NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

STAFFSIM, AN INTERACTIVE SIMULATION FOR RAPID, REAL TIME COURSE OF ACTION ANALYSIS BY U.S. ARMY BRIGADE STAFFS

by

William E. Bohman

June 1999

Thesis Advisor:	Arnold H. Buss
Thesis Co-Advisor:	Bard Mansager

Approved for public release; distribution is unlimited.

19991207 034

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1999	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE STAFFSIM, AN INTERACTIVE SIMULATION FOR RAPID, REAL TIME COURSE OF ACTION ANALYSIS BY U.S. ARMY BRIGADE STAFFS	5. FUNDING NUMBERS
6. AUTHOR(S) William E. Bohman	
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 U.S. Government.

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

13. ABSTRACT *(maximum 200 words)*

The U.S. Army has fielded a wide range of simulations for tactical units. The purpose of these simulations range from training individual skills to collective training for corps staffs. Currently fielded simulations are not designed for operational use. Most are operated by contract civilian personnel and require fixed base facilities. Furthermore, many of these simulations require extensive lead-time to initiate useable scenarios. When the army rolls to the field, its simulations are left behind.

The army's staff planning process places huge cognitive demands unit staffs, often resulting in sub-optimal decision making. Simulations can provide a useful tool to help staffs visualize and understand complex time-space relationships and unit interactions. Eliminating the need for these factors to be visualized in the mind's eye allows staffs to focus their cognitive abilities on synchronizing mission plans.

This thesis develops a prototype simulation for operational use by brigade staffs. The simulations purpose is course of action analysis as described in the war gaming step of the staff planning process. To be used operationally, the simulation must be easy to use, provide rapid scenario development, enable fast course of action analysis and run on a personal computer. To meet these requirements the simulation presented in this thesis is built using reusable software components and loosely coupled program modules.

14. SUBJECT TERMS Software Components, Staff Planning Process, Simulation, Loosely Coupled Software Components	15. NUMBER OF PAGES 125
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFI-CATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

STAFFSIM, AN INTERACTIVE SIMULATION FOR RAPID, REAL TIME COURSE OF ACTION ANALYSIS BY U.S. ARMY BRIGADE STAFFS

William E. Bohman
Major, United States Army
B.S., University of Cincinnati, 1987

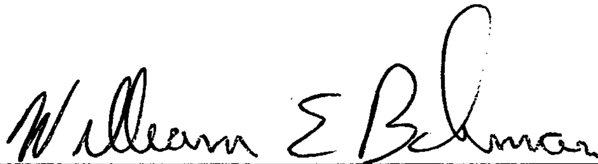
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING VIRTUAL ENVIRONMENTS AND SIMULATION

from the

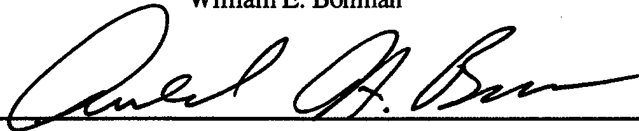
NAVAL POSTGRADUATE SCHOOL
June 1999

Author:

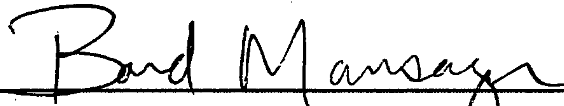


William E. Bohman

Approved by:



Arnold H. Buss, Thesis Advisor



Bard Mansager, Thesis Co-Advisor

Michael Zyda, Academic Associate
Modeling Virtual Environments and Simulation Academic Group

Michael Zyda, Chairman
Modeling Virtual Environments and Simulation Academic Group

ABSTRACT

The U.S. Army has fielded a wide range of simulations for tactical units. The purpose of these simulations range from training individual skills to collective training for corps staffs. Currently fielded simulations are not designed for operational use. Most are operated by contract civilian personnel and require fixed base facilities. Furthermore, many of these simulations require extensive lead-time to initiate useable scenarios. When the army rolls to the field, its simulations are left behind.

The Army's staff planning process places huge cognitive demands on unit staffs, often resulting in sub-optimal decision making. Simulations can provide a useful tool to help staffs visualize and understand complex time-space relationships and unit interactions. Eliminating the need for these factors to be visualized in the mind's eye allows staffs to focus their cognitive abilities on synchronizing mission plans.

This thesis develops a prototype simulation for operational use by brigade staffs. The simulations purpose is course of action analysis as described in the war gaming step of the staff planning process. To be used operationally, the simulation must be easy to use, provide rapid scenario development, enable fast course of action analysis and run on a personal computer. To meet these requirements the simulation presented in this thesis is built using reusable software components and loosely coupled program modules.

TABLE OF CONTENTS

I. INTRODUCTION 1

A. MOTIVATION 1
B. BACKGROUND 3
C. FIELDED SIMULATIONS 5
1. Janus 6
2. Brigade/Battalion Battle Simulation BBS 8
3. Corp Battle Simulation CBS 9
D. COMBAT TRAINING CENTERS 10
E. SUMMARY 11
F. SUMMARY OF CHAPTERS 12

II. THE MILITARY DECISION MAKING PROCESS (MDMP) 15

A. PURPOSE OF THE MDMP 15
B. METHODOLOGY 15
1. Mission Analysis 16
2. Course of Action Development 16
3. Course of Action Analysis 17
C. COURSE OF ACTION ANALYSIS IN PRACTICE 19
D. POTENTIAL ROLE OF SIMULATION IN THE MDMP 21

III. STAFF SIMULATION (STAFFSIM) 23

A. INTRODUCTION 23
B. SOFTWARE COMPONENTS 24
C. STAFFSIM COMPONENTS 26
1. The Components 26
2. Component Communication 30
D. SUMMARY 33

