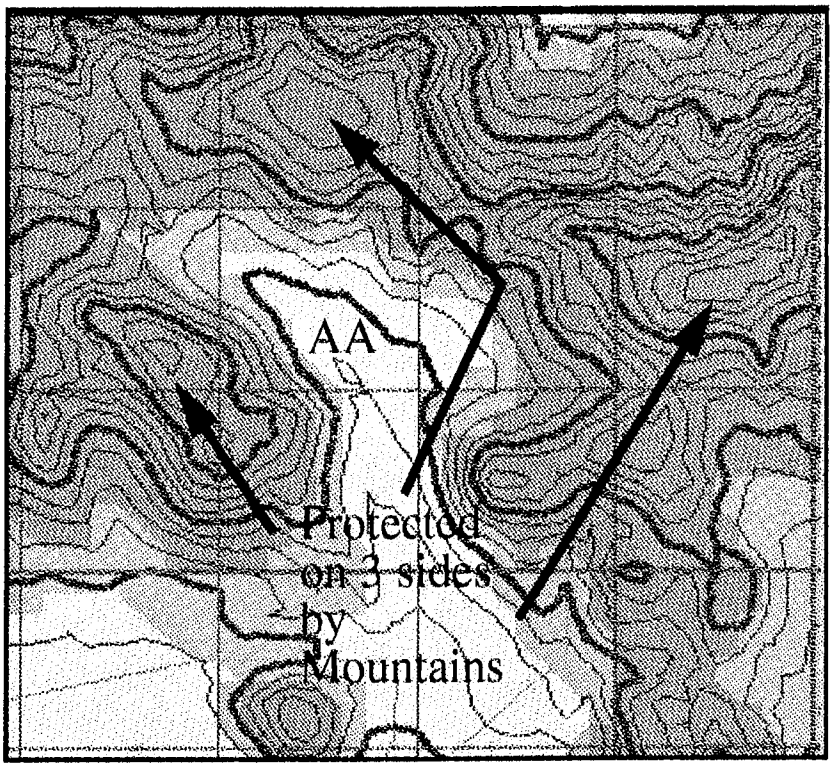


Figure 14: NTC, Ft. Irwin, 1:50,000 Scale

locations, and plans a road route to get to that location. If the tentative assembly area location is suitable in both size and gradient, he establishes this location as the company objective. He then plans a company road march to get the company to the assembly area location. The company road march includes a start point (SP), a release point (RP), and the route itself. These control measures are used by the company commander to control the movement of the company during the move to the assembly area. The company road march plan is sent to the company commander, who prepares the unit for the march. The First Sergeant then moves to the release point, the last road point of the company road march, and awaits the arrival of the company.

2. Company Tasks

The company (minus the First Sergeant) continues its previously assigned mission. When the First Sergeant has identified a suitable assembly area location, the company prepares for the road march to the assembly area. The company commander designates the

route, SP, and RP, along with an order of march. This order of march specifies the order the platoons and headquarters tanks will use during the execution of the company road march. The company then conducts the road march, each unit reporting when they have arrived at the SP and the RP. When a unit arrives at the RP, it moves into the assembly area location and occupies its designated locations.

3. Platoon Tasks

Each platoon, as part of the company road march, conducts its own individual platoon road march. Platoons begin their march in accordance with the company order of march, and report in all control measures to the company commander. In addition to the platoon road march, there are several tasks that a platoon continuously performs. These tasks are reactive in nature and are termed platoon drills. Platoon drills are conducted in response to specific changes in the environment. Environmental changes that may cause the execution of a platoon drill include: sighting enemy ground vehicles or air vehicles, receiving fire from enemy vehicles, or receiving indirect fire.

The first type of drill is a contact drill. A contact drill is often used when the sighted enemy is not considered a great threat to the platoon, or it is critical for the platoon to continue its movement in its original direction. A platoon performing a contact drill will fire at the enemy but continue its movement in the same direction it was originally travelling before making enemy contact.

Another drill is the action drill. An action drill is used when the sighted enemy poses a threat to the platoon. A platoon performing an action drill will return fire on the enemy and change its direction of movement to respond to the enemy. The purpose of this movement is to reorient the forward edge of the tank toward the enemy, placing the maximum armor toward the threat. The platoon will continue to move toward the enemy until it determines there is no longer a threat.

V. DESIGN STRATEGY

This chapter outlines the design considerations for the development of the company level mission "Assembly Area". It includes the goals for the design and an outline of the stages of this mission. The Company Level Assembly Area Mission is uniquely different from most ModSAF unit level tasks in that the unit performs its own mission planning, including route selection and identification of the assembly area location. The intent is to free the operator from having to designate the specifics for the mission and only provide the general area for the assembly area.

A. DESIGN GOALS

ModSAF currently does not have company level tasks. The goal of this thesis is to show by proof-of-concept that we can simulate company level tasks utilizing ModSAF's asynchronous augmented finite state machine architecture. The premise is that a finite state machine abstracted to the company commander level can spawn and control existing platoon and vehicle tasks in ModSAF.

The Computer Generated Forces in ModSAF are termed "Semi-Automated" not autonomous. ModSAF behaviors at the vehicle and platoon level exhibit fairly realistic behaviors. However, it is still the responsibility of the operator to provide the realistic interactions between platoons when portraying a higher level unit, like a company or a battalion. One of the design goals was to show that the finite state machine architecture could additionally provide mission planning at the company level, issuing platoon and vehicle instructions to accomplish a mission. Not only will this approach provide more realistic behaviors at the company level, it will reduce the parametric input responsibilities of the operator, allowing him to control a greater number of forces.

The addition of a limited degree of terrain reasoning at the company level, and the ability for units to identify and create their own road routes are features that ModSAF currently does not offer. This will be discussed in more detail in Chapter VI, Design Implementation.

B. TASK ORGANIZATION

The task organization for the company assembly area mission (as outlined in Chapter IV) includes fourteen M1 tanks, and a utility truck for the First Sergeant. The First Sergeant is given the responsibilities of the quartering party. The purpose of adding the First Sergeant's vehicle was to provide a non-combat type vehicle that performs the reconnaissance and route selections for the company assembly area mission. The M1 tanks of the company continue their current missions while the First Sergeant performs the initial stages of the assembly area mission.

C. ASSEMBLY AREA MISSION STAGES

The first step in the design process was identifying a sequence of operations describing the assembly area mission. The sequence of operations used to represent the company assembly area mission are listed in Figure 15.

Assembly Area (AA) Stages:

- Identification of Assembly Area (AA) Search Location, (operator)
- Planning the Route to the AA Search Location, (1SG)
- Moving to AA Search Location, (1SG)
- Reconnaissance of Search Area, (1SG)
- Designating Unit AA Locations, (1SG)
- Route Planning for Company, (1SG)
- Company Moves to AA, (Company)
- Occupy AA, (Company).

Figure 15: Steps for Company Assembly Area

The first step is identifying the search area for the assembly area. The operator provides this information when the assembly area mission is first assigned. The First Sergeant (1SG) performs the next stages of the mission; planning a route to get to the location, and a terrain reconnaissance of the area. The final stages are performed by the

company; movement to the assembly area and occupation of their positions. Each of these actions is discussed in greater detail in the following paragraphs.

1. Identification of Assembly Area Search Location

One of the design goals requires the unit to conduct the mission planning for the assembly area task. The parametric data supplied by the operator for most ModSAF tasks specifies an end goal for a unit (Figure 17). The assembly area mission is somewhat different in its parametric input. Instead of having the operator provide a point location to establish an assembly area, we want a point location that will determine the center of mass of a search area. The unit will determine where to establish the assembly area given the bounding search region. The size of the search area was selected to be a three kilometer by three kilometer square area surrounding the operator's selected center of mass (Figure 16).

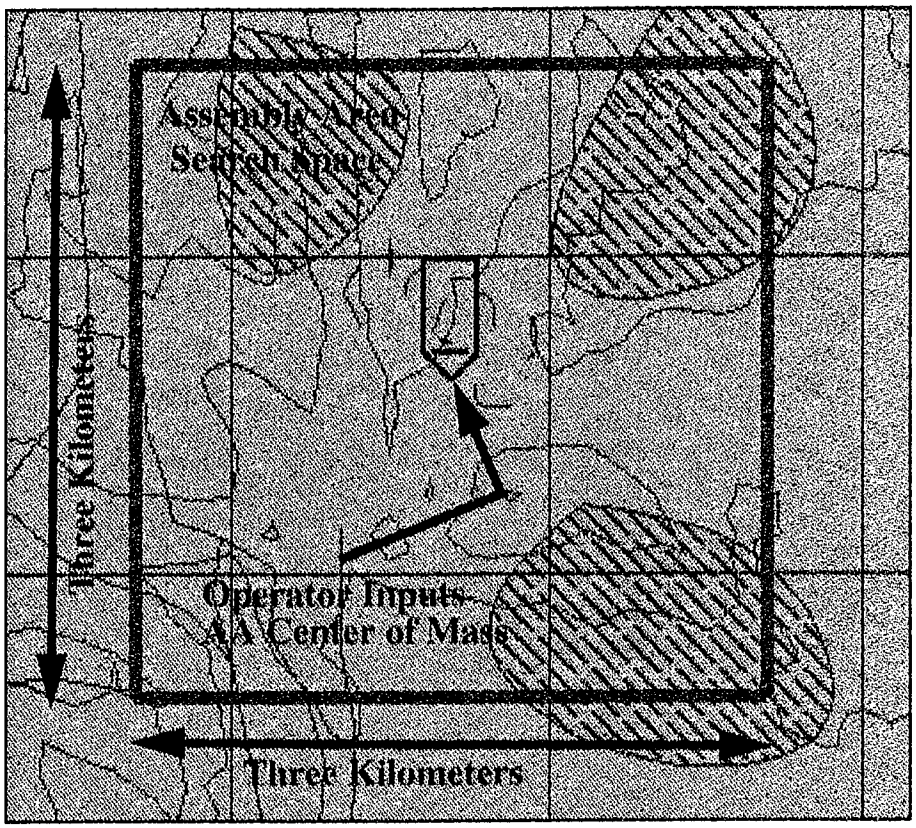


Figure 16: Assembly Area Search Space

2. Planning the Route to the Search Location

Many of the ModSAF unit and vehicle tasks -- like move, travel, and assault -- require the operator to input either a point, a line, or some text that provides the parametric input for the task. The unit then performs its task, and often the goal of the task is to move to the point or line designated by the user. Figure 17 shows an example of the options the operator has when specifying a unit's movement route. Shown are three alternative route inputs; a point location, a line route, and a line route that uses road networks. Regardless of which object the operator utilizes to designate the unit route, it is the operator that supplies the route for the unit.

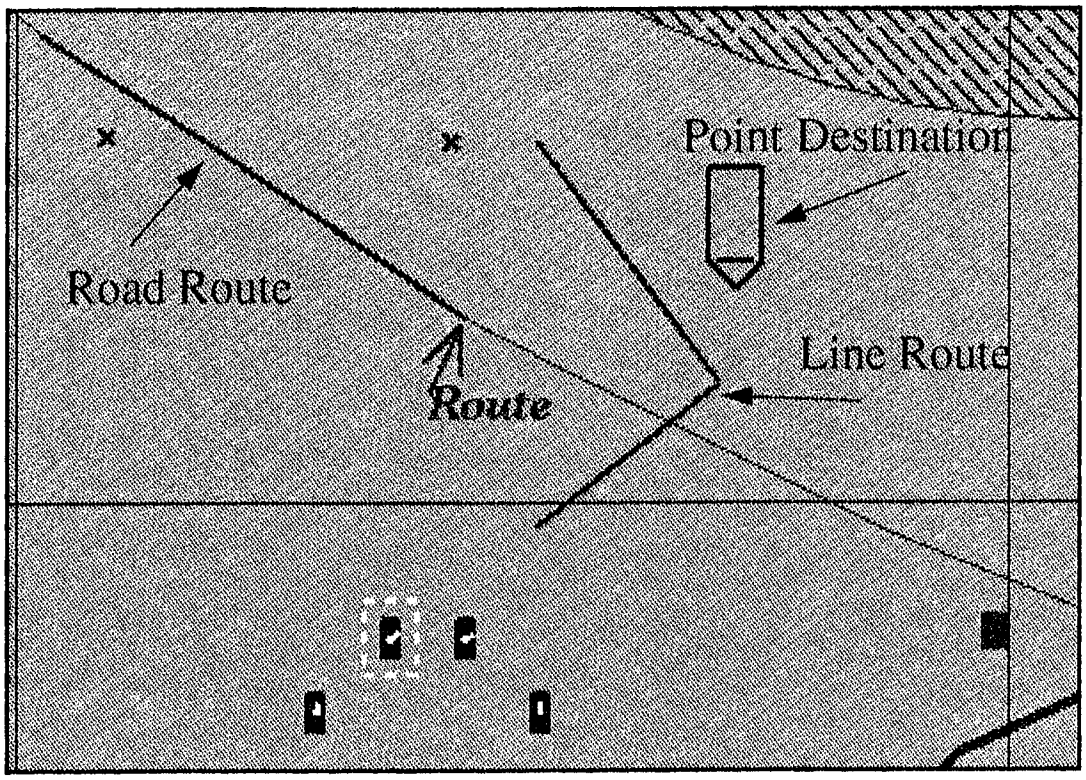


Figure 17: Unit Travel Route Selection

As part of the design strategy, the assembly area mission attempts to utilize road networks to conduct unit movements. Since it was a design goal to reduce the burden on the operator for providing parametric inputs, like providing the routes for a unit, the unit itself should plan its own route. Without modification, however, ModSAF does not provide

the functional ability for a unit to determine its own road route. It is a design strategy and goal to provide this capability to ModSAF. The utilization of the road networks is discussed more in Route Planning for the Company, page 39.

3. Moving to Assembly Area Search Location

Once the First Sergeant identifies a route to the search area, he conducts a ModSAF Vehicle Move Task to get to that location. Where the operator normally provides the parametric inputs for the Vehicle Move task, assembly area routines select the route and pass this information to the Vehicle Move task. The company (minus the 1SG) continues its assigned mission. Once the 1SG arrives at the search area location, he performs a reconnaissance of the area.

4. Reconnaissance of the Search Area

After the 1SG arrives in the search area, he begins a reconnaissance. He is looking for areas that provide sufficient space and ideally, both cover and concealment. For the purposes of this mission, we narrowed the search criteria to include only tree canopies large enough to support a company assembly area. The result of this reconnaissance is either the identification of a tree canopy large enough to support a company sized assembly area within the confines of the search area box, or the center of mass location provided by the user. The simplified search mechanism for finding a suitable canopy location is discussed in Chapter VI.

5. Designating Unit Assembly Area Locations

In a company assembly area, each platoon is given a "piece of ground" to occupy within the boundaries of the assembly area. The platoon positions within the assembly area must provide 360 degree security for the company. The amount of space allocated to each platoon depends on the overall size of the assembly area. We can vary the size of our assembly area based on terrain constraints. The minimum radius for our assembly area was chosen to be thirty meters. The maximum, and default assembly area radius, is 250 meters.

By distributing the three platoons (twelve company vehicles) in thirty degree increments around a 360 degree assembly area, the platoon locations provide all around security for the company. The platoon occupation positions are sized in accordance with the minimum radius of the selected assembly area. An example diagram of an assembly area is shown below in Figure 18. The headquarters vehicles -- 7, the First Sergeant, 66, the Company Commander and 65, the Executive Officer -- orient their positions facing the enemy direction. The platoon positions on the perimeter of the assembly area are not changed with respect to the enemy's direction.