IV. BATTLE SIMULATION (BATTLESIM) 35

A. INTRODUCTION 35
B. BUILDING BLOCK COMPONENT MODELS 37
1. Mover and BasicMover 37
2. Sensor and BasicSensor 38
3. Weapon and BasicWeapon 43
4. FireControl 43
C. COMPONENT CONTAINERS 47
1. Vehicle 47
2. Unit 49
D. COMPONENT INTERACTIONS 50
1. Introduction 50
2. Event Handling 51
a. Listeners *51*
b. Event Classes *52*
c. Event Scheduling *53*
E. SUMMARY 54

V. STAFFSIM IMPLEMENTATION **55**

A. INTRODUCTION 55
B. SCENARIO DEVELOPMENT 58
1. Order of Battle 58
a. Opposing Orders of Battle *58*
b. Order of Battle Input to STAFFSIM *60*
2. Courses of Action 60
a. OPFOR Course of Action *61*
b. Friendly Course of Action *61*
c. Course of Action Input to STAFFSIM *63*
C. SCENARIO EXECUTION 64
D. STAFFSIM VERSUS SIMULATION REQUIREMENTS 69
E. SUMMARY 72

VI. CONCLUSIONS **75**

A. CONCLUSIONS 75
B. FUTURE WORK 76
1. High Resolution Combat Models 76
2. Battlefield Operating Systems 76
3. System Performance 77
4. Field Experimentation 77
C. SUMMARY 77

APPENDIX A: SELECTED IMPLEMENTATION CODE LISTINGS **79**

APPENDIX B: ACROYMNS **105**

LIST OF REFERENCES **107**

INITIAL DISTRIBUTION LIST **111**

LIST OF FIGURES

Figure 3.1: STAFFSIM Components 26
Figure 3.2: Flora 27
Figure 3.3: SimBuilder Displaying the CompanyBuilder Panel 28
Figure 3.4: ExecutiveOfficer Configured to Build Movement Orders 29
Figure 3.5: Component Interface Flow Chart Diagram 32
Figure 3.6: STAFFSIM Component Communication 33
Figure 4.1: Component Communication Within a Container 35
Figure 4.2: BattleSim Component Types 36
Figure 4.3: Event Graph Snippet for Movement Event Scheduling 38
Figure 4.4: Smooth Linear and Linear Acceleration Movement Models 38
Figure 4.5: Interplay of Vehicle Sensors, the Registrar and the Mediators 39
Figure 4.6: Event Graph of the Detection Sequence 41
Figure 4.7: Detection Sequence Model 42
Figure 4.8: Event Graph of the Engagement Sequence 45
Figure 4.9: FireControl Decision Flow Chart 46
Figure 4.10: BattleSim Component Interactions 50
Figure 4.11: BattleSim Event Hierarchy 53
Figure 4.12: BattleSim Event Scheduling 54
Figure 5.1: OPFOR Order of Battle 59
Figure 5.2: Friendly Forces Order of Battle 59
Figure 5.3: OPFOR Course of Action 62
Figure 5.4: Friendly Course of Action 62
Figure 5.5: Assigning Unit Orders with ExecutiveOfficer 63
Figure 5.6: CRPs Enter Sector and Make Contact 64
Figure 5.7: AGMB Assaults 65
Figure 5.8: AGMB Penetrates the Defense 66
Figure 5.9: Main Body Penetrates the Defense 68

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my thesis advisors, Dr. Arnold Buss and LTC (Ret.) Bard Mansager for their wise counsel, patience and dedication throughout my work on this thesis.

I would specially like to thank Dr. Michael Zyda, the creator of the Modeling, Simulations and Virtual Environments (MOVES) curriculum at the Naval Postgraduate School. Dr Zyda is an outstanding educator and visionary in his field. He has provided the insight necessary to understand the complex problems facing the army's modeling and simulation community while providing the analytic and cognitive tools necessary to find workable solutions for those problems.

Finally, I must recognize the loving support and self-sacrifice of my wife Hyun. She has endured countless late nights, absent weekends and postponed family time all without complaint. Without her understanding and support this work could not have gotten off the ground.

I. INTRODUCTION

A. MOTIVATION

The old campaigner's dictum that "in war all things are simple, but the simplest of things is extremely difficult" is certainly truer today than at any other point in history. The complexity and variety of military equipment and doctrine has grown exponentially over the past hundred years. Equally confounding are the speed at which operations can be conducted and the extreme distances over which a strike can be delivered. The net effect of these developments have combined to give an antagonist a myriad of potential options while compressing the time available to consider them into impossibly short decision cycles.

A brigade commander in today's army must employ a wide variety of weapons and combat multipliers in order to accomplish the mission. These weapons range in complexity from automatic rifles to multi-million dollar tanks, helicopters, and jet aircraft. On the battlefield, however, weight of numbers or degree of technological sophistication alone cannot produce victory. Soldiers win battles. The soldier who can out-think and out-fight his opponent is usually victorious. The brigade commander's job is to not only lead soldiers into battle but also to employ soldiers and their weapons at the right place and at the right time in order to produce the best possible outcome. Thus, commanders must decide how to best employ the weapons and soldiers under their command.

The art of properly employing soldiers and weapons is not easily learned. The real difficulty comes from employing the pieces of the team such that the whole is greater than the sum of the parts. When a brigade combat team is properly employed it is said to have synchronized itself for the mission. The complex task of synchronizing a brigade during a combat mission falls upon commanders and their staffs. These officers must ensure each element of the brigade is properly employed. Brigade units must be given missions that are supported by and in turn support other brigade units. Each unit must be utilized to magnify its strengths and mask its weaknesses. When employed in this manner, a brigade is a synchronized team and the potential for success is very high. Otherwise, the outcome of the battle could come down to the flip of a coin or worse.

In its capstone doctrinal manual, FM 100-5 Operations [1], the army has described synchronization as follows.

> **Synchronization is arranging activities in time and space to mass at the decisive point. ... It means that the desired effect is achieved by arranging activities in time and space to gain that effect. Synchronization includes, but is not limited to, the massed effects of combat power at the point of decision. ... Synchronization usually requires explicit coordination among the various units and activities participating in any operation. By itself, however, such coordination is no guarantee of synchronization unless commanders first visualize the consequences to be produced and how they sequence activities to produce them. ... Synchronization thus takes place first in the minds of commanders and then in the actual planning and coordination of movements, fires, and supporting activities.**

FM 100-5 concludes its discussion of synchronization with a clear statement of the purpose, or End State, of synchronization.

> **In the end, the product of effective synchronization is maximum use of every resource to make the greatest contribution to success. ... To achieve this requires the anticipation that comes with thinking in depth, mastery of time-space-purpose relationships, and a complete understanding of the ways in which friendly and enemy capabilities interact.**

In the final analysis, the brigade's fight must be synchronized if victory is to be assured. Conversely, failure to synchronize can result in battlefield defeat. For soldiers at the sharp end, life itself hangs in the balance. How, then, is synchronization achieved? How do army staffs solve the synchronization problem, if at all? Can simulation be used to help staffs achieve synchronization?

These questions are of central interest for any army that hopes to win on the modern battlefield. The complex sophistication of modern equipment combined with the decreasing time available in which to properly analyze tactical options push staffs towards incomplete analysis. Incomplete or hasty analysis provides for poor decision making. In essence, the weight of the clock drives decision making instead of rigorous analysis.

For a military organization poor decision making is an unacceptable state of affairs. In an army that is adverse to casualties, poor decision making cannot be tolerated. The real question is whether proper analysis can be completed in the time available and, if so, how? To answer these questions an understanding of how unit commanders and their staffs arrive at tactical decisions and the tools they use is required. To answer the question as to whether a simulation can be used as a decision-making tool, this thesis presents a prototype of such a simulation and evaluates its utility in speeding the decision making process and improving the quality of the analysis.

B. BACKGROUND

Brigade commanders and their staffs achieve synchronization by carefully planning missions within the framework of the Military Decision-Making Process (MDMP). When a brigade-sized unit receives a mission the commander and staff must develop and implement a plan to accomplish said mission. The template they follow to arrive at the best possible course of action is the MDMP. The MDMP is broken down into a seven-step sequence [2].

Step 1. Receipt of Mission.

Step 2. Mission Analysis.

Step 3. Course of Action Development.

Step 4. Course of Action Analysis.

Step 5. Course of Action Comparison.

Step 6. Course of Action Approval.

Step 7. Orders Production.

This process is meant to provide a logical framework that allows commanders and their staffs to rapidly arrive at, and execute, a course of action [3].

Within the course of action analysis step of the MDMP, the tool commanders and staffs use to analyze, and thus synchronize, their developed course of action is the war game. Within the context of the MDMP the war game is a mental exercise in which the assembled staff officers simulate how they believe a course of action will unfold. One or more officers role-play the enemy commander and fight a potential enemy course of action against the friendly course of action being analyzed. In general, war games are conducted around a map board. Staff officers move unit icons across the map simulating the enemy and friendly course of action. When opposing units come in contact, each officer involved attempts to visualize in his mind's eye how the battle will unfold. He tries to find shortcomings in his own plan that the enemy might exploit and then corrects the foreseen problems. Conversely, he looks for shortcomings to exploit in the enemy course of action as well. The war game is thus a comparison of each potential friendly course of action against each possible enemy course of action [4].

The actual resolution of battles during the war game is discussed among the staff without the use of any analytical tools. For example, when two opposing forces come into contact, the officer playing the enemy force might propose that, based on relative size of

forces, the friendly unit would be destroyed with the loss of a portion of the enemy force. The officers then debate the merit of the suggestion while discussing options open at this point to each commander. The product of the debate is not only who loses what, but also a better understanding of what can potentially happen at this point in the battle and what must be done for friendly forces to be successful. Better understanding leads to better synchronization and better decision making.

It is easy to see that the war game places great cognitive demands on commanders and staffs, since they must visualize the complex interaction of weapons and units in time and space. To properly synchronize the course of action during the war game, staff officers must fully understand time-space relationships, friendly and enemy unit capabilities/weaknesses, and probable outcomes of friendly/enemy unit interactions. Commanders and staffs must master the impact of time and space factors on the battlefield. The arrangement of activities in time and space is a key challenge facing commanders today [5].

During the war game, can the debate about 'who shot John' be replaced with a computer simulation that produces a probable outcome? Inserting a simulation into the war game as an analytical tool can potentially reduce the cognitive burden on staff officers in two important ways. First, the mental visualization a staff officer must currently develop of the battle in his mind's eye is replaced by the visualization presented by the simulation. Second, the actual thought processes of evaluating the interplay of weapons, units, terrain and time are replaced by the combat modeling of the simulation. The reduction of cognitive workload will allow officers to focus their mental powers on synchronizing the plan and will provide them the extra time needed to do the job right.