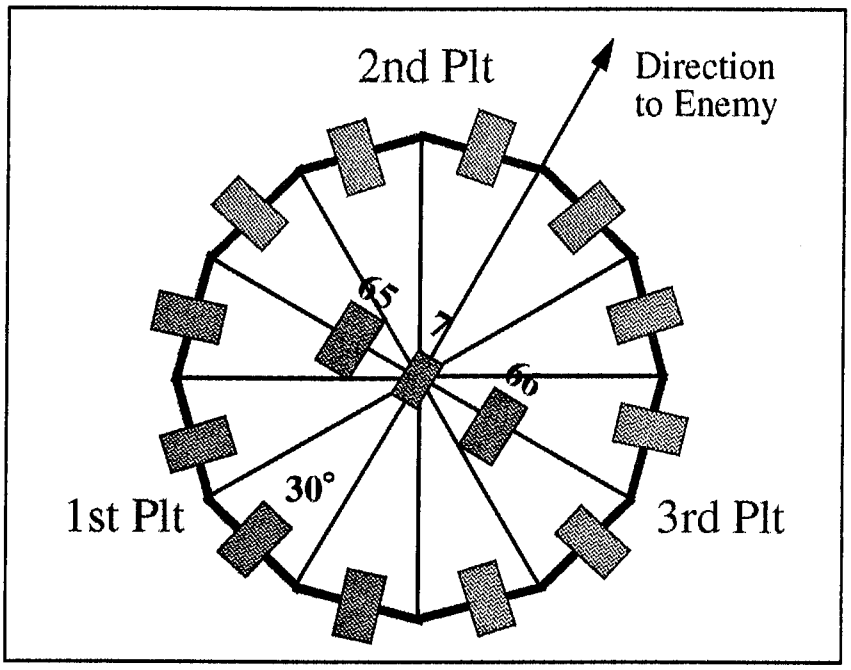


Figure 18: Occupation Positions for Assembly Area

Determining the individual points for a platoon's occupation is based on the radius of the assembly area. Given the center of mass of the assembly area, **X** and **Y**, and the minimum radius of the assembly area, ***radius,*** we can determine the coordinates of each of the points. Figure 19 identifies the five distinct points used to establish the occupy position line used by ModSAF's Prepare for Occupy Position Task. The equations are provided for

one platoon only; the others are derived in a similar fashion. The equations used to determine these points are included in Table 2 . The coordinate data shown does not consider any rotation to orient the assembly area towards the enemy.

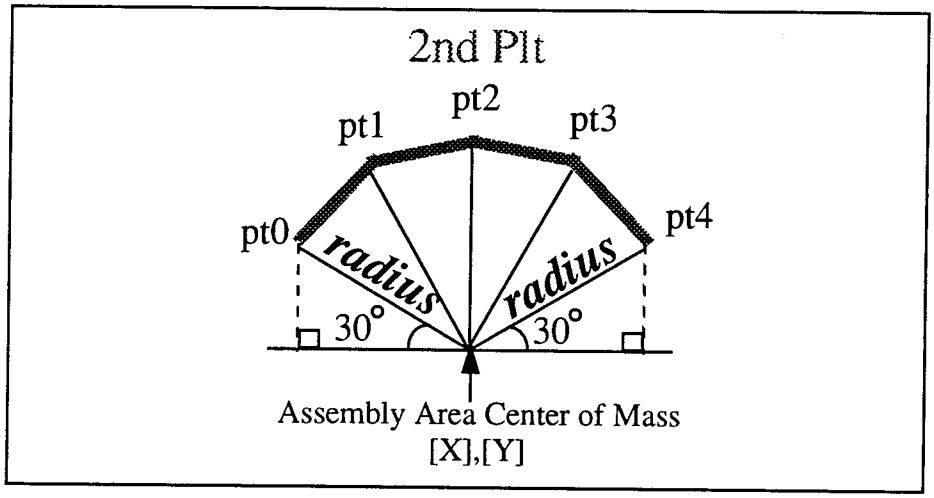


Figure 19: 2nd Platoon's Assembly Area

Point	X	Y
pt0	**X** - cos(30) * ***radius***	**Y** + sin(30) * ***radius***
pt1	**X** - sin(30) * ***radius***	**Y** + cos(30) * ***radius***
pt2	**X**	**Y** + ***radius***
pt3	**X** + sin(30) * ***radius***	**Y** + cos(30) * ***radius***
pt4	**X** + cos(30) * ***radius***	**Y** + sin(30) * ***radius***

Table 2: Coordinates for 2nd Platoon's Occupation Position

6. Route Planning for the Company

After the 1SG completes his reconnaissance, he has either found a suitable tree canopy location for the company, or establishes the assembly area at the center of mass location provided by the operator. He then identifies and selects a road route for the company. The

route will include a start point (SP), a release point (RP), and the route itself, the three basic control measures utilized for a military move operation.

When the operator desires to designate a route for a unit, he uses ModSAF's line editor. A line editor option is to generate a road route from the operator's input. The operator selects a start point and end point for the route. ModSAF utilizes an A-star search to identify a road route that connects the start point and end point, if one exists. The result of this function is a road route that gets stored as an object in the PO Database, or a system error message stating a road route could not be found. It was proposed that instead of having the operator use the line editor to specify a unit's route, that the unit have access to the road route building functions and build their own road routes. How this was accomplished is covered in more detail in Chapter VI, Design Implementation. Figure 20 shows a unit road route with the control measures (SP and RP) discussed above.

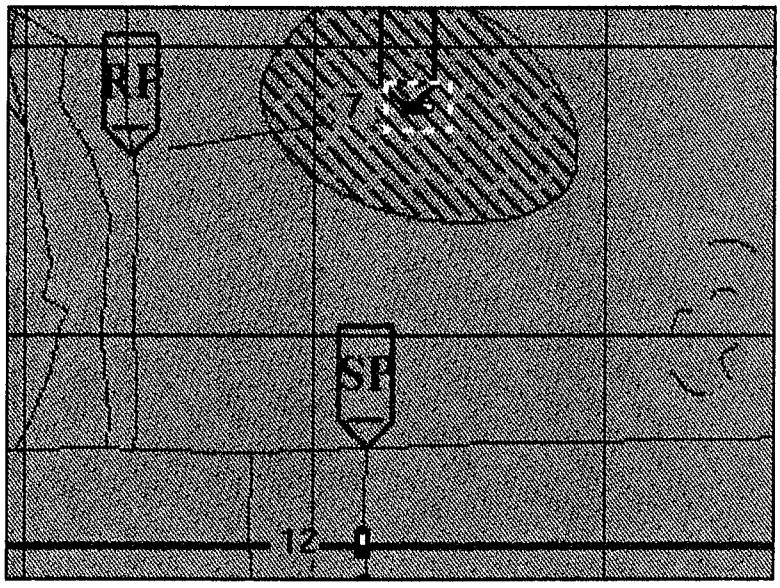


Figure 20: Road Route for Assembly Area

7. Company Movement to the Assembly Area

The company road march is conducted as individual platoon road marches. The platoon closest to the assembly area moves first, and reports when it has arrived at the start point (SP). The commander, the second to move, then begins his movement. The

commander in turn reports the SP and the next closest platoon begins its move. The company road march continues as determined by the company order of march. By breaking the company movement into platoon and vehicle moves, the assembly area mission can utilize the existing ModSAF Unit Travel Task -- a platoon level task.

8. Occupation of the Assembly Area

The assembly area mission uses the ModSAF "Occupy Position" task. Each platoon and headquarters tank is given a position to occupy within the assembly area. The assembly area support functions pass the required parameters to the occupy position task, which includes a line object representing the position to occupy, and three target reference points (TRPs), a left TRP, a right TRP, and an Engagement Area TRP. These TRPs are used to designate sectors of responsibility for each platoon. Shown below in Figure 21 is the ModSAF Occupy Position Task Entry Frame showing the occupation line, and the left, right, and engagement TRPs.

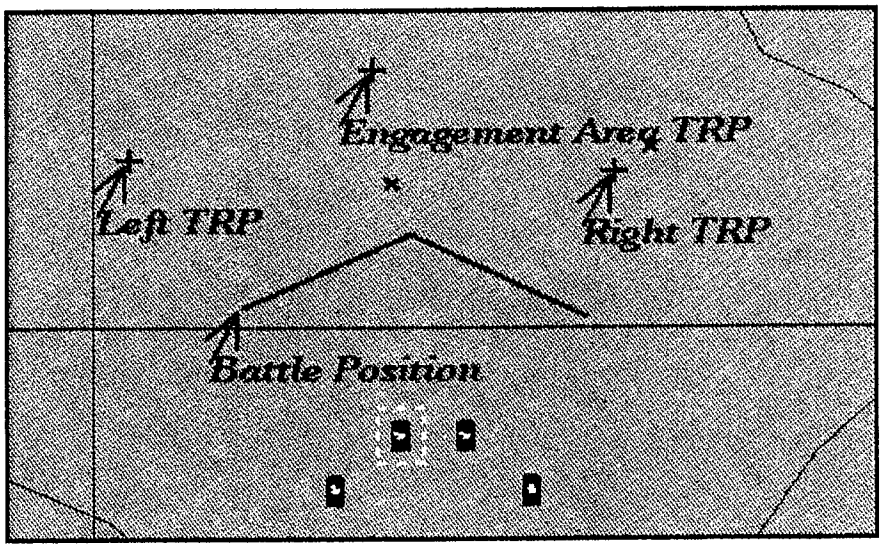


Figure 21: ModSAF's Occupy Position Task

D. ModSAF VEHICLE, UNIT, AND REACTIONARY TASKS

In addition to the ModSAF Vehicle Move Task used by the First Sergeant, and the Unit Travel and Occupy Position Tasks used by the platoons, several vehicle level tasks

including collision avoidance, path planning, sensor, turret, gun control, and the unit reactionary task, "Actions on Contact", are used by all of the vehicles when performing the assembly area mission.

E. FINITE STATE MACHINE ARCHITECTURE

The development of a finite state machine that represents the company mission assembly area was derived from Figure 15, Assembly Area Stages. The resulting finite state machine used for this mission is shown in Figure 22. The reverse path from "Moving_To

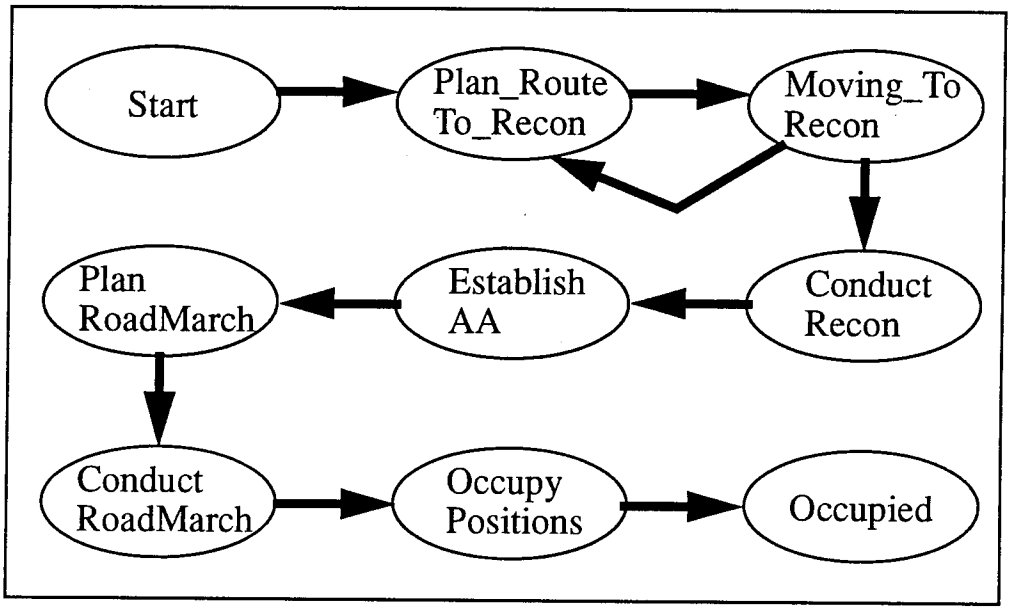


Figure 22: Assembly Area Finite State Machine

Recon" back to "Plan_Route_To_Recon" was a design strategy that allows the First Sergeant to plan multiple routes during his reconnaissance mission. He first plans a route to get to the assembly search area. He then plans a route from that location to the closest tree canopy that fulfills the requirements for the assembly area location.

F. ASSEMBLY AREA LIBRARY MODULE

The result of the design for the company assembly area mission will be an independent ModSAF library module, "Uassembly". The design plan was to currently limit its execution to the specific task organization of fourteen M1 tanks and a First Sergeant's

vehicle. The assembly area library module uses the same architecture and design as the other unit level tasks currently implemented in ModSAF.

G. COMMUNICATION BETWEEN AUTONOMOUS AGENTS

The assembly area mission attempts to capture the battlefield communication between the platoons, the Company Commander and the First Sergeant. The ability to display the intercommunication between units exists in ModSAF and was used within the assembly area mission.

H. SUMMARY

The design strategies adopted for research include:

- Utilize the Augmented Asynchronous Finite State Machine (AAFSM) architecture.
- Add a company level command abstraction to the AAFSM architecture.
- Provide company level mission planning.
- Reduce the number of operator inputs -- Generalize the mission tasking.
- Provide limited company level terrain reasoning in support of the assembly area mission.
- Add the capability to ModSAF for units to select their own routes.
- Provide a proof-of-concept prototype company mission that demonstrates company level missions can be realistically constructed using ModSAF's current command and control architecture.

VI. DESIGN IMPLEMENTATION

This chapter describes how the stages of the company assembly area mission described in Chapter V were incorporated into ModSAF. It provides a detailed analysis of what existing ModSAF functionality was used by the assembly area mission, and the changes and additions made to the ModSAF code in support of the company prototype mission "Assembly Area". It also introduces the command abstracted finite state machine used to control the platoon and vehicle level tasks.

A. TASK ORGANIZATION

ModSAF offers an M1 tank company as one of its unit types. The assembly area mission requires adding a new unit type, "M1 Company w/1SG", which includes fourteen M1 tanks and a First Sergeant's truck. Adding this new unit organization requires modifying "echelon.rdr" in the ModSAF Library Module "Libechelondb". The "echelon.rdr" file is a ModSAF "reader file". A reader file is a text based file that allows easy modification by the developer when changing the parameters for ModSAF. The Libechelondb library provides standard military echelon unit organizations, from section to battalion and higher. [ModSAF Libech, 93] A unit organization is developed by adding vehicle leaf nodes or other unit tree nodes to the organization. A tree node is another unit organization previously defined in "echelon.rdr" and is recursively expanded when the unit is created. The echelondb format for an M1 Armor Platoon is:

```
(unit_US_M1_Platoon ((leaf vehicle_US_M1 "??1")
                     (leaf vehicle_US_M1 "??2")
                     (leaf vehicle_US_M1 "??3")
                     (leaf vehicle_US_M1 "??4"))).
```

The question marks at the end of the leaf designation represent an inheritance for the vehicle's numbering system. This platoon will inherit the company's letter designation and the platoon's number. So if this is the 1st Platoon of C Company, the numbering for the vehicles listed above would be C11, C12, C13, and C14.

The ModSAF M1 Company in echelondb format is:

```
(unit_US_M1_Company ((leaf vehicle_US_M1 "?66")
                     (leaf vehicle_US_M1 "?65")
                     (tree unit_US_M1_Platoon "?1 ")
                     (tree unit_US_M1_Platoon "?2 ")
                     (tree unit_US_M1_Platoon "?3 ")))
```

Here, the company letter is determined by the operator when the unit is created. If the operator specifies this unit as "D" Company, the first two leaf vehicles will be D66 (the Commander's tank) and D65 (the Executive Officer's tank). The following three tree entries will be expanded according to the previous definition of "unit_US_M1_Platoon" in "echelondb.rdr". The numbering for the first tree unit inherits the company designation "D" and a "1". This platoon "D1" is expanded using the definition for "unit_US_M1_Platoon". As previously shown, this platoon will be expanded to four leaf vehicles, D11, D12, D13, and D14. The process continues until all of the entries for the unit are leaf nodes.

Using the echelondb format, creating a new unit entry for "M1 Company w/1SG" is:

```
(unit_US_M1_Company ((leaf vehicle_US_HUMMV "?7")
                     (leaf vehicle_US_M1 "?66")
                     (leaf vehicle_US_M1 "?65")
                     (tree unit_US_M1_Platoon "?1 ")
                     (tree unit_US_M1_Platoon "?2 ")
                     (tree unit_US_M1_Platoon "?3 "))).
```

The characteristics for a specific vehicle type are contained in parametric reader files. These files are located in the "entities" subdirectory. An example model parameter file for the US M1 Abrams Tank is contained in Appendix A. Since these files are reader files they can be tailored by the operator for a specific application. The model parameter file for the US HUMMWV, the First Sergeant's vehicle, was not included in version 1.0 of ModSAF, and was therefore added to the existing set of vehicle model parameter files.

B. CREATING A NEW TASK LIBRARY

As described in Chapter III, behaviors for vehicles and units in ModSAF are controlled by tasks. The steps to create a new task, and its associated behaviors, is described in the ModSAF 1.0, Developer's Class Work Book [Robasky, 94-2], and is shown in Figure 23.

- Creating the SAF Object Class,
- Creating a Makefile for the Task Library,
- Creating the New Task Finite State Machine File,
- Modifying the Task Source and Header Files,
- Creating the Reader File for the Task,
- Modifying ModSAF's "taskframes.rdr" file,
- Modifying ModSAF's "standard_params.rdr" file,
- Building the Task,
- Documenting the Task.

Figure 23: Steps to Create a Task

For additional information about the specific steps for creating a new task library, consult the ModSAF 1.0, Developer's Class Work Book. [Robasky, 94-2]

C. DEVELOPMENT OF THE FINITE STATE MACHINE CODE

The first step in creating the Unit Assembly Area Mission was to create the library module directory for the task. This library was named "libuassembly" and was included in "/modsaf/common/libsrc". The finite state machine file for the unit assembly area mission was named "uassembly_task.fsm" and the task named "uassembly". After designing the finite state diagram shown in Chapter V (Figure 22), we considered which existing ModSAF tasks the assembly area finite state machine would utilize to conduct its mission. These ModSAF tasks become subtasks for the uassembly task.

1. Vehicle Tasks

The First Sergeant moves on his own to the assembly area search site. This being a single vehicle move, the existing ModSAF task Vehicle Move was selected for inclusion as a subtask. Additionally, the reactive tasks for "Actions on Contact" were included as a subtask in the "taskframes.rdr" file for the uassembly task. By including the reactive task in the "taskframes.rdr" file, all entities within the uassembly task have these reactive tasks running concurrently with the assembly area tasks.

2. Platoon Level Tasks

Two of the existing ModSAF unit tasks were selected as subtasks for the assembly area mission. The first was the Unit Travel task. Each platoon, and both of the headquarters tanks, use a Unit Travel task to perform the company roadmarch phase of the mission. When the units arrive at the assembly area location, they occupy their respective positions within the assembly area. The ModSAF task "Occupy Position" performs this task sequence. Chapter V describes the occupation task for the assembly area mission.

3. ModSAF Finite State Machine Protocol Language

The Assembly Area finite state machine is written in accordance with the specifications described in the LibTask Programmer's Guide, 1993 [Smith, 93]. The Programmer's Guide sets forth a finite state machine protocol language which allows the use of the ModSAF Asynchronous Augmented Finite State Machine (AAFSM) Code Generator. The code generator converts a finite state machine source file into C code. Thus, a simple finite state machine protocol language is utilized to describe the structure of the task. The supporting routines, written in C, are added by the developer to support the behaviors of his task. The assembly area finite state machine (fsm) source code "uassembly_task.fsm" written in the fsm protocol language, without the added support routines, is included as Appendix B.

D. FINITE STATE MACHINE SUPPORT ROUTINES

The finite state machine support routines for a ModSAF task are created by the developer to perform the desired actions while in a particular state. In the following paragraphs we discuss the support routines needed to perform the actions for each state of the finite state machine. The support routines developed for each of the states of the assembly area finite state machine are shown in Figure 24.

Support routines for the FSM

Start
static void initialize_statics();

Plan_Route_To_Recon
static int32 compute_recon_route();
static void create_new_route_to_objective_com();
static void create_aa_bound_box();

Moving_To_Recon
static void send_arrival_report();

Conduct_Recon
static int32 search_for_tree_canopies();

Establish_AA
static void build_assembly_area();

Plan_Roadmarch
static void build_roadmarch();

Conduct_Roadmarch
static void send_sp_report();
static void send_rp_report();
static int32 check_for_SP();

Occupying_Positions
static void clean_march_pts();

End
static void clean_occupy_pts();
static int32 delete_graphic_com();

Figure 24: Assembly Area Support Routines

1. Plan Route to Recon

The assembly area mission begins in the state ***"Plan_Route_To_Recon"***. The purpose of this state is to:

- Establish the search space for the assembly area,
- Create a graphic entry in the unit's overlay,
- Establish a route for the First Sergeant to get to this location,
- Create the route graphic for the First Sergeant's overlay, and
- Spawn a ModSAF Vehicle Move Task for the First Sergeant.

The support functions for this state include:

- "compute_recon_route",
- "create_new_ route_to_objective_com", and
- "create_aa_bound_box".

We transition to the next state, ***"Moving_To_Recon"*** after spawning the First Sergeant's Vehicle Move (VMOVE) task. The selection of a unit's route will be discussed in more detail in Paragraph 5, Planning the Roadmarch to the Assembly Area. The implementation of the assembly area bounding box graphic entry is discussed below.

One of the fundamentals of developing a ModSAF task is the ability to relate graphic control measures depicted on the terrain map display with a particular unit or entity. ModSAF maintains a unit overlay for each entity it simulates. When the operator selects a unit, the overlay for that unit is displayed on the terrain map. We wish to add a bounding box graphic in the company overlay for the assembly area mission. When the operator selects the assembly area mission, he is asked to designate a center of mass location for the search area. This location is stored in a private variable "private->objective_com" associated with the unit. The center of mass (COM) for the search space is used by the function "create_aa_bound_box" to create a graphical box entry in the company overlay. The COM location determines the four end points of the line object which are shown in Table 3. The coordinate units are expressed in meters.

Point \ Location	X	Y
Point 0	COM - 1500.0	COM + 1500.0
Point 1	COM + 1500.0	COM + 1500.0
Point 2	COM + 1500.0	COM - 1500.0
Point 3	COM - 1500.0	COM - 1500.0

Table 3: Bounding Box Coordinates

Once the data for the line object has been entered, we must save the line object into the PO Database -- the database ModSAF utilizes to store the necessary information about the entities it is simulating. We want to check to find if their already exists an assembly area bounding box. If there exists a box, we update its position. If it does not exist, we create a new object and store it in the PO Database. We save this line object as a state variable for the company as "state->assy_area". That way if we change the location for the assembly area, we have a state variable we can lookup and modify. In Chapter III we discussed the use of the state variables (available to the network) and the private variables (available only to the SAFSim's) which "augment" the finite state machine architecture. The resulting graphic that is entered into the company overlay is shown in Figure 25. The support function "create_aa_bound_box" is included as Appendix C.

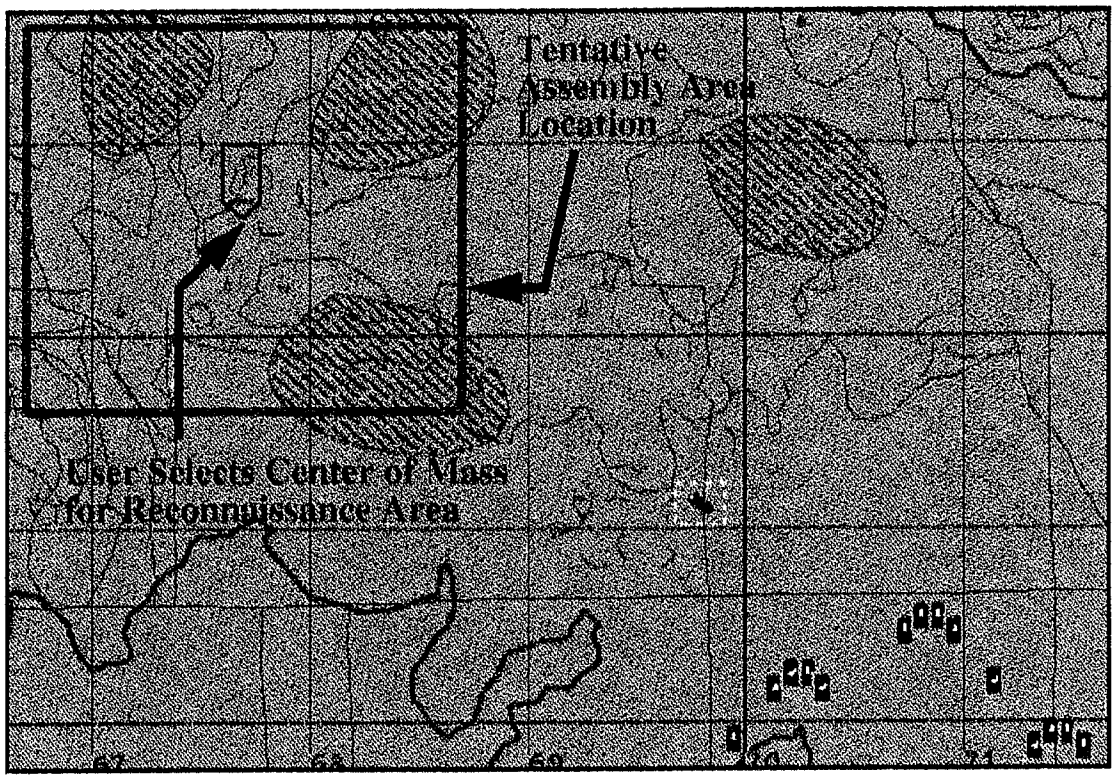


Figure 25: Assembly Area Bounding Box

2. Moving to the Reconnaissance Site

The purpose of the state ***"Moving_to_Recon"*** is to monitor the progress of the Vehicle Move task spawned by ***"Plan_Route_To_Recon"***. The support function for this state includes "send_arrival_report". We transition to the next state, ***"Conduct_Recon"*** after the First Sergeant's Vehicle Move (VMOVE) task transitions to an "arrived" state.

The Vehicle Move task is an existing ModSAF unit task and includes several states of its own as shown in Figure 26. The previous state, ***"Plan_Route_To_Recon"*** spawns the Vehicle Move task and then switches the state of the assembly area finite state machine to ***"Moving_To_Recon"***. When the First Sergeant arrives at the search area, the Vehicle Move task transitions to the "arrived" state. ***"Moving_To_Recon"*** sends an arrival report to the commander, and transitions to the next state, ***"Conduct_Recon"***. Figure 27 shows the company assembly mission's ***"Moving_To_Recon"*** state

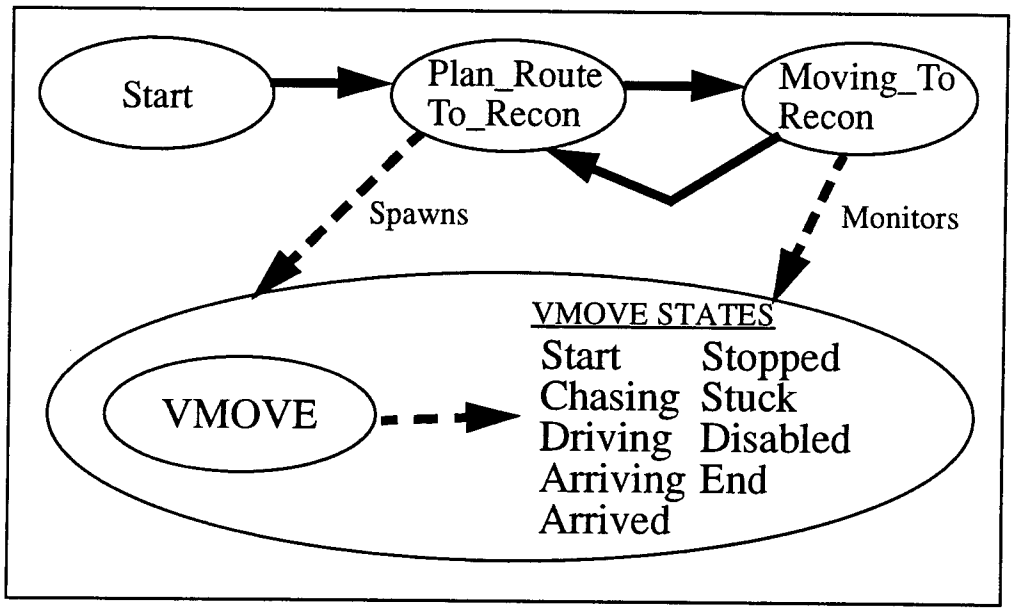


Figure 26: State Plan Route to Recon

3. Conducting the Reconnaissance

The purpose of the state ***"Conduct_Recon"*** is to identify the best location to establish an assembly area within the confines of the assembly area bounding box. The support

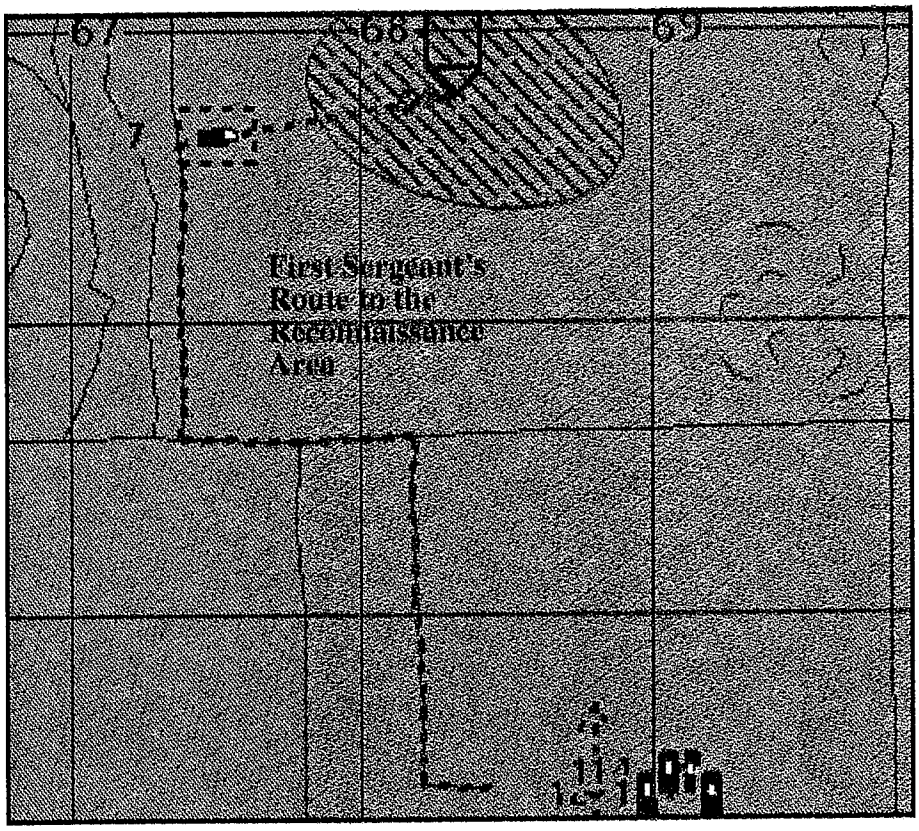


Figure 27: Moving to the Reconnaissance Area

function for this state is "search_for_tree_canopies". If a tree canopy is found within the bounding box that is large enough to support the assembly area, we transition back to ***"Moving_To_Recon"*** and move to the tree canopy. Otherwise, we did not find a tree canopy and we establish the assembly area location at the center of mass of the search area, and transition to the next state ***"Establish_AA"***.

One of the design goals of the assembly area mission was to add a degree of company level mission planning. There are currently only a few terrain analysis functions available in ModSAF. We incorporated a search of the QUAD Tree database for tree canopy locations and a very simplistic algorithm that estimates the center of mass location for the tree canopy. The goal here was not to perfect the terrain analysis algorithm, but to use it as the means to demonstrate the ability of a unit to conduct some level of autonomous mission planning.

Tree canopies are features contained in the Quad Tree Database of the ModSAF "terrain" library. A tree canopy is composed of a set of points that establishes the perimeter of the canopy. Very simply, we average these point locations to estimate a center of mass location for the canopy. This does not work for concave canopies. When a canopy extends beyond the boundaries of our assembly area bounding box, we do not consider those points in the center of mass calculation. The result is a very unsophisticated algorithm that estimates the center of mass of tree canopies, providing a limited terrain analysis ability for tree canopies that currently does not exist in ModSAF. The results of a tree canopy selection is shown in Figure 28. The "search_for_tree_canopies" function is included as Appendix E..