Can a simulation replace the mapboard and unit icons of today's war game? Can human estimates of unit abilities be replaced with accurate computer models of those units? Can the subjective be replaced with the objective to improve analysis in the war game? Instead of officers estimating the probable outcomes of battles, can those battles be modeled accurately in a computer simulation? In other words, can the subjective judgement of humans be replaced with probabilistic combat models that significantly reduce cognitive workload? Are such simulations already available in the army's training base today?

The army has fielded a large variety of simulations for use by tactical units. The purpose of these simulations run the gamut from training rifle marksmanship to staff training for brigade, division and corps staffs. The purpose of all these simulations is to train soldiers. The key point is that they are not for operational use. When units deploy, simulations are left behind. Thus the world's most technologically advanced military machine enters combat with its staffs using paper, pencil, acetate and colored markers as the primary tools with which to develop and analyze courses of action. Can one of the currently fielded simulations be easily adapted for operational use as a war game? In the following section we will examine some of the existing combat simulations and assess their suitability for this proposed usage.

C. FIELDED SIMULATIONS

Before examining the simulations currently in use by the army it is important to understand some of the constraints under which brigade staffs operate. The primary constraint is time. How much time do brigade staffs have to conduct a war game and thus analyze a course of action? The army's doctrinal manuals do not specify an amount of time to allocate to the war game. However, based on unit experiences in the field, the army's Center for Lessons Learned (CALL) has published example time lines for the MDMP. Those timelines allocate between one and three hours for the war game [18]. Any simulation must therefore be marshaled and analyzed within that time frame as well.

Brigade staffs labor under other significant constraints as well. Staffs do not have trained computer technicians nor network experts available. The computers available are generally mid-grade personal computers (PCs) at least one generation old. The amount of available electricity is fixed and cannot support a large computer infrastructure. Even if power was available, space is at a premium. Brigade headquarters are mobile entities that are frequently packed, moved and quickly re-established at a new location. The load carrying capacity of the headquarters vehicles is fixed. When moved, brigade headquarters must be fully functional in a matter of hours, usually less than four.

These constraints easily translate into a baseline of requirements that a simulation must meet if it is to be used in operational war gaming situations. Military officers who are not computer literate must be able to easily use the simulation. At no time must specially

trained technicians, civilian or military, be required. The simulation must not be static, but interactive, allowing staffs to stop the action, rewind and explore the course of action in detail. The scenario must be easily changed on the fly. The simulation must provide this functionality while meeting the three-hour time constraint for the war game.

In terms of physical requirements the overriding factor is that it must be hosted on PCs currently used by brigade staffs. The simulation must run on these machines without displacing other applications. The only acceptable modifications to these machines to accommodate the simulation must be inexpensive. Upgrades such as larger hard drives, improved graphics cards and additional memory are acceptable. The ideal situation would be a simulation that can run on the latest generation laptop computers or possibly even one of the new generation of handheld personal assistants.

It must also be remembered that when and where a unit may be committed to combat is completely unpredictable. For a simulation to be used in the war game it must be able to integrate new terrain and unit databases in a very short time. For example, the ready brigade of the 82d Airborne Division could be committed to combat from its barracks at Ft. Bragg, North Carolina, in less than twenty-four hours. Following brigades can be in action days later. Clearly, it is essential that any simulation that expects operational use be able to add or update databases in a matter of hours.

While reviewing the simulations currently fielded it will be instructive to keep these constraints/requirements in mind. The objective is to find a simulation that can be easily adapted for field use in the war game, or failing that to conclude that a new simulation is needed. We will examine three simulations, Janus, Brigade/Battalion Simulation and Corps Battle Simulation.

1. Janus

Janus is an interactive simulation originally developed at the Lawrence Livermore National Laboratory to model nuclear effects. Janus has since evolved into three main versions used extensively by both the combat developments and training communities within the army [7]. The Janus simulation is an interactive, high-resolution model of ground combat at the entity level. Entities within the simulation represent individual soldiers, tanks, aircraft

etc. In the training mode, Janus allows staffs at the brigade and battalion level to train synchronization of the Battlefield Operating Systems (BOS) [8].

The hardware requirements to run Janus are substantial. Janus is a networked over a thin wire Ethernet. Its standard configuration consists of two sets of Hewlett Packard (HP) 715/50 workstations. Each set consists of eight workstations with a ninth providing host services. The minimum possible configuration is two workstations, one for each side [8].

Janus is currently undergoing several major upgrades. The HLA Warrior project involves updating the software architecture and porting the source code from Fortran 77 to C++. The target host computer for the new system is a Pentium 133 PC [8]. A second initiative to move Janus to PCs is currently being fielded to National Guard units. This version hosts the simulation on notebook computers running the LINUX operating system [16].

In most fielded configurations, Janus requires some degree of contract civilian support staff to operate and maintain the simulation. The LINUX version requires the least support staff overhead. The National Guard units fielding this version receive New Equipment Training (NET) when fielded then assumes complete responsibility for operating and maintaining the simulation [16]. At the other extreme, many active duty component installations have as many as one civilian technician per workstation.

Terrain databases are a another shortcoming of Janus with respect to operational use. Currently, there are approximately 286 terrain databases ranging in size from 7 by 7 kilometers (km) to 100 by 100 km. Terrain databases require one to two days to develop and place into play. The databases are developed from digital terrain data provided by the National Imagery and Mapping Agency (NIMA) [16].

Janus takes a significant amount of time to configure for a specific scenario. Inputting the units for a brigade size fight can take upwards of three days. Once the simulation has been populated it can then take an additional two to eight hours to position the units and assign them their initial orders [16].