4. Establishing the Assembly Area

The purpose of the state ***"Establish_AA"*** is to create the graphic entries for all of the unit's "Occupy Position" tasks. The support function for this state is "build_assembly_area". Once the positions are created, we transition to the next state, ***"Plan_Roadmarch"***.

The function "build_assembly_area" establishes the occupation positions and target reference points for each of the platoons and headquarters tanks, and stores these graphic entries in the respective unit overlay. These graphics are required for the ModSAF Occupy Position tasks as previously outlined in Chapter V (Figure 21). The assembly area positions were implemented as outlined in Chapter V, Designating Unit Assembly Area Locations and as shown in Figure 18. The function "build_assembly_area" is included as Appendix D. Figure 29 shows the occupation position and target reference points for the third platoon for a company assembly area.

5. Planning the Roadmarch to the Assembly Area

The purpose of the state ***"Plan_Roadmarch"*** is to establish a route for the company to get to the AA, and create the route graphic for the company overlay. The support function

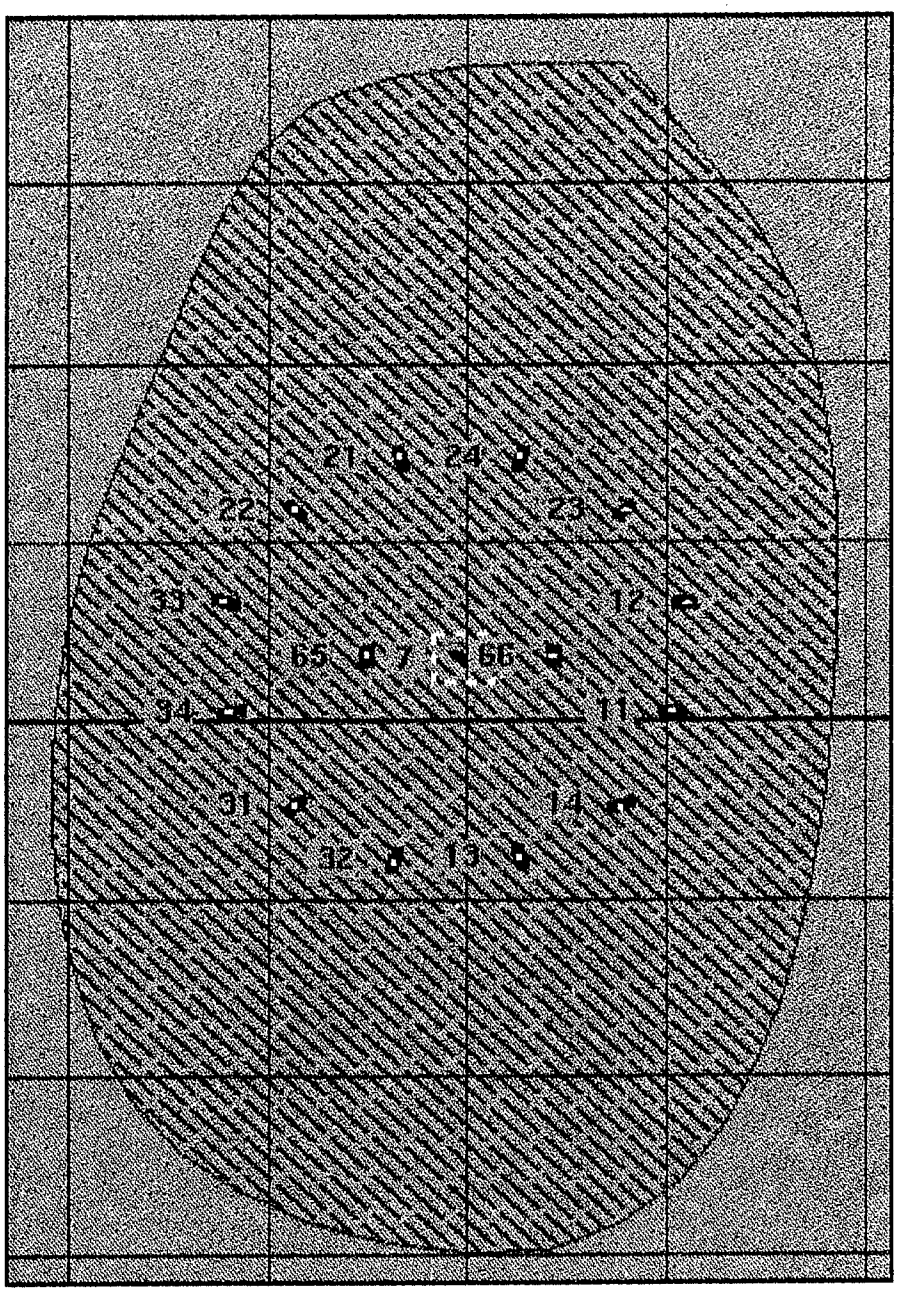


Figure 28: Canopy Center of Mass Selection

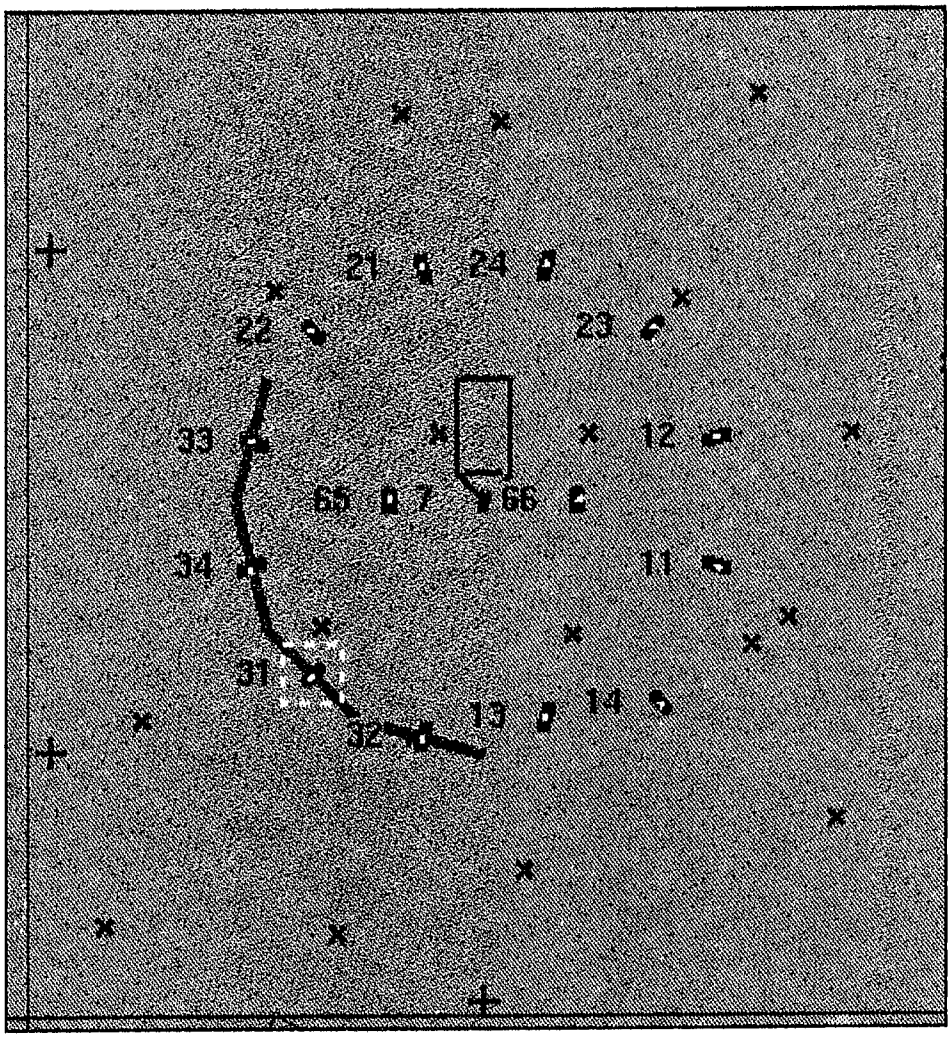


Figure 28: Graphic Entry for Occupy Position

for this state is "build_roadmarch". After building the route for the company, we transition to the next state, ***"Conduct_Roadmarch"***.

First we discuss adding the functionality to ModSAF that will allow a unit to develop its own road route. Then we discuss how we utilize this new functionality to establish the road routes for the First Sergeant and for the company.

First, we needed to determine what ModSAF used to allow the operator to implement a road route. The ability for the operator to provide a start point and end point and have ModSAF determine a connecting road route was briefly discussed in Chapter V. See "Route Planning for the Company" on page 39. We analyzed the C code of "edt_line.c" in

the "Libeditor" directory of the ModSAF source library. Building a road route consists of reading in the operator's mouse location as he designates the start point and updating the road route as the mouse is moved towards the end point. The parametric inputs are the X and Y locations of the mouse as the operator is making the route, and are passed to a function "rt_allocate_road_route_from_networks" contained in "rt_roads.c" of the "Libroute" directory. This function searches the Quad Tree Database and selects the closest road segment for the starting road point, and using an A-star search, attempts to make a road route that connects the start point to the current position of the mouse cursor. The "rt_allocate_road_route_from_networks" function takes as input arguments the Quad Tree Database being used, the segment number of the closest road segment, the start location, the segment number of the ending road segment, and the ending location. It then attempts to create a road route that starts at the start point, gets on the road at the near segment, travels on the road to the far segment, and moves to the end point. The function returns a "ROUTE_LIST" which describes the road route in terms of the road segments, the individual points of the road segments used for the route, and an ordering of these points. An example diagram that depicts the input parameters for this function including: the start point (X0, Y0), the near road segment (Seg0), the end point (X1, Y1), and the far road segment (Seg1) is shown in Figure 29.

The parameters that we did not currently have to make a direct call to "rt_allocate_road_route_from_networks" were the segment numbers of the near and far road segments. We found a function "find_nearest_segment" in "road_routes.c" of the "Libquad" directory. This function, however, was an internal private function. We needed the ability to determine the segment numbers of the nearest road to a given location which this function provides. We therefore added "find_nearest_segment" to the header file for "libquad.h" making it a publicly accessible function. With these modifications, we could utilize "rt_allocate_road_route_from_networks" directly. By providing the unit's current location as the start point, the desired ending location for the end point, and the return road

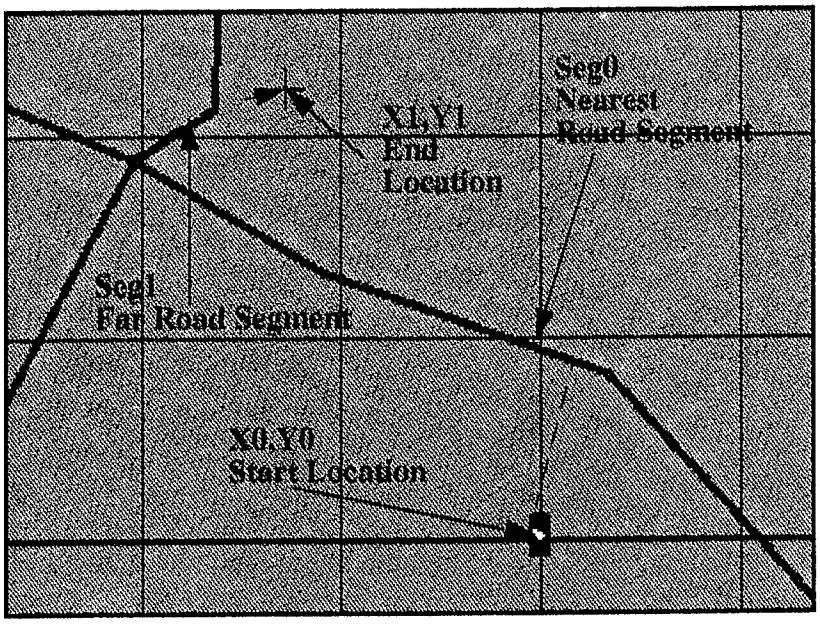


Figure 29: Allocating a Route From Roads

segment numbers from calls to "find_nearest_segment" for both of these points, ModSAF will build a road route for our unit.

In the design strategy, we decided that the road route should include a start point (SP), a release point (RP), and the road route itself. The RP was selected to be the last road point of the road route. The route object was the "ROUTE LIST" returned by the call to "rt_allocate_road_route_from_networks". The start point had to be "massaged" to prevent a problem encountered when using ModSAF's Unit Travel task. When a unit is assigned a route and performs the Unit Travel task, it first determines the "optimal" starting point to get on to the route. This "optimization" could lead a unit to skip the start point, which is not allowed in a standard military road march. Travelling to the start point when conducting a military road march is not an option. The optimization behavior exhibited in the Unit Travel task is shown in Figure 30 and Figure 31. Figure 30 shows a road route and the start point for the road route. The vehicle is assigned a Unit Travel task that uses this route.

Figure 31 shows the vehicle as it just gets to its optimized starting point for the selected route. Here, clearly, the vehicle does not travel to the start point, and it was this optimization behavior of the Unit Travel task that had to be circumvented to perform a

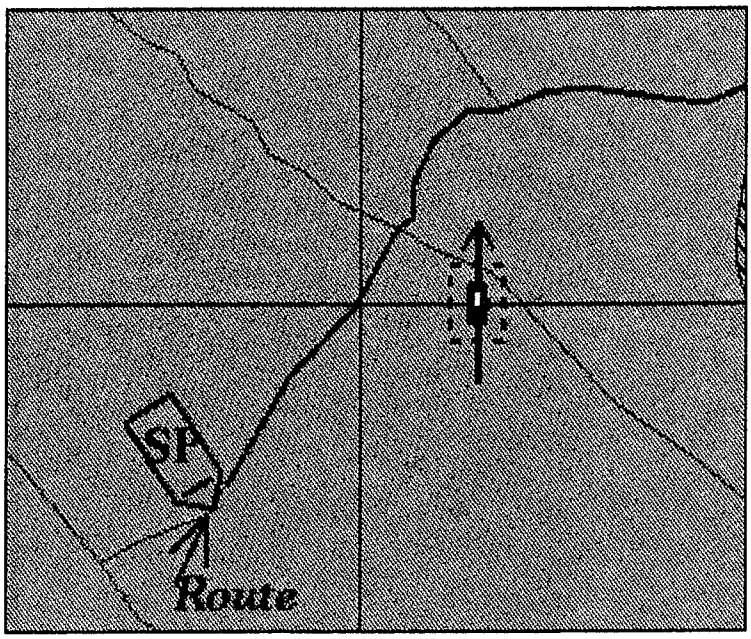


Figure 30: Unit Travel Road Route Frame 1

more realistic military road march. Therefore, we move the SP point out successively along the road route until the SP is closer to the assembly area than any of our units.

6. Moving to Assembly Area

The purpose of the state ***"Conduct_Roadmarch"*** is to conduct the company's roadmarch to the assembly area and act as an abstracted command finite state machine. The supporting functions for this state include: "check_for_SP", "send_sp_report", and "send_rp_report". We transition to the next state, ***"Occupying_Positions"*** when all of the units have completed their Unit Travel tasks.

The assembly area state ***"Conduct_Roadmarch"*** not only controls the movement of the company to the assembly area, but also monitors and assigns the appropriate task to each platoon depending on their situations. It is this part of the assembly area finite state machine that makes it unique from most ModSAF unit tasks. In a typical ModSAF unit task, we only change states when the unit performing the task meets the transition requirements of the task. For instance, if a unit is given a mission to move to a point and then occupy a position, that unit must complete its movement before it can transition into

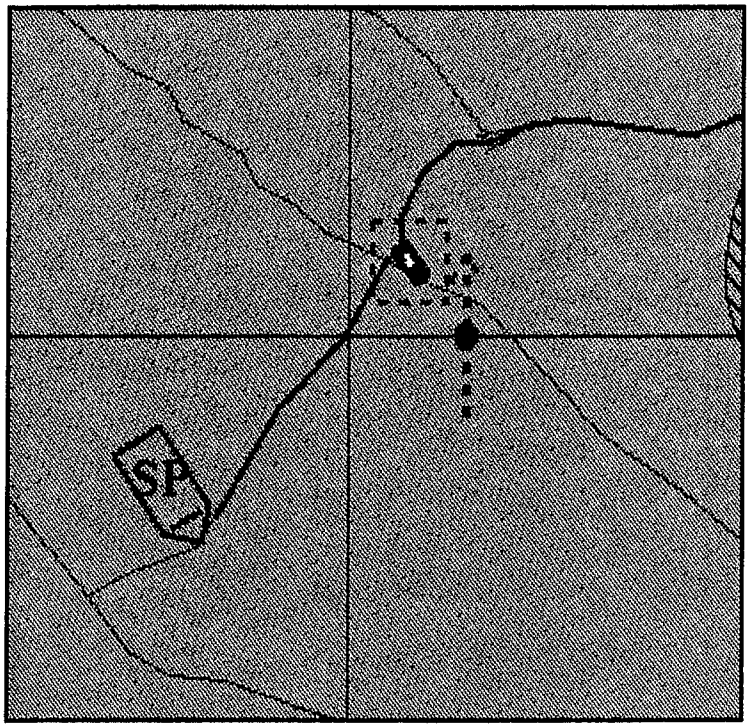


Figure 31: Unit Travel Road Route Frame 2

occupying the position. If we applied this rigidity to the assembly area mission, the company road march must be completed by the entire company before the first platoon begins its occupation of the assembly area. We could end up with a company of tanks in a column formation sitting on the road waiting for the last platoon to finish its roadmarch. In an actual roadmarch, the platoons do not stop at the release point, but begin occupation of the assembly area as soon as they arrive at the release point. This is in compliance with [USA FM 71-1, 88] which states: "Move vehicles from Release Point (RP) into assembly area without stopping." Therefore the ***"Conduct_Roadmarch"*** state of the assembly area finite state machine acts as a sequencer for the subordinate unit's tasks, and abstracts the command and control of the company -- like the control provided by the Company Commander. It is this command and control element that we wished to capture in the assembly area mission.

The roadmarch is actually five separate Unit Travel tasks -- one for each platoon, and one for each headquarters tank. The order of march for the company is determined according to the proximity of the platoons to the assembly area location. The closest platoon becomes the lead platoon. The Commander (66 tank) follows the lead platoon. The next closest platoon is next in the order of march and is followed by the Executive Officer (65 tank). The platoon farthest from the assembly area is last in the order of march.

The movement of the individual units is monitored and controlled by the ***"Conduct_Roadmarch"*** state of the assembly area finite state machine. When the lead platoon arrives at the SP location, it sends a report to the commander ("send_sp_report") and then the Commander begins his movement. Likewise, when the Commander reaches the SP, he communicates this to the company and the next closest platoon begins its movement. In this way, we maintain adequate spacing between the units and control their movement along the company route.

The command level abstraction of the assembly area mission is most prominently displayed in the transition between the company beginning its roadmarch and then occupying the assembly area. The purpose is to allow the platoons to independently change states while the company commander monitors their states. This was a design implementation that we felt provided an abstracted command and control finite state machine. In fact, there typically is a time during the assembly area mission when the lead platoon is occupying the assembly area task, the next platoon is conducting its roadmarch, and the last platoon is waiting to begin its roadmarch. Figure 32 shows the Company Assembly Area task in its ***"Conduct_Roadmarch"*** state with several platoons in different states.

The significance of the differences between a typical ModSAF unit level task finite state machine and this command and control finite state machine is discussed in further detail. See "TASKS AND TASK FRAME MANAGEMENT" on page 65.

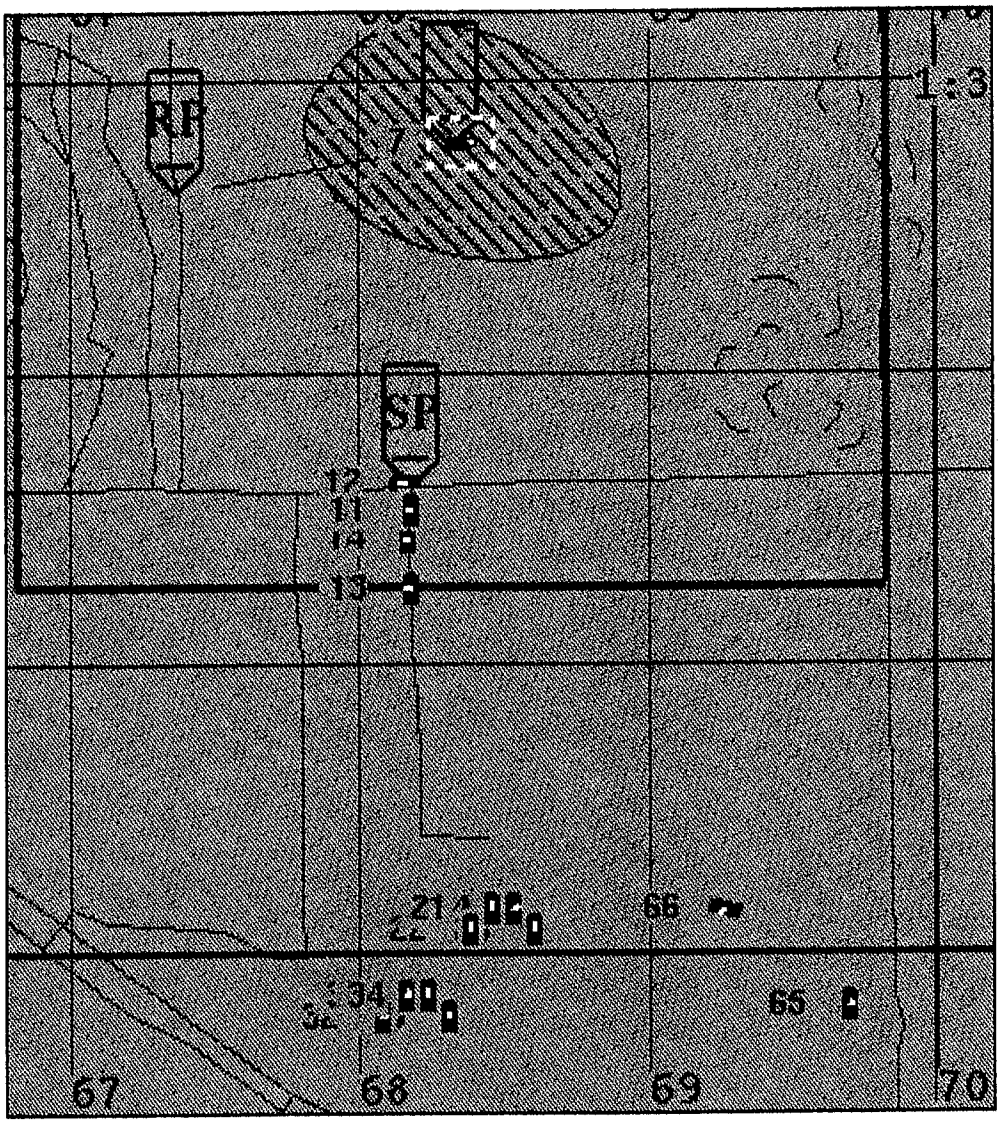


Figure 32: Conduct Roadmarch in Multiple States

7. Occupying the Assembly Area

The purpose of the state ***"Occupying_Positions"*** is to monitor the status of the unit Occupy Position tasks spawned in the previous state, ***"Conduct_Roadmarch"***. The supporting function for this state is "clean_march_points". We transition to the next state, ***"Occupied"*** when all of the units have finished their Occupy Position tasks. The "clean_march_points" function is a utility that removes the graphic entries used to conduct the company roadmarch to the assembly area.

One of the advantages of having the assembly area finite state machine utilize the existing ModSAF Occupy Position task is to capture the individual vehicle behaviors assigned in the ModSAF task. If the assembly area reconnaissance does not find a canopy, the Occupy Position task still requires each vehicle to find cover when occupying its position in relation the engagement area target reference point. Shown in Figure 34 is a vehicle finding cover during the Occupy Position task. The yellow tick marks represent the boundary of his search space. The black line segment is the position he is to occupy. The red tick marks are point positions that provide cover with respect to the engagement area target reference point. The blue target reference point is the chosen occupy point location. For a more detailed description of the algorithm used to identify a covered position, consult the ModSAF Libupoccpos Programmer's Guide. [ModSAF Libpocc, 93]

Figure 34: Occupy Position Finding Cover

8. Unit Occupied Assembly Area

The purpose of the state ***"Occupied"*** is to maintain the assembly area location until the operator terminates the mission. This state required no supporting functions and transitions to the ***"End State"*** when the operator terminates the mission or selects a different mission for the unit.

9. Ending the Assembly Area Mission

The purpose of the state ***"End"*** is to remove from the Persistent Object Database the graphic entries introduced for the occupy position tasks. The support functions for this state are "clean_occupy_pts" and "delete_graphic_com". The assembly area finite state machine exits after completing this state.

E. ASSEMBLY AREA LIBRARY MODULE

The end result of the design implementation is a self-contained library module "Libuassembly" incorporated into the ModSAF Library Source files. The library module includes the following files:

- "Makefile" (Compilation instructions),
- "libuassembly.h" (the header file),
- "libuassembly.info" (text information file),
- "libuassembly.texinfo" (tex format based information file),
- "uassembly.rdr" (reader file for the mission),
- "uassembly.xrdb" (X-window based commands for editor),
- "uassembly_access.c" (Allows access to mission information),
- "uassembly_class.c" (The class hierarch for the mission),
- "uassembly_init.c" (Initiation procedures for the mission),
- "uassembly_params.c" (Parametric data handling),
- "uassembly_task.fsm" (Mission in fsm protocol language),
- "uassembly_task.c" (Resulting C code after AAFSM conversion),
- "uassembly_task.h" (Resulting header file after AAFSM conversion),
- "uassembly_util.c" (Utility functions).

F. TASKS AND TASK FRAME MANAGEMENT

A company level task cannot be generalized to being simply a collection of three identical platoon level tasks running concurrently. As discussed earlier, in the assembly area mission, the company first conducts a road march to the assembly area location and then occupies the location. We do not, however, want the entire company to complete the roadmarch before beginning the occupation of the assembly area. The platoons are conducting different individual missions that encompass the overall company level mission. We need a mechanism to decide the particular unit task an individual platoon should execute. The problem with attempting to generalize a company level mission from a like platoon level mission can best be explained using a company attack. A platoon on its own conducting an attack may maneuver and fire towards the enemy. A company attack, however, is not simply three platoons conducting concurrent attacks on the same enemy location. The company commander must analyze the enemy situation as soon as the lead platoon makes contact. He may then decide to establish an attack by fire position for the lead platoon and maneuver the remaining two platoons to capitalize on the enemy's weak side. It is proposed that the command finite state machine as used by the assembly area mission could interpret the changing environment (identify an enemy weakness) and initiate the necessary platoon level actions to capitalize on this weakness.

In general, our argument is that the company level missions cannot be developed as a set list of company level tasks, executed in sequence, and managed by the ModSAF task manager. The role of the task manager, as outlined in Chapter III, is to ensure the prerequisite tasks and follow-on tasks are performed for a particular unit level task. Instead, we supply a company level task manager, operating as its own finite state machine, that can decide which platoon level tasks to execute. It is this ability to trigger platoon level tasks based on the environment that is a unique approach to the command and control architecture previously used in ModSAF.

G. SUMMARY

The implementation of the assembly area mission included making modification to the existing ModSAF code. These changes included:

- Providing the ability for a unit to determine its own route.
- Adding a limited ability to conduct tree canopy terrain analysis.
- Adding an abstracted command finite state machine to sequence the unit tasks for the platoons and vehicles of the company.

The end result is a stand alone library module "Libuassembly" that performs the Company Assembly Area mission, and provides the new capabilities listed above.

VII. RESULTS AND CONCLUSIONS

A. SUMMARY

This chapter evaluates the Company Level Assembly Area mission described in this thesis and proposes areas for future work. With only minor modification to the existing code and the addition of some new source code, ModSAF 1.0 was used to develop and implement a prototype company-level mission. While this research only attempts to scratch the surface of a much larger research problem, it establishes a foothold to continuing research in the development of higher level mission planning, both within the architecture of ModSAF, and in other related architectures.

B. ASSESSMENT OF ASSEMBLY AREA MISSION

1. Route Planning

The ability for a unit to determine its own road route given a starting location and and ending location was added to ModSAF. We attempted to portray a standard military road march which includes some basic control measures. This included (as a minimum) a start point, a route, and a release point. We discussed having to work around some of the "features" that ModSAF provides in its Unit Travel task, like the optimization of the entrance point to a route that conflicts with having to cross the start point before using the route. Future developments in ModSAF may allow the designation of an absolute starting point for the route, which would more closely replicate a standard military road movement.

2. Terrain Analysis

Much additional work is needed in this area. We provide only a limited algorithm that returns a gross approximation of a tree canopy center of mass; this is used only to portray the ability of the units to analyze the terrain and perform mission planning.

3. Abstracted Command Finite State Machine Architecture

We used a finite state machine architecture that abstracted the command and control of a company level mission by controlling the independent platoon tasks which make up this mission. Our implementation of the finite state machine architecture is different from most existing ModSAF tasks. Adding real-time mission planning to a company level mission more closely resembles the Company Commander's real-time decision making process occuring on the battlefield. With more research, the use of a command-level finite state machine may be shown to be instrumental in controlling company level tasks as a collection of independent platoon level tasks. The trade off of real-time mission planning is the potential time delays introduced into a real time simulation system. However, without a greater degree of autonomy, the goal of alleviating the operator from providing the necessary realism between platoons and companies will prevent him from replicating a force much larger than a battalion.

4. Realism and Testing of Assembly Area Mission

Adequate testing for realistic behavior is an entire research area by itself. Testing the assembly area mission for realism included a visual inspection of the behaviors of the vehicles and units using NPSNET. Using NPSNET, local subject matter experts among the students and faculty of the Naval Postgraduate School, provided feedback as to the realism of the assembly area mission. Whereas the company road march received high reviews for its realistic execution and more natural (and human-like) command and control, the selection of routes needs additional work. Algorithms to assist in choosing a best route, i.e. when a direct line is more appropriate than a road route, is a needed design improvement.

5. The Library Module Assembly Area

Similar to all other ModSAF unit tasks, the assembly area task is encapsulated in a stand-alone library module and has been incorporated into the ModSAF Library Source files.

C. CONCLUSIONS

The primary purpose of this research was to establish a proof-of-concept that higher level tasks, specifically company level tasks and missions, could be developed and incorporated into ModSAF. The result is a prototype company level mission -- Occupy an Assembly Area -- using the finite state machine architecture of ModSAF 1.0. This mission provides realistic timing constraints, communication amongst the autonomous agents, and an abstracted commanding finite state machine that provides the building blocks for the more complex company-level missions Attack and Defend. The limited complexity of the Assembly Area mission permitted rapid development and testing while utilizing the finite state machine architecture. The individual behaviors of the autonomous agents could be individually analyzed and selectively modified.

In our developed Company Assembly Area mission, the First Sergeant, operating independently of the company, conducted a reconnaissance type mission with specific parameters -- identifying a suitable assembly area location. The Company Commander, communicating with the First Sergeant, developed a company road march plan, integrating the individual platoons' road marches into a company level road march. The ability to control multiple platoons performing different unit level tasks is demonstrated in the Assembly Area state ***"Conduct_Roadmarch"***. The ability to control platoon level tasks at an abstracted company commander level utilizing ModSAF 1.0's current finite state machine architecture is both possible and promising.

One of the underlying themes of autonomous agents is to free the operator from replicating low level behaviors. These behaviors should be automated but still portray realism. For an operator to effectively control higher level units, at the company, battalion, or higher levels, he must have the ability to implement company missions, which exceeded the current abilities of ModSAF 1.0.

The analysis of the existing ModSAF architecture, the development of a new company level mission, and the testing and evaluation of this mission leads us to draw the following conclusions:

- ModSAF entities require additional terrain reasoning algorithms.
- ModSAF entities should perform some degree of mission planning.
- Company level missions should be designed as a collection of independent platoon level tasks.
- The current AAFSM architecture of ModSAF can be utilized to develop realistic company-level missions.
- An abstracted command-level finite state machine controlling the platoon level finite state machines is one approach to higher-level command and control.