Because of these drawbacks, Janus is not a candidate for operational use in the field. Although Janus can be hosted on PCs and terrain databases can be developed with relative ease, the time requirements to populate and initialize scenarios is too great. The primary driver for these long lead times in the basic entity size. Deploying, orienting and assigning

orders to each individual vehicle in a brigade sized unit and its opposing enemy unit is a tedious and time consuming task that takes much longer than the three hours the brigade staff has to conduct the wargame.

2. Brigade/Battalion Battle Simulation (BBS)

BBS was designed to be a Command Post Exercise (CPX) driver for brigade and battalion staffs. BBS allows commanders to conduct exercises for the training of staff procedures and integration [10]. BBS uses high-resolution combat models to simulate the interplay of combat units from single vehicle through brigade in size. The basic level entity is an individual vehicle. Like Janus, BBS is interactive and models a very wide range of activities typically found on the modern battlefield including air, ground, and a variety of logistics operations [11].

Like Janus, the hardware requirements to run BBS are extensive. Five Digital Equipment Corporation (DEC) Microvax 3100 computers support ten workstations in the standard configuration [11]. Each workstation consists of three DEC VT320 terminals, a printer, an Amiga HD PC for graphic overlays, a laser video disc player for the terrain model and a 26 inch color monitor [10].

The initial terrain model used in BBS suffered from the same availability problems Janus terrain does. The estimated cost to develop a new terrain database is in the range of $150, 000 and requires approximately six months to complete [12]. The latest version of BBS has significantly improved both the cost and time required to develop new terrain databases. The laser videodisc format has been replaced with a digital terrain model based on digital terrain products readily available NIMA. With the digital terrain model a new database can be developed in as little as three weeks for an average cost of between $12,000 and $15,000 [12].

Can BBS be modified for operational use by a brigade staff? Undoubtedly the system could be completely redesigned to meet the requirements but the cost to do so would be high. Although turn around time for new terrain databases has improved significantly it is still too slow for operational use. Furthermore, new databases require outside support to develop. Brigade staffs do not have the time required to do this. To be used operationally terrain database creation must be simple enough that brigade staffs can build new ones directly from

NIMA products without third party assistance. The hardware requirements for BBS fall completely outside the ability of brigade staffs to transport and install. To support such a system addition vehicles and personnel would have to be added to the unit tables of organization. In the end, BBS is an excellent training system but its utility ends there.

3. Corps Battle Simulation (CBS)

CBS is the army's division and corps staff trainer. Like BBS, CBS is primarily used by the army as a CPX driver. Unlike Janus and BBS, CBS does not employ high-resolution combat models. Instead, CBS uses an attrition combat model based on Lanchester equations [13]. The size of the basic entity in CBS is the battalion. CBS models ground combat, rotary and fixed wing aviation, logistics and special forces [14]. Although CBS is targeted for staffs at echelons above brigade, most brigade and battalion staff officers have participated in multiple CBS driven exercises. Because CBS uses a different system of combat modeling, Lanchester equations versus high-resolution models, it is useful to study its feasibility for use at lower echelons.

The CBS hardware suite is fairly extensive. CBS is a networked simulation run over a local or wide area network. The simulation is hosted on a DEC VAX 7620 computer. The host is networked with multiple MicroVAX 3100/40 computers each of which support up to three workstations. A workstation is typically configured with a television monitor, graphics pad, laser video disc player, graphics generator, printer and three video terminals. A recent system upgrade has replaced the VAX 3100/40 computers with VAX 3100/85 computers. The new computer can support up to six workstations. A typical division level exercise requires approximately 60 to 75 workstations [17].

CBS requires support staff to setup and run the simulation. To execute a generic division level simulation, a minimum of four trained technicians is required per shift. This figure assumes that all the workstations and the host computer are co-located. The setup time for just the equipment is approximately two man-hours per workstation. The lead-time to populate the simulation with the correct mix of units can be as long as 30 days. However, in our circumstances it can be assumed that the unit database is already built. Once the unit database is established it can take upwards of 100 man-hours to position units and assign missions [17].

The terrain databases for CBS are extremely limited. There are currently seventeen such databases [15, 17]. The lead-time to develop new terrain databases can be as much as six months with an average cost between $50,000 and $100,000 [17].

CBS does not appear to be a good candidate for operational use by brigade staffs. The simulation cannot be hosted on a single PC, terrain databases cannot be easily generated, and civilian or specially trained military support staffs are required. Although an aggregate combat model holds out the potential for reduced computational requirements, and thus a higher probability of hosting the simulation on a single machine, CBS is not the answer.

The currently fielded simulations were designed to be training tools for unit commanders and staffs. These simulations are all run from fixed facilities. The army continues to use these simulations in this role with great success but they are not suitable for use in a field environment. To develop a simulation for field use it is important to understand how the army trains units in a field environment. The most realistic and demanding training environments in the army today are found at the army's Combat Training Centers (CTC).

D. COMBAT TRAINING CENTERS

The army has three CTCs, The National Training Center (NTC), Joint Readiness Training Center (JRTC) and Combat Maneuver Training Center (CMTC). A CTC is a military installation that provides the most realistic, stressful and intense training environment possible short of live combat. Brigade and battalion size units travel to a CTC to train in that environment for what is typically a one month period. A unit visit to a CTC is termed a rotation and units typically visit a CTC once every 18 to 24 months.

The CTCs provide a wide variety of services to the visiting unit. The three most important are a dedicated, live, free playing opposing force (OPFOR), an instrumented battlefield and a cadre of Observer/Controllers (OCs).