D. RECOMMENDATIONS FOR FUTURE WORK

Continued development utilizing a company-level finite state machine that orchestrates individual platoon tasks to accomplish a company-level mission should be applied to the more complex, and needed, company missions of Attack and Defend. These two company missions are essential for the growth and development of ModSAF. Having gained some insight into the development of higher level mission planning, follow-on researchers and developers in this critical area at the Naval Postgraduate School may spend more time on the development of the behaviors and less time on understanding the ModSAF architecture and system. The cooperative use of information amongst the platoon entities, and the command and control decisions made by the company commander with respect to this information is a promising area for research.

Another area of potential research deals with the integration of external mission planners to coordinate company level missions. ModSAF 1.0 has the capability to interact with SOAR; SOAR is being utilized to enhance the realistic behaviors of complex flight systems and aircraft. The use of SOAR or other external mission planners interacting with ModSAF to plan and control company and higher level missions is another promising are of research in the future development of ModSAF. [Rosenbloom et al, 94]

APPENDIX A

This is the finite state machine for the company level mission "Assembly Area" written in the finite state machine protocol language as set forth in the LibTask Programmer's Guide [Smith, 93].

```
uassembly UASSEMBLY_PARAMETERS UASSEMBLY_STATE UASSEMBLY_VARS

SUBTASK: Vehicle_Move vmove_task SM_VMove VMOVE_PARAMETERS 1
         vmove_init_task_state BCKGRND
{
subtask->route          = state->recon_route[which];
subtask->speed          = private->params->speed;
subtask->speed_limit    = private->params->speed;
subtask->move_flags = VMOVE_DRIVE_AT_SPEED | VMOVE_STOP_WITH_DIRECTION;
subtask->termination    = VMOVE_TERM_NEVER;
subtask->update_distance = 200.0;
subtask->driving_style  = VMOVE_NORMAL;
}
END_SUBTASK

SUBTASK: Unit_Travel utraveling_task SM_UTravel UTRAVELING_PARAMETERS 1
         utrav_init_task_state
{
 subtask->route          = state->recon_route[which];
 subtask->speed          = private->params->speed;
 subtask->speed_limit    = private->params->speed;
 subtask->form_type      = 0; /* OPEN FORMATION */
 subtask->roadmarch      = 1; /* ROADMARCH STATION KEEPING */
 subtask->conform        = 0;
 bzero(&subtask->leader, sizeof(VehicleID));
 subtask->follow_dist    = 0.0;
 subtask->follow_angle   = 0.0;
 strcpy(subtask->formation, "column");
}
END_SUBTASK

SUBTASK: Unit_Prep_Occupy_Position upoccupypos_task SM_UPrepOcpyPos
         UPOCCPOS_PARAMETERS 4 upoccpos_init_task_state
{
 subtask->battle_position      = state->plt_position[which];
 subtask->trp_left             = state->left_trp[which];
 subtask->trp_right            = state->right_trp[which];
 subtask->engagement_area      = state->ea_trp[which];
}
END_SUBTASK
```

```
`
 tick()
`
 params()
`
START
`
initialize_statics(private, state, unit_entry);

switch (state->state)
{
 case PLAN_ROUTE_TO_RECON:
 case MOVING_TO_RECON:
 case CONDUCT_RECON:
 case ESTABLISH_AA:
 case PLAN_ROADMARCH:
 case CONDUCT_ROADMARCH:
 case OCCUPYING_POSITIONS:
 case OCCUPIED:
  break;
 case ended:
 default:
  ^PLAN_ROUTE_TO_RECON;
}
`
PLAN_ROUTE_TO_RECON
`     tick
{
 int32                i, vmove_substate, found_canopy = FALSE;
 PO_DB_ENTRY      *subord_entry[UNITORG_MAX_BREADTH];

 unitorg_get_context(unit_entry, (int32)TRUE, (PO_DB_ENTRY **)NULL,
                     (PO_DB_ENTRY **)NULL, (int32 *)NULL, (int32)0,
                     subord_entry, &private->n_subordinates, 32);

/* This function initializes the objective_com and recon_route
* for the 1SG to the AA.
*/

 if (compute_recon_route(vehicle_id, unit_entry, parameters,
                         private, state))
                         SPAWN &PO_OBJECT_ID(subord_entry[0]) Vehicle_Move 0
 else
     UPDATE Vehicle_Move 0
^MOVING_TO_RECON;
}
`          params
initialize_statics(private, state, unit_entry);
```

```
`
MOVING_TO_RECON
`     tick
{
 int32 vmove_substate;
 if(!(private->print_MOV_TO_RECON))
  {
   printf("\nSTATE = MOVING_TO_RECON \n");
   private->print_MOV_TO_RECON = TRUE;
   printf("Direction to enemy = %f\n", parameters->orientation);
  }

 vmove_substate = vmove_state(private->po_db, &state->vmove_task[0]);
 if (vmove_substate == VMOVE_ARRIVED)
  {
   send_arrival_report(vehicle_id, private);
   ^CONDUCT_RECON;
  }
}

`          params
 if(compute_recon_route(vehicle_id, unit_entry, parameters,
                        private, state))
      UPDATE Vehicle_Move 0
 ^MOVING_TO_RECON;
`
CONDUCT_RECON
`     tick
{
 static int32          found_canopy = FALSE;
 int32                 num_canopies = 0, i = 0, vmove_substate;
 float64               my_pos[3];
 PO_DB_ENTRY           *subord_entry[UNITORG_MAX_BREADTH];
 QUAD_LIST             *feature_index_list = NULL;

/* ********************** *
* CONDUCT RECON *
* ********************************************************
* In this state we search the assembly area bounding box for
* for a tree canopy that has a minimum radius for the company.
* If a canopy is not found, establish the assembly area at the
* center of mass of the search area. The boundary of the
* AA box is considered an absolute edge boundary and our
* position cannot be outside it.
*
* Future Work: Schedule a set of tree canopies to conduct
* our reconnaissance.  This should include the canopies
* of interest, and if no canopies areas are found find the
* best hide positions (to be developed).  We must take
* ********************************************************
*/
```

```
for(i=0;i<3;i++) my_pos[i] = 0.0;

 if(!(private->print_CONDUCT_RECON))
  {
   printf("\nSTATE = CONDUCT_RECON \n");
   private->print_CONDUCT_RECON = TRUE;
  }

 unitorg_get_context(unit_entry, (int32)TRUE, (PO_DB_ENTRY **)NULL,
                          (PO_DB_ENTRY **)NULL, (int32 *)NULL, (int32)0,
                          subord_entry, &private->n_subordinates, 32);

 ent_get_position(vehicle_id, my_pos);

 feature_index_list = quad_get_quad_nodes(private->quad_data, 0,
                                                  my_pos[X],
                                                  my_pos[Y],
                                                  my_pos[X],
                                                  my_pos[Y]);

 num_canopies = private->quad_data->canopies.num_models;

 if(!found_canopy)
  {
   for (i=0; i<= num_canopies; i++)
    {
     if(search_for_tree_canopies(private->quad_data, i, my_pos[X], my_pos[Y], private))
      found_canopy = TRUE;
    }
  }

/* The above function returns the largest canopy within the AA box.
* It does not schedule visiting the other points of
* interest.  Future Work. Make it select all canopies within
* the aa box and mark them graphically. (working).
* Tailor the bounding box algorithm to fit within
* the AA box and find com of a partial canopy within the AA box.
*/

 if(found_canopy)
  {
   printf("The COM of the Selected Canopy is: %f, %f\n",
        private->canopy_com[X], private->canopy_com[Y]);
   printf("With a Minimum Radius = %f\n", private->radius);

   if((int32)private->objective_com[X] == (int32)private->canopy_com[X] &&
     (int32)private->objective_com[Y] == (int32)private->canopy_com[Y])
    {
     printf("WERE HERE\n");
     ^ESTABLISH_AA;
    }
```

```
      else
       {
        printf("Changing the objective com location\n");
        private->objective_com[X] = private->canopy_com[X];
        private->objective_com[Y] = private->canopy_com[Y];

        create_new_route_to_objective_com(vehicle_id, unit_entry,
                                  private, private->objective_com,
                                  state, parameters);

        UPDATE Vehicle_Move 0
         initialize_statics(private, state, unit_entry);
        ^MOVING_TO_RECON;
       }

    }
   else
    {
     printf("No canopies found ... SO WERE HERE \n");
     private->radius = 250.0;
     ^ESTABLISH_AA;
    }
}
`           params
`
ESTABLISH_AA
`      tick
{
 if(!(private->print_EST_AA))
   {
    printf("\nSTATE = ESTABLISH_AA \n");
    private->print_EST_AA = TRUE;
   }

 /* build_assembly_area adds to the each individual unit overlay
  * the graphics needed to occupy an assembly area.
  */

 build_assembly_area(vehicle_id, unit_entry, private,
              state, private->objective_com, private->radius, parameters);

 ^PLAN_ROADMARCH;
}
`           params
```

```
`
PLAN_ROADMARCH
`       tick
{

 /* Here we look to find the unit closest to the AA and then find
  * a road_route (if one exists) to get us there.
  * Here is where I want to call a routine that establishes the overlay
  * objects for the roadmarch, including the Start Point (SP), Control Points
  * (optional) and a Release Point (RP).
  */

 if(!(private->print_PLAN_RD))
  {
   printf("\nSTATE = PLAN_ROADMARCH \n");
   private->print_PLAN_RD = TRUE;
  }

 build_roadmarch(vehicle_id, unit_entry, private, state);

 ^CONDUCT_ROADMARCH;
}
`           params
`
CONDUCT_ROADMARCH
`       tick
{
 int32        i=0, order = 0, prev_order = 0, all_occupying = FALSE;
 PO_DB_ENTRY   *subord_entry[UNITORG_MAX_BREADTH];
 PO_DB_ENTRY   *subsubunit_entry[UNITORG_MAX_BREADTH];
 UnitClass     *unit;

 if(!(private->print_CONDUCT_RD))
  {
   printf("\nSTATE = CONDUCT_ROADMARCH \n");
   private->print_CONDUCT_RD = TRUE;
  }

unitorg_get_context(unit_entry, (int32)TRUE, (PO_DB_ENTRY **)NULL,
              (PO_DB_ENTRY **)NULL, (int32 *)NULL, (int32)0,
              subord_entry, &private->n_subordinates, 32);

for (i=1; i<6; i++)
 {
  order = private->march_order[i];
  if(private->moving[order]) /* Vehicle is moving. Hit SP? */
   {
    if(!(private->sp_hit[order]))
     {
      if(check_for_SP(order, unit_entry, private, state))
       {
```

```
            private->sp_hit[order] = TRUE;
            printf("Unit %d just hit SP\n", order);
           }
         }
        else /* have already hit SP, are we there yet */
         {
          if(!(private->occupying[order]))
           if(
            FINISHED Unit_Travel order
            )
            {
             DELETE Unit_Travel order
             send_rp_report(order, unit_entry, private, state);
           SPAWN &PO_OBJECT_ID(subord_entry[order]) Unit_Prep_Occupy_Position order
             private->occupying[order] = TRUE;
             printf("Unit %d is now occupying position\n", order);
            }
         }
       }
     else /* this unit is not moving */
      {
       if(i == 1) /* if this is the closest platoon get it going */
        {
         printf("Unit %d is first to move\n", order);
         SPAWN &PO_OBJECT_ID(subord_entry[order]) Unit_Travel order
         private->moving[order] = TRUE;
        }
       else
        {
         prev_order = private->march_order[i-1];
         if(private->sp_hit[prev_order]) /* Unit before me sp'd? */
          {
           printf("Prev unit %d sp'd. Unit %d is now moving\n",
                    prev_order, order);
           SPAWN &PO_OBJECT_ID(subord_entry[order]) Unit_Travel order
           private->moving[order] = TRUE;
          }
        }
      }
   }

all_occupying = TRUE;

for (i=1; i<6; i++)
 if(!(
    FINISHED Unit_Travel i
   ))
  all_occupying = FALSE;

if(all_occupying)
 ^OCCUPYING_POSITIONS;
```

```
}
`
        params
`
OCCUPYING_POSITIONS
`    tick
{
 int32 i = 0, occupied_yet = FALSE;

 if(!(private->print_OCC_AA))
     {
      printf("\nSTATE = OCCUPYING_POSITIONS \n");
      private->print_OCC_AA = TRUE;
     }

 clean_march_pts(private, state); /* Delete pts/lines from march */

 occupied_yet = TRUE;

 for (i = 1; i < 6; i++)
  {
   if (!(
       FINISHED Unit_Prep_Occupy_Position i
       ))
    occupied_yet = FALSE;
  }


 /* If all the vmove(s) and upoccp(s) are completed goto next state */
 if (occupied_yet)
  ^OCCUPIED;  All done assembly, go to non-end state.
}
`
        params
`
OCCUPIED
`    tick
`
        params
`
END
`
 printf("ENDED STATE \n");
 clean_occupy_pts(private, state);
 if(delete_graphic_com(private,
                private->po_db,
                parameters->objective))
  printf("We deleted the objective \n");
`
SUSPEND
`
 /* NOP */;
`
```

APPENDIX B

```
/* ***********************************
* CREATE_AA_BOUND_BOX *
* ***********************************************************************
* This creates a bounding box around the tentative assembly area
* search space.  It remains until a new mission is selected. We
* selected a bounding box that is 3 kilometers by 3 kilometers.
* The right edge of the box is 1500 meters from the center of mass.
* ***********************************************************************/
static void create_aa_bound_box (unit_entry, private, state)
PO_DB_ENTRY             *unit_entry;
UASSEMBLY_VARS          *private;
UASSEMBLY_STATE         *state;
{
  char                  buf[2048];
  UnitClass             *unit;
  LineClass             *line = (LineClass *)buf;
  PO_DB_ENTRY           *line_entry;

bzero(line, sizeof(buf));
unit = &PO_UNIT_DATA(unit_entry);
line->overlayID = unit->overlayID;
line->style = LSplain;
line->color = OCOverlayDefault;
line->pointCount = 4;
line->thickness = 3;
line->width = 0;
line->beginArrowHead = noArrowHead;
line->endArrowHead = noArrowHead;
line->closed = TRUE;
line->dashed = FALSE;
line->splined = FALSE;
line->route = FALSE;
line->munition = 0;
line->density = 0.0;
line->points[0].pointNumber = 0;
line->points[0].pointType = PTLocation;
line->points[0].variant.location.x =  private->objective_com[X] - 1500;
line->points[0].variant.location.y =  private->objective_com[Y] + 1500;
line->points[1].pointNumber = 1;
line->points[1].pointType = PTLocation;
line->points[1].variant.location.x = private->objective_com[X] + 1500;
line->points[1].variant.location.y = private->objective_com[Y] + 1500;
line->points[2].pointNumber = 2;
line->points[2].pointType = PTLocation;
line->points[2].variant.location.x =  private->objective_com[X] + 1500;
line->points[2].variant.location.y =  private->objective_com[Y] - 1500;
```

```
line->points[3].pointNumber = 3;
line->points[3].pointType = PTLocation;
line->points[3].variant.location.x = private->objective_com[X] - 1500;
line->points[3].variant.location.y = private->objective_com[Y] - 1500;

/* If an AA box already exists, update it. Otherwise create a new one */

if ((line_entry = po_get_object(private->po_db, &state->assy_area)) &&
    (PO_OBJECT_CLASS(line_entry) == objectClassLine))
      line_entry = po_change_object(private->po_db, line_entry, line,
                                    PRO_PO_LINE_CLASS_SIZE(line->pointCount));
else
   line_entry = po_create_object(private->po_db, NULL, objectClassLine,
                                 FALSE, line,
                                 PRO_PO_LINE_CLASS_SIZE(line->pointCount),
                                 NULL);
state->assy_area = PO_OBJECT_ID(line_entry);
}
```