The OPFOR is a resident unit at a CTC that fights against the visiting unit is a series of battles during the rotation. The mission of the OPFOR is to decisively defeat the visiting unit using the weapons and doctrine of an enemy force. For example, in the 1980s, OPFOR equipment and doctrine closely resembled that of the Soviet Union. It is important to understand that the OPFOR is not a harnessed enemy. OPFOR commanders are given a mission within a scenario and the freedom to accomplish that mission as they see fit. The

only constraint is that of OPFOR doctrine. The purpose of the OPFOR is to provide the visiting unit a doctrinally correct representation of an enemy unit. To ensure realism the OPFOR is a free playing, thinking opponent whose sole goal is defeat of the visiting unit.

The instrumented battlefield is critical to successfully capturing the strengths and weaknesses of the visiting unit. For example, the computer systems at the NTC capture the movement and engagements of almost all the combat vehicles participating in an battle. After the battle has ended, it can be replayed on a computer screen and studied in order to discover what occurred and why. The benefit is obvious: the detailed study of both successes and failures allow units to correct deficiencies and sustain strengths. The huge amount of data captured by the instrumentation replaces human perceptions of what happened in a confused battle situation with facts. The replacement of perception with fact is a significant step towards objective analysis.

The final service provided to the visiting unit by a CTC is the cadre of OCs, seasoned officers who observe the visiting unit plan, prepare and execute each mission. At the NTC, for example, a group of approximately 400 OCs fans out across all elements of a 3000-man visiting brigade. The purpose of an OC is to observe the unit as they plan, prepare, and execute a mission and then provide objective feedback to the unit. The army uses the After Action Review (AAR) to provide feedback to the visiting unit. The AAR is an objective look at what happened and why. The OC leads the discussion during an AAR and helps units see their strengths and weaknesses.

The CTC OCs provide an additional service for the army as a whole. After each rotation a summary of observed strengths and weaknesses is compiled and sent to the Center for Army Lessons Learned (CALL). At CALL the observations are catalogued against specific training tasks and then published for army wide use. Thus, at the unit level, brief synopsis of observed training trends are available as a resource. The trends presented in these publications are the latest training data available and represent the wealth of the operational knowledge and experience found in the OC groups.

E. SUMMARY

The difficult task of synchronizing a brigade combat team requires a high level of cognitive thought from staff officers. The tools currently available do not help staff officers

think through synchronization problems nor to visualize complex time-space-unit capability relationships. A tool is needed to reduce mental workload on staff officers so that they can focus on synchronization issues. Employment of a simulation during the war gaming step of the MDMP would be an example of such a tool.

The simulations currently fielded by the army do not measure up to the task at hand. In general, they cannot be hosted on a single PC, they require specially trained staff to operate, the time to build a scenario is far too lengthy, and terrain databases take far too long to generate. The simulations currently fielded by the army were initially fielded in the early to mid 1980s. PC technology at that time could not support complex combat simulations. As a result, the current suite of simulations run on UNIX machines and has outdated methods of developing terrain databases. These simulations are essentially static, require large numbers of outdated computers, and use obsolete graphics rendering hardware.

A new simulation is needed to support real time use in the field. The simulation must be hosted on a single PC. Terrain databases must be easily generated from digital terrain data available from NIMA. It must be possible to develop these databases in a matter of hours, potentially while in route to a new area of operations. Scenario generation must be fast, preferably less than thirty minutes, and the simulation must run in less than three hours. Of equal importance is the simulation's ease of use. The simulation must not require advanced computer skills or special training of any nature. Brigade staffs do not have the time or the personnel to dedicate to operating and maintaining a simulation. In short the simulation must resemble commercial application software found on modern PCs: easy to use and maintenance free.

F. SUMMARY OF CHAPTERS

The rest of this thesis justifies the requirement for a new simulation and presents a prototype for the type of simulation required to support wargaming step of the staff planning process. The remaining chapters of this thesis are organized as follows.

- Chapter II: The Military Decision-Making Process. The MDMP is explored to a moderate degree of depth so that the process a new simulation will support is fully understood.

- Chapter III: STAFFSIM. The software component architecture of the simulation is explained.
- Chapter IV: BattleSim. The software component architecture for the simulation module of STAFFSIM is developed.
- Chapter V: STAFFSIM Implementation. A typical scenario presented to brigades training at the NTC is presented and run using STAFFSIM. The results of the run are analyzed against the requirements for a simulation tool presented in earlier chapters.
- Chapter VI: Conclusions. The utility and limiting factors of the new simulation or discussed. Recommendations for future work are also suggested.

II. THE MILITARY DECISION MAKING PROCESS

A. PURPOSE OF THE MDMP

Tactical decision making is an ongoing process. Even while one battle is being fought, unit staffs are busy planning and preparing for the next. Decisions about ongoing operations must be undertaken concurrently with decisions and planning for future operations. The MDMP provides the framework within which the commander and staff make decisions [3]. Within the MDMP information is collected and logically analyzed enabling the commander and staff to develop the best possible course of action COA to achieve the mission [4].

In order to be timely and effective, a staff's implementation of the MDMP must be flexible, comprehensive, continuous, and focused on the future [2]. Flexibility relates primarily to the time available to complete the process. Staffs must not become rigid; they must be able to smoothly transition to an abbreviated decision making process when the situation warrants. Staffs must ensure all factors affecting the mission are carefully considered. These factors include friendly forces and capabilities, likely enemy forces that will be encountered and the environment. The staff planning process has no real beginning or end. Staff estimates are continuously updated as new information becomes available. In turn, if new information warrants, combat plans and orders are updated as well. Finally, decision making is about arranging activities in time and space such that future events cause the enemy to be defeated. "Statistical record keeping is of little value" [2]. Military decision making is about making decisions that will influence future events, not keeping an accurate log of what has or is happening.

B. METHODOLGY

As mentioned in chapter one, the MDMP process has seven steps: mission receipt, mission analysis, course of action development, course of action analysis, course of action comparison, course of action approval and orders production. This section will briefly describe three of these steps; mission analysis, course of action development, and course of action analysis. This discussion will allow the reader to gain an appreciation for the context

in which it is proposed to use simulations as a real time, operational decision support tool.

1. Mission Analysis

Mission analysis is the framing of the problem at hand, usually a tactical mission. The purpose of mission analysis is to allow the commander and staff to "see the terrain, see the enemy and see themselves within the context of the higher headquarters fight" [6]. It is important to understand that mission analysis is not the study of 'how to' accomplish a mission but is instead a study of what must be accomplished, what resources are available, and what constraints exist. In essence, mission analysis serves to ensure that the problem at hand is fully understood before potential solutions are developed. During mission analysis, the staff gathers facts bearing on the mission, makes planning assumptions where gaps in the available information exist and analyzes the higher commanders mission and intent as given in the operations order [4].

The end state of mission analysis is the mission statement for the unit. The commander participates with the staff in these activities as time permits, but as a minimum he must approve the unit mission statement and then issue planning guidance to the staff [2]. The time available to complete mission analysis at the brigade level generally ranges from one hour and 45 minutes to three hours [6, 18]. These times include the time required for the commander to give planning guidance to the staff.

2. Course of Action Development

Having gained a full appreciation for the problem through mission analysis, the staff must develop potential solutions. The army terms the solution to a tactical problem a course of action. Thus, the next step in the MDMP process is course of action development. A course of action is a "plan open to the commander that would accomplish the mission" [4]. Depending on time and resources available, the staff develops two to three courses of action as a minimum. If time is available the staff should develop several courses of action for each potential enemy course of action [4]. The time available to develop courses of action generally ranges from one to two hours [6, 18].

In its staff manual, FM 101-5, the army defines five qualities of a viable course of action.

> **<u>Suitability</u>. It must accomplish the mission and comply with the commander's guidance.**
>
> **<u>Feasibility</u>. The unit must have the capability to accomplish the mission in terms of available time, space and resources.**
>
> **<u>Acceptability</u>. The tactical or operational advantage gained by executing the COA must justify the cost in resources, especially casualties.**
>
> **<u>Distinguishability</u>. Each COA must differ significantly from any others. Significant differences may result from the use of reserves, different task organizations, day or night operations or a different scheme of maneuver.**
>
> **<u>Completeness</u>. It must be a complete mission statement [2].**

A completed COA that embodies these qualities is not necessarily a detailed and complete plan of operations. Instead, it is a more general outline that will be fleshed out during course of action analysis.

The development of a COA is a six-step process. Development begins with an analysis of force ratios and proceeds through generation of options, arraying forces, development of a scheme of maneuver, assignment of headquarters and ends with the drafting of COA statements and sketches. Good COAs position the force for future operations, allow flexibility to meet unforeseen circumstances and provide the maximum latitude possible for subordinates to exercise initiative [2].

3. Course of Action Analysis

The purpose of course of action analysis is to identify the single COA developed above that accomplishes the mission while minimizing casualties and best positions the force for future operations [2]. COA analysis helps determine how to maximize combat power, protect the force, and minimize collateral damage. During COA analysis the commander and staff develop a shared vision of the battle, determine resources required and how to allocate them, how to focus the intelligence collection effort and identify coordination requirements in order to produce a synchronized brigade plan of operations [2].

The primary tool used by brigade staffs to analyze COAs is the war game. The war game is an attempt to visualize how a battle will develop [2]. It stimulates thought about the COA and provides insights that otherwise might not be understood. The process of war gaming fleshes out a generalized COA into a plan of operations. In other words, the details of

the COA are worked out and synchronized. During the war game the strengths and weaknesses of each COA are determined [2].

The central framework used by the staff in the war game is a discussion of the battle in terms of action, reaction and counter-action [2]. For example, if the enemy attacks a friendly unit, that is an action. How the friendly force responds to that attack is a reaction. The enemy's response to the friendly reaction is then a counter-action. Thus, a COA is analyzed by a discussion of action/reaction/counter-action at each anticipated critical point in the battle. The visualization of how actions, reactions and counter-actions will unfold and their interplay on the battlefield is an entirely mental process for each of the involved staff officers.

Like most parts of the MDMP, the war game has rules that govern its conduct and steps that are followed to execute it. It is informative to review these rules because they shed light on how difficult a mental process the war game actually is and they demonstrate the natural pitfalls that must be avoided if the war game is to be successfully completed.

1. Remain objective, do not allow personality nor the sensing of what the commander wants to influence decisions. Officers must avoid defending a COA solely on the grounds that they developed it.
2. Accurately record advantages and disadvantages of each COA as they become apparent.
3. Continually assess feasibility, acceptability and suitability of the COA. If a COA fails any of these tests it must be rejected.
4. Avoid drawing premature conclusions and the gathering of facts to support such conclusions.
5. Avoid comparing one COA with another during the war game. Course of action comparison occurs only after all COAs have been analyzed [2].

Because the war game makes such high cognitive demands these rules are frequently violated, thus damaging the validity of the war game and reducing the quality of the analysis. The first and fourth rules are particularly easy to violate. It is simple human nature to give the boss what one perceives the boss wants. It is just as easy to reach a premature conclusion and then analyze subsequent data in light of that conclusion instead of using that data to reach an objective conclusion.