APPENDIX C

```
/* ********************************
 * BUILD_ASSEMBLY_AREA
 * ********************************************************************
 * This function builds the graphic entries for the occupy position
 * task. It establishes the assembly area for the company.
 * We build the assembly area with relation to the current position of
 * the 1SG's (7's) vehicle.
 * ********************************************************************
 */
static void build_assembly_area(vehicle_id, unit_entry, private,
                                 state, objective_com, radius, parameters)

    int32                       vehicle_id;
    PO_DB_ENTRY                 *unit_entry;
    UASSEMBLY_VARS              *private;
    UASSEMBLY_STATE             *state;
    int32                       *objective_com;
    float64                     radius;
    UASSEMBLY_PARAMETERS *parameters;
{
  char              buf[2048];
  int32             i;
  float64           my_pos[3];
  UnitClass         *unit;
  LineClass         *line = (LineClass *)buf;
  PO_DB_ENTRY       *line_entry[6],*point_entry[6];
  PO_DB_ENTRY       *subord_entry[UNITORG_MAX_BREADTH];
  PointClass        point;

  for(i=0;i<3;i++) my_pos[i] = 0.0;

/* ****************************************************************
* The 1SG is already in the center of the assembly area.  We need
* to construct the occupy positions for the 2 HQ Tanks, and 3 platoon
* positions. For each position we need 3 TRP's -- a Left TRP, Right TRP,
* and an Engagement Area TRP.
* For our assembly area location we simplify the TRP's by having
* the platoons share TRP's (see diagram below).
* ****************************************************************
*/
```

```
/* ************************************************************
*              Diagram of Assembly Area
* ************************************************************
*
*
*
*
*
*
*
*
*
*
*
*
* ************************************************************/

unitorg_get_context(unit_entry, (int32)TRUE, (PO_DB_ENTRY **)NULL,
                    (PO_DB_ENTRY **)NULL, (int32 *)NULL, (int32)0,
                    subord_entry, &private->n_subordinates, 32);

/* Get 7's position and direction */
ent_get_position(vehicle_id, my_pos);
ent_get_direction(vehicle_id, private->orig_direction);

bzero(line, sizeof(buf));
bzero(&point, sizeof(point));

/* ************************************************************ *
* Attributes common to all platoon positions -- line structure *
* ************************************************************ */
line->style = LSplain;
line->color = OCOverlayDefault;
line->thickness = 4; /* lets make it easier to see */
line->width = 0;
line->beginArrowHead = noArrowHead;
line->endArrowHead = noArrowHead;
line->closed = FALSE;
line->dashed = FALSE;
line->splined = FALSE;
line->route = FALSE;
line->munition = 0;
line->density = 0.0;
line->pointCount = 5;
/ * ************************** *
* Attributes common to TRP's *
* ************************** */
point.style = PSTRP;
point.color = OCOverlayDefault;
point.dashed = 0;
```

```
/* ******************************** *
* Occupy Position Line for 2nd Plt *
* ******************************** */
unit = &PO_UNIT_DATA(subord_entry[3]); /* 2nd Plt's Overlay */
line->overlayID = unit->overlayID;
point.overlayID = unit->overlayID;

line->points[0].pointNumber = 0;
line->points[0].pointType = PTLocation;
line->points[0].variant.location.x =  objective_com[X] - .8660254 * radius;
line->points[0].variant.location.y =  objective_com[Y] + .5 * radius;
line->points[1].pointNumber = 1;
line->points[1].pointType = PTLocation;
line->points[1].variant.location.x = objective_com[X] - .5 * radius;
line->points[1].variant.location.y = objective_com[Y] + .8660254 * radius;
line->points[2].pointNumber = 2;
line->points[2].pointType = PTLocation;
line->points[2].variant.location.x =  objective_com[X];
line->points[2].variant.location.y =  objective_com[Y] + radius;
line->points[3].pointNumber = 3;
line->points[3].pointType = PTLocation;
line->points[3].variant.location.x = objective_com[X] + .5 * radius;
line->points[3].variant.location.y = objective_com[Y] + .8660254 * radius;
line->points[4].pointNumber = 4;
line->points[4].pointType = PTLocation;
line->points[4].variant.location.x =  objective_com[X] + .8660254 * radius;
line->points[4].variant.location.y =  objective_com[Y] + .5 * radius;
line_entry[0] = po_create_object(private->po_db, NULL, objectClassLine,
                                 FALSE, line,
                                 PRO_PO_LINE_CLASS_SIZE(line->pointCount),
                                 NULL);
state->plt_position[3] = PO_OBJECT_ID(line_entry[0]);

/* ******************** */
/* Left TRP for 2nd plt */
/* ******************** */
point.location.x = (int32) objective_com[X] - 2.0 * 0.8660254 * radius;
point.location.y = (int32) objective_com[Y] + 2.0 * 0.5 * radius;
point_entry[0] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->left_trp[3]  = PO_OBJECT_ID(point_entry[0]);
/* ********************** */
/* Right TRP for 2nd Plt */
/* ********************** */
point.location.x = (int32) objective_com[X] + 2.0 * 0.8660254 * radius;
point.location.y = (int32) objective_com[Y] + 2.0 * 0.5 * radius;
point_entry[1] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->right_trp[3] = PO_OBJECT_ID(point_entry[1]);
```

```
/* ************************************** */
/* Engagement Area TRP for 2nd Plt */
/* ************************************** */
point.location.x = (int32) objective_com[X];
point.location.y = (int32) objective_com[Y] + 2.0 * radius;
point_entry[2] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->ea_trp[3] = PO_OBJECT_ID(point_entry[2]);

/* ******************************** */
/* Occupy Position Line for 66 */
/* ******************************** */
unit = &PO_UNIT_DATA(subord_entry[2]); /* 66's Overlay */
line->overlayID = unit->overlayID;
point.overlayID = unit->overlayID;

line->pointCount = 2;
line->points[0].variant.location.x = (int32) objective_com[X] + (0.25 * radius) *
    cos(parameters->orientation);
line->points[0].variant.location.y = (int32) objective_com[Y] + (0.25 * radius) *
    sin(parameters->orientation);
line->points[1].variant.location.x = (int32) objective_com[X] + (0.50 * radius) *
    cos(parameters->orientation);
line->points[1].variant.location.y = (int32) objective_com[Y] + (0.50 * radius) *
    sin(parameters->orientation);

line_entry[1] = po_create_object(private->po_db, NULL, objectClassLine,
                                 FALSE, line,
                                 PRO_PO_LINE_CLASS_SIZE(line->pointCount),
                                 NULL);
state->plt_position[2] = PO_OBJECT_ID(line_entry[1]);

/* *************** */
/* Left TRP for 66 */
/* *************** */
point.location.x = (int32) objective_com[X] - 2.0 * 0.8660254 * radius;
point.location.y = (int32) objective_com[Y] + 2.0 * 0.5 * radius;
point_entry[0] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->left_trp[2]  = PO_OBJECT_ID(point_entry[0]);

/* ***************** */
/* Right TRP for 66 */
/* ***************** */
point.location.x = (int32) objective_com[X] + 2.0 * 0.8660254 * radius;
point.location.y = (int32) objective_com[Y] + 2.0 * 0.5 * radius;
point_entry[1] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->right_trp[2] = PO_OBJECT_ID(point_entry[1]);
```

```
/* ************************* */
/* Engagement Area TRP for 66 */
/* ************************* */
point.location.x = (int32) objective_com[X];
point.location.y = (int32) objective_com[Y] + 2.0 * radius;
point_entry[2] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->ea_trp[2] = PO_OBJECT_ID(point_entry[2]);

/* ******************************** */
/* Occupy Position Line for 3rd Plt */
/* ******************************** */
unit = &PO_UNIT_DATA(subord_entry[4]);
line->overlayID = unit->overlayID;
point.overlayID = unit->overlayID;

line->pointCount = 5;
line->points[0].pointNumber = 0;
line->points[0].pointType = PTLocation;
line->points[0].variant.location.x = objective_com[X];
line->points[0].variant.location.y = objective_com[Y] - radius;
line->points[1].pointNumber = 1;
line->points[1].pointType = PTLocation;
line->points[1].variant.location.x = objective_com[X] - 0.5 * radius;
/* sin 30 * radius */
line->points[1].variant.location.y = objective_com[Y] - 0.8660254 * radius;
/* cos 30 * radius */
line->points[2].pointNumber = 2;
line->points[2].pointType = PTLocation;
line->points[2].variant.location.x =  objective_com[X] - 0.8660254 * radius;
/* cos 30 * radius */
line->points[2].variant.location.y =  objective_com[Y] - 0.5 * radius;
/* sin 30 * radius */
line->points[3].pointNumber = 3;
line->points[3].pointType = PTLocation;
line->points[3].variant.location.x = objective_com[X] - radius;
line->points[3].variant.location.y = objective_com[Y];
line->points[4].pointNumber = 4;
line->points[4].pointType = PTLocation;
line->points[4].variant.location.x =  objective_com[X] - 0.8660254 * radius;
/* cos 30 * radius */
line->points[4].variant.location.y =  objective_com[Y] + 0.5 * radius;
/* sin 30 * radius */
line_entry[1] = po_create_object(private->po_db, NULL, objectClassLine,
                                 FALSE, line,
                                 PRO_PO_LINE_CLASS_SIZE(line->pointCount),
                                 NULL);
state->plt_position[4] = PO_OBJECT_ID(line_entry[1]);
```

```
/* ******************** */
/* Left TRP for 3rd Plt */
/* ******************** */
point.location.x = (int32) objective_com[X];
point.location.y = (int32) objective_com[Y] - 2.0 * radius;
point_entry[0] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->left_trp[4] = PO_OBJECT_ID(point_entry[0]);
/* ********************* */
/* Right TRP for 3rd Plt */
/* ********************* */
point.location.x = (int32) objective_com[X] - 2.0 * 0.8660254 * radius;
/* 2 * cos 30 * radius */
point.location.y = (int32) objective_com[Y] + 2.0 * 0.5 * radius;
 /* 2 * sin 30 * radius */
point_entry[1] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->right_trp[4]  = PO_OBJECT_ID(point_entry[1]);

/* ******************************** */
/* Engagement Area TRP for 3rd Plt */
/* ******************************** */
point.location.x = (int32) my_pos[X] - 2.0 * 0.8660254 * radius;
/* 2 * cos 30 * radius */
point.location.y = (int32) my_pos[Y] - 2.0 * 0.5 * radius;
/* 2 * sin 30 * radius */
point_entry[2] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->ea_trp[4] = PO_OBJECT_ID(point_entry[2]);

/* ********************************* */
/* Occupy Position Line for 1st Plt */
/* ********************************* */
unit = &PO_UNIT_DATA(subord_entry[1]);
line->overlayID = unit->overlayID;
point.overlayID = unit->overlayID;

line->points[0].pointNumber = 0;
line->points[0].pointType = PTLocation;
line->points[0].variant.location.x =  objective_com[X] + 0.8660254 * radius;
line->points[0].variant.location.y =  objective_com[Y] + 0.5 * radius;
line->points[1].pointNumber = 1;
line->points[1].pointType = PTLocation;
line->points[1].variant.location.x = objective_com[X] + radius;
line->points[1].variant.location.y = objective_com[Y];
line->points[2].pointNumber = 2;
line->points[2].pointType = PTLocation;
line->points[2].variant.location.x = objective_com[X] + 0.8660254 * radius;
line->points[2].variant.location.y = objective_com[Y] - 0.5 * radius;
line->points[3].pointNumber = 3;
line->points[3].pointType = PTLocation;
```

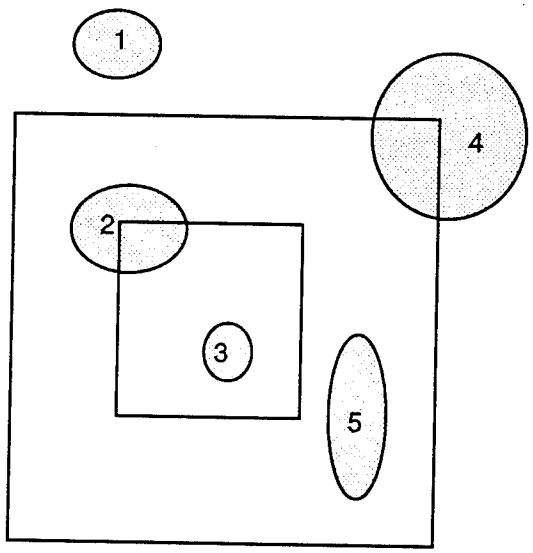
```
line->points[3].variant.location.x =  objective_com[X] + 0.5 * radius;
line->points[3].variant.location.y =  objective_com[Y] - 0.8660254 * radius;
line->points[4].pointNumber = 4;
line->points[4].pointType = PTLocation;
line->points[4].variant.location.x = objective_com[X];
line->points[4].variant.location.y = objective_com[Y] - radius;

line_entry[2] = po_create_object(private->po_db, NULL, objectClassLine,
                                 FALSE, line,
                                 PRO_PO_LINE_CLASS_SIZE(line->pointCount),
                                 NULL);
state->plt_position[1] = PO_OBJECT_ID(line_entry[2]);

/* ********************** */
/* Left TRP for 1st Plt */
/* ********************** */
point.location.x = (int32) objective_com[X] + 2.0 * 0.8660254 * radius;
point.location.y = (int32) objective_com[Y] + 2.0 * 0.5 * radius;
point_entry[0] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->left_trp[1] = PO_OBJECT_ID(point_entry[0]);

/* ************************ */
/* Right TRP for 1st Plt */
/* ************************ */
point.location.x = (int32) objective_com[X];
point.location.y = (int32) objective_com[Y] - 2.0 * radius;
point_entry[1] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->right_trp[1] = PO_OBJECT_ID(point_entry[1]);

/* ************************************ */
/* Engagement Area TRP for 1st Plt */
/* ************************************ */
point.location.x = (int32) my_pos[X] + 2.0 * 0.8660254 * radius;
point.location.y = (int32) my_pos[Y] - radius;
point_entry[2] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->ea_trp[1] = PO_OBJECT_ID(point_entry[2]);

/* ******************************** */
/* Occupy Position Line for 65 */
/* ******************************** */
unit = &PO_UNIT_DATA(subord_entry[5]); /* 65's Overlay */
line->overlayID = unit->overlayID;
point.overlayID = unit->overlayID;
line->pointCount = 2;
line->points[0].variant.location.x = (int32) objective_com[X] - (0.25 * radius)
* cos(parameters->orientation);
line->points[0].variant.location.y = (int32) objective_com[Y] - (0.25 * radius)
* sin(parameters->orientation);
```

```
line->points[1].variant.location.x = (int32) objective_com[X] - (0.5 * radius)
* cos(parameters->orientation);
line->points[1].variant.location.y = (int32) objective_com[Y] - (0.5 * radius)
* sin(parameters->orientation);
line_entry[1] = po_create_object(private->po_db, NULL, objectClassLine,
                                 FALSE, line,
                                 PRO_PO_LINE_CLASS_SIZE(line->pointCount),
                                 NULL);
state->plt_position[5] = PO_OBJECT_ID(line_entry[1]);

/* ****************** */
/* Left TRP for 65 */
/* ****************** */
point.location.x = (int32) objective_com[X] - 2.0 * 0.8660254 * radius;
point.location.y = (int32) objective_com[Y] + 2.0 * 0.5 * radius;
point_entry[0] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->left_trp[5]  = PO_OBJECT_ID(point_entry[0]);

/* ******************** */
/* Right TRP for 65 */
/* ******************** */
point.location.x = (int32) objective_com[X] + 2.0 * 0.8660254 * radius;
point.location.y = (int32) objective_com[Y] + 2.0 * 0.5 * radius;
point_entry[1] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->right_trp[5] = PO_OBJECT_ID(point_entry[1]);

/* ********************************** */
/* Engagement Area TRP for 65 */
/* ********************************** */
point.location.x = (int32) objective_com[X];
point.location.y = (int32) objective_com[Y] + 2.0 * radius;
point_entry[2] = po_create_object(private->po_db, NULL, objectClassPoint,
                                  FALSE, &point, sizeof(point), NULL);
state->ea_trp[5] = PO_OBJECT_ID(point_entry[2]);
}
```

APPENDIX D

```
/* ***************************************
 * SEARCH_FOR_TREE_CANOPIES
 * ************************************************************************
 * This function is used by the First Sergeant when conducting
 * his reconnaissance.  It looks for tree canopies close to his
 * position,  and determines the area of the canopy by
 * gross approximation.
 *
 * This function will reset the private->objective[3] COM and
 * will set the private->radius for the assembly area.
 *
 * Note: the 1sg will need an UPDATED Vmove to the new objective
 * COM.  Also, do we want the 1sg to travel to the com of each
 * canopy before deciding? Future Work.
 *
 * For now we will choose the canopy that is closest to the
 * company and large enough for us to fit.
 * ************************************************************
 */
static int32 search_for_tree_canopies (quad_datap, canopy_index, x, y, private)
    QUAD_DATA           *quad_datap;
    int32               canopy_index;
    float64             x, y;
    UASSEMBLY_VARS      *private;
{

 QUAD_MODEL    *canopy_ptr;
 static int32  which_canopy = 0;
 int32         i = 0, numpts = 0;
 int32         *points;
 float64       xmin = 0.0, xmax = 0.0, ymin = 0.0, ymax = 0.0;
 float64       num_l = 0.0, num_r = 0.0, num_t = 0.0, num_b = 0.0;
 float64       avg_l = 0.0, avg_r = 0.0, avg_t = 0.0, avg_b = 0.0;
 float64       aaxmin, aaxmax, aaymin, aaymax;
 float64       canopy_com[3];
 float64       the_radius = 0.0, curr_x = 0.0, curr_y = 0.0;

 /* the AA is +/- 1500.0 */

 for (i=0;i<3;i++) canopy_com[i] = 0.0;

 xmin = x - 3000.0;
 xmax = x + 3000.0;
 ymin = y - 3000.0;
 ymax = y + 3000.0;
```

```
aaxmin = x - 1500.0;
aaxmax = x + 1500.0;
aaymin = y - 1500.0;
aaymax = y + 1500.0;

num_l = 0.0;
num_r = 0.0;
num_t = 0.0;
num_b = 0.0;
avg_r = 0.0;
avg_l = 0.0;
avg_t = 0.0;
avg_b = 0.0;

canopy_ptr = quad_datap->canopies.model_ptr + canopy_index;
numpts = 0;

/*
```



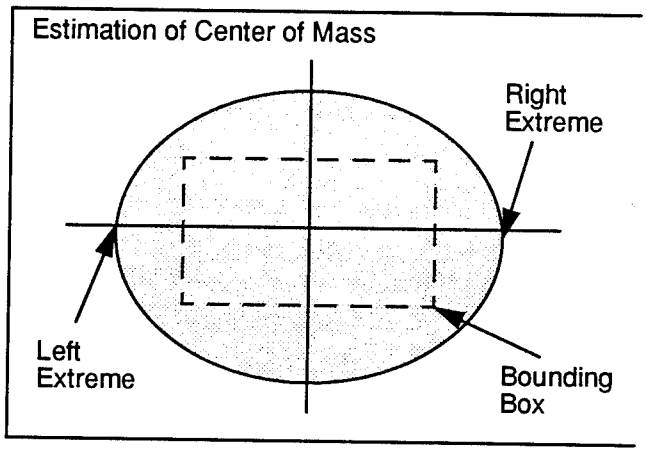

```
/* In the diagram above, we want to consider the canopies that have
* at least one of its extremes within the outer box. We can
* immediately reject canopy #1 since it has none of its extremes
* within the outer box (x_min, x_max, y_min, y_max). We will
* consider 2, 3, 4, and 5 as candidate canopies. We next check
* if there are any points of the candidate canopies within the inner
* box. If there are no points within the inner box for the candidate
* canopy, it is rejected. We therefore eliminate canopies 4 and 5.
* That leaves us canopies 2 and 3. Canopy 2 COM and avg_left, right
* top and bottom are determined using the bounding box algorithm. */
```

```
/* This canopy has one of its extremes within the outer box */

 if ((((float64)canopy_ptr->x_min >= xmin) &&
    ((float64)canopy_ptr->x_min <= xmax)) ||
   (((float64)canopy_ptr->x_max <= xmax) &&
    ((float64)canopy_ptr->x_max >= xmin)))

  {
   if((((float64)canopy_ptr->y_min >= ymin) &&
      ((float64)canopy_ptr->y_min <= ymax)) ||
     (((float64)canopy_ptr->y_max <= ymax) &&
      ((float64)canopy_ptr->y_max >= ymin)))
    {
     printf ("\nTree canopy #%d\n", canopy_index);
     printf ("Points (%d):\n", canopy_ptr->num_pts);

     if((canopy_ptr->x_min < aaxmin) ||
       (canopy_ptr->x_max > aaxmax) ||
       (canopy_ptr->y_min < aaymin) ||
       (canopy_ptr->y_max > aaymax))
      {
       printf("This canopy extends beyond AA box\n");
       /* only include those points strictly within the AA box */
       /* Are there any points within the AA box? */
       for (i=0, points = canopy_ptr->points; i<canopy_ptr->num_pts;
          i+=2, points+=2)
        {
         curr_x = (float64)*points;
         curr_y = (float64)*(points+1);

         /* printf ("  x,y = %f, %f\n", (float64)*points,(float64)*(points+1)); */
         if((curr_x < aaxmin) ||
           (curr_x > aaxmax) ||
           (curr_y < aaymin) ||
           (curr_y > aaymax))
          {
           printf(" ");
           /* printf("Not including this point of canopy\n"); */
          }
         else
          {
           canopy_com[X] += (float64)*points;
           canopy_com[Y] += (float64)*(points+1);
           numpts++;
           /* printf("total xpts = %f, total ypts = %f\n",
            *     canopy_com[X], canopy_com[Y]);
            */
          }
        }
```

```
/* DEALING HERE WITH A CANOPY EXTENDING BEYOND AA BOX */
        if(numpts)
         {
          canopy_com[X] /= (float64)numpts;
          canopy_com[Y] /= (float64)numpts;

          for (i=0, points = canopy_ptr->points; i<canopy_ptr->num_pts;
              i+=2,  points+=2)
           {
            if(((float64)*points >= canopy_com[X]) &&
              ((float64)*points < aaxmax))
             {
              avg_r += (float64)*points;
              num_r++;
             }
            if(((float64)*points <= canopy_com[X]) &&
              ((float64)*points > aaxmin))
             {
              avg_l += (float64)*points;
              num_l++;
             }
            if(((float64)*(points+1) >= canopy_com[Y]) &&
              ((float64)*(points+1) < aaymax))
             {
              avg_t += (float64)*(points+1);
              num_t++;
             }
            if(((float64)*(points+1) <= canopy_com[Y]) &&
              ((float64)*(points+1) > aaymin))
             {
              avg_b += (float64)*(points+1);
              num_b++;
             }
           }

          avg_r /= num_r;
          avg_l /= num_l;
          avg_t /= num_t;
          avg_b /= num_b;
         }
        else
         {
          printf("\nThis canopy is totally outside AA Box -- rejecting it\n");
          return FALSE;
         }
       }
      else
       {
        /* DEALING HERE WITH A CANOPY WITHIN THE AA BOX */
        printf("\nThis canopy is within the AA box\n");
        /* First we need to figure out the center of mass */
```

```
    for (i=0, points = canopy_ptr->points; i<canopy_ptr->num_pts;
        i+=2, points+=2)
     {
      /* printf ("  x,y = %f, %f\n", (float64)*points,(float64)*(points+1)); */
      canopy_com[X] += (float64)*points;
      canopy_com[Y] += (float64)*(points+1);
      /* printf("total xpts = %f, total ypts = %f\n",
       *     canopy_com[X], canopy_com[Y]);
       */
     }
    canopy_com[X] /= (0.5 * (float64)canopy_ptr->num_pts);
    canopy_com[Y] /= (0.5 * (float64)canopy_ptr->num_pts);

    for (i=0, points = canopy_ptr->points; i<canopy_ptr->num_pts;
        i+=2,  points+=2)
     {
      if((float64)*points >= canopy_com[X])
       {
        avg_r += (float64)*points;
        num_r++;
       }
      if((float64)*points <= canopy_com[X])
       {
        avg_l += (float64)*points;
        num_l++;
       }
      if((float64)*(points+1) >= canopy_com[Y])
       {
        avg_t += (float64)*(points+1);
        num_t++;
       }
      if((float64)*(points+1) <= canopy_com[Y])
       {
        avg_b += (float64)*(points+1);
        num_b++;
       }
     }

   avg_r /= num_r;
   avg_l /= num_l;
   avg_t /= num_t;
   avg_b /= num_b;
 }

printf ("x_min,y_min = %d, %d  x_max,y_max = %d, %d\n",
      canopy_ptr->x_min,
      canopy_ptr->y_min, canopy_ptr->x_max, canopy_ptr->y_max);

printf ("Height = %d\n", canopy_ptr->height);
printf ("Impenetrable = %d\n", canopy_ptr->impenetrable);
```

```
        printf ("COM X = %f, COM Y = %f\n", canopy_com[X], canopy_com[Y]);

        /* Before we continue let's see if the com of this canopy is within our
         * assy area bounding box.
         */

/* Now lets get the average bounding box for this canopy
*
* We do this by:
*
```




```
*
* Find the average x value in the right section.
* Find the average x value in the left section.
* Find the average y value in the top section.
* Find the average y value in the bottom section.
*
* This will give us a gross box with which we can calculate an
* approximate area.
*/

        printf("Xpts R of COM: %f\n", num_r);
        printf("Xpts L of COM: %f\n", num_l);
        printf("Ypts T of COM: %f\n", num_t);
        printf("Ypts B of COM: %f\n", num_b);
        printf("The tree canopy averages are:\ntop: %f\nbottom: %f\nright: %f\nleft: %f\n",
             avg_t, avg_b, avg_r, avg_l);

        /* Will need an array structure to hold the following information
         * obtained from above:
         * The number of canopies found
         * The center of mass of each canopy
         * The minimum radius of each canopy
         */
```

```
        /* For our selection process, we need to refine
         * whether the canopy com is within our user
         * specified bounding box
         */

        /* find the closest canopy to the company */

        /* determine the minimum radius */
        the_radius = 1.0E+100;
        if ((canopy_com[X] - avg_l) < the_radius)
          the_radius = (canopy_com[X] - avg_l);
        if ((avg_r - canopy_com[X]) < the_radius)
          the_radius = (avg_r - canopy_com[X]);
        if ((avg_t - canopy_com[Y]) < the_radius)
          the_radius = (avg_t - canopy_com[Y]);
        if ((canopy_com[Y] - avg_b) < the_radius)
          the_radius = (canopy_com[Y] - avg_b);
        if(the_radius > private->radius &&
           the_radius >= 30.0)
         {
           private->radius = the_radius;
           which_canopy = canopy_index;
           private->canopy_com[X] = canopy_com[X];
           private->canopy_com[Y] = canopy_com[Y];
         }
        return TRUE;
      }
     else
      return FALSE;
   }
  else
   return FALSE;
}
```

LIST OF REFERENCES

[Booker et al, 93] Booker, Brooks, Garrett, Giddings, Salisbury, Worley, 1993 DMSO Survey of Semi-Automated Forces, July 30, 1993, Defense Modeling and Simulation Office (DMSO), University of Central Florida, Orlando, Florida.

[Calder et al, 93] Calder, Smith, Courtemanche, Mar, Ceranowicz, ModSAF Behavior Simulation and Control, Third Conference on Computer Generated Forces and Behavioral Representation, March 17-19, 1993, University of Central Florida, Orlando, Florida.

[Ceranowicz, 93-1] Ceranowicz, Andrew Z., Modular Semi-Automated Forces, Modular Semi-Automated Forces: Recent and Historical Publications, May 13, 1994, Loral Advanced Distributed Simulation, Cambridge, Massachusetts.

[Ceranowicz, 93-2] Ceranowicz, Andrew Z., ModSAF and Command and Control, Modular Semi-Automated Forces: Recent and Historical Publications, May 13, 1994, Loral Advanced Distributed Simulation, Cambridge, Massachusetts.

[Ceranowicz, 94] Ceranowicz, Andrew Z., ModSAF Capabilities, Fourth Conference on Computer Generated Forces and Behavioral Representation, May 4-6, 1994, University of Central Florida, Orlando, Florida.

[Ceranowicz et al, 94] Ceranowicz, Coffin, Smith, Gonzalez, Ladd, Operator Control of Behavior in ModSAF, Fourth Conference on Computer Generated Forces and Behavioral Representation, May 4-6, 1994, University of Central Florida, Orlando, Florida.

[Culpepper, 92] Culpepper, Michael E., Tactical Decision Making in Intelligent Agents: Developing Autonomous Forces in NPSNET, Masters Thesis, March 1992, Naval Postgraduate School, Monterey, California.

[Hearne, 93] Hearne, John Henry, Jr., NPSNET: Physically Based, Autonomous, Naval Surface Agents, Masters Thesis, September 1993, Naval Postgraduate School, Monterey, California.

[Mathews, 93] Mathews, William., Budget war: Aspin drops $2.5 billion bomb, Army Times, February 15, 1993, Army Times Publishing Company, Springfield, Virginia.

[ModSAF, 94] ModSAF User Manual, Version 1.0, Loral Advanced Distributed Simulation, Inc., March 2, 1994, Cambridge, Massachusetts.

[ModSAF Libech, 93] ModSAF Libechelondb Programmer's Guide, Loral Advanced Distributed Simulation, Inc., June 14, 1993, Cambridge, Massachusetts.

[ModSAF Libpocc, 93] ModSAF Libupoccpos Programmer's Guide, Loral Advanced Distributed Simulation, Inc., August 23, 1993, Cambridge, Massachusetts.

[Robasky, 94-1] Robasky, Kim, ModSAF 1.0 Developer's Course Handouts, LADS Document No. 94017 v. 1.01, May 9, 1994, LORAL Advanced Distributed Simulation, Cambridge, Massachusetts.

[Robasky, 94-2] Robasky, Kim, ModSAF 1.0 Developer's Class Work Book, LADS Document No. 94006 v. 1.01, June 24, 1994, LORAL Advanced Distributed Simulation, Cambridge, Massachusetts.

[Rosenbloom et al, 94] Rosenbloom, Johnson, Jones, Koss, Laird, Lehman, Rubinoff, Schwamb, Intelligent Automated Agents for Tactical Air Simulation: A Progress Report, Fourth Conference on Computer Generated Forces and Behavioral Representation, May 4-6, 1994, University of Central Florida, Orlando, Florida.

[Smith, 93] Smith, J. E., ModSAF LibTask Programmer's Guide: LibTask, ModSAF Documentation, 1993, Loral Advanced Simulation, Cambridge, Massachusetts.

[Standard, 93] Standard for Information Technology - Protocols for Distributed Interactive Simulation Applications, Version 2.0, Third Draft, 28 May 1983, STRICOM, DMSO, Orlando, Florida.

[Stanzione, 89] Stanzione, T., Terrain Reasoning in the SIMNET Semi-automated Forces System, BBN Systems and Technologies Corp., October 1989, Cambridge, Massachusetts.

[Stanzione et al, 93] Stanzione, Smith, Brock, Mar, Calder, Terrain Reasoning in the ODIN Semi-Automated Forces System, Third Conference on Computer Generated Forces and Behavioral Representation, March 17-19, 1993, University of Central Florida, Orlando, Florida.

[USA FM 71-1, 88] Headquarters, Department of the Army, Field Manual No 71-1, Tank and Mechanized Infantry Company Team, November 1988, Washington, D.C.

[USA ARTEP 71-1-MTP, 88] Headquarters, Department of the Army, ARTEP 71-1-MTP, Mission Training Plan for the Tank and Mechanized Infantry Company and Company Team, October 1988, Washington, D.C.

[Van Brackle et al, 93] Van Brackle, Petty, Gouge, Hull, Terrain Reasoning for Reconnaissance Planning in Polygonal Terrain, Third Conference on Computer Generated Forces and Behavioral Representation, March 17-19, 1993, University of Central Florida, Orlando, Florida.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5101	2
3.	Department Chairman, Code CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor David R. Pratt, Code CS/Pr Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	4
5.	Professor Michael J. Zyda, Code CS/Zk Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	2
6.	Artificial Intelligence Center HQDA, OCSA ATTN: DACS-DMA Naval Postgraduate School Pentagon, RM 1D659 Washington, DC 20310-0200	1
7.	U.S. Army Simulation, Training, and Instrumentation Command ATTN: Mr. Stan Goodman 12350 Research Parkway Orlando, FL 32826-3276	1
8.	Major Gary M. McAndrews 120 Powell Drive Leesville, LA 71446	1