Remember that during the war game the commander and staff are trying to visualize the complex time-space relationships and unit interactions of a future battle. Within that visualization they are simultaneously attempting to find shortcomings in their own and the enemy's COA while ensuring that they remain completely objective. It is readily apparent that the war game places huge cognitive demands on the officers involved. Given the high mental workload imposed by the war game, how well do unit staffs measure up to the task of war gaming COAs into synchronized plans of battle?

C. COURSE OF ACTION ANALYSIS IN PRACTICE

Perhaps the best source of information on how well units conduct course of action analysis is the cadre of observer/controllers at the army's Combat Training Centers (CTC). The CTC OCs routinely observe unit staffs at the brigade and battalion level conduct the MDMP to include war gaming. The OCs coach unit staffs to improve their execution of the MDMP and document observed shortcomings. OCs observe units from all over the army and from all branches. They see it all. The army has no other group of officers with as much direct experience with war gaming, its benefits and its typical pitfalls.

The documented observations of OCs are collected and published for army wide use by the army's Center for Army Lessons Learned (CALL) at Fort Leavenworth, Kansas. CALL publishes a list of observed training deficiencies on a roughly semi-annual basis. These lists of observed trends provide the best possible insight into how well the army is conducting the staff planning process. Some observations on COA analysis and war gaming in particular, as well as the issue of CALL's Priority Trends in which they appeared are provided below.

> **Units have the most difficulty with war gaming. During a rotation most units improve their performance with the various phases of the MDMP with wargaming being the one exception [19].**
>
> **The war gaming phase of the Military Decision-Making Process (MDMP) is habitually a weakness for the task force staff [19].**
>
> **War gaming is the most difficult step in the Military Decision-Making Process (MDMP) for units to complete successfully. Units have continued to struggle with this training issue for the past 10 years [19].**
>
> **Wargaming is not universally understood and conducted by staffs to the degree and level necessary to ensure success [25].**
>
> **The greatest shortfall in the planning process is the inability to synchronize the task force because of inadequate wargaming [24].**

Units continue to experience problems during execution that can be traced back to flawed wargaming during the planning process [23].

While somewhat general in nature these quotes bring to light two important facts. First, the war game is a significant problem for most unit staffs. Staffs are not conducting the war game to standard, so course of action analysis suffers as a result. If course of action analysis is faulty, decisions based on that analysis are then potentially compromised. Second, the difficulty with war gaming is not a recent phenomenon but has hamstrung unit staffs for at least a decade. This problem is not isolated to a single staff or to staffs from a particular region, but is prevalent throughout the force.

The CTC trends also speak directly to the issue of synchronizing the course of action during the war game.

Wargaming at the task force level rarely results in a synchronized plan at the conclusion of the wargaming process [20].

The selected COA is never wargamed sufficiently to achieve effective synchronization [21, 22].

Products derived from the wargame are rarely useable, doing little to synchronize the plan or to key the commander to critical tactical decisions during mission execution [21].

If the war game is not producing a synchronized plan it is failing to achieve its purpose. Further study of the CTC trends sheds some light on why the war game is failing.

The task force XO does not facilitate the process (wargaming), and the battle staff loses its focus on the critical events that need to be wargamed and the relationship between events and the decisive point [19].

Wargaming is not focused and does not synchronize the task force plan [22].

The wargame ends up taking all day or night with only the most aggressive participants providing input and the rest of the staff writing their annex without fully synchronizing their BOS (Battlefield Operating System) [22].

Usually, the S-2 and S-3 fight it out at the map board while the remainder of the staff observes in silence [23].

Task Force staff's wargaming either gets too detailed and never finished, or is extremely superficial [21].

These quotes provide some insight as to why staffs have problems with the war game. Failure to focus on actual analysis during the war game could be due to many factors. One of these is almost certainly the heavy cognitive demand the war game makes upon the participants. It is very easy for a staff officer to be a passive bystander, one who observes the interplay but is not actually thinking about the plan. A second issue is how personalities can affect the outcome of war gaming. Often, aggressive, dominant personalities tend to force

their opinion on the others. This is fine if the most aggressive officers are also always the best analysts. Unfortunately, this cannot be guaranteed.

It is readily apparent that the war game is a difficult task for many, if not most, unit staffs to effectively accomplish. An effective war game is absolutely vital to synchronizing a combat plan. When a unit enters combat with a flawed plan it cannot achieve its full potential on the battlefield.

D. POTENTIAL ROLE OF SIMULATION IN THE MDMP

The war game is an excellent candidate for introduction of a simulation into a real-time, operational decision making process. A computer simulation can easily generate the visualization of time-space-unit capability relationships that will enable staffs to better synchronize their plans. The training simulations discussed in chapter one do exactly that. These simulations have gained acceptance and are in widespread use as training tools throughout the army. Unfortunately, they do not meet the requirements for operational use, as discussed in chapter one.

If the visualization of the interplay of unit activity in time and space can be presented to staff officers in the war game, then their mental workload can be greatly reduced. The reduction in cognitive effort will allow staff officers to more fully focus on synchronization issues. The simulation should serve the added purpose of keeping the war game focused. Officers will no longer be caught up in trivial details of 'who shot whom' but can instead focus on the bigger picture of how well a COA is synchronized. The next two chapters of this thesis present a prototype simulation that demonstrates how a simulation could be used as an analytic tool during the war game.

III. STAFF SIMULATION

A. INTRODUCTION

In chapter one the importance the army places on synchronization was established. The war game is the tool the army uses to synchronize COAs. In chapter two it was established that the war game generally fails to synchronize a COA. It was proposed that a simulation could greatly enhance the synchronization through a reduction in mental workload imposed on staff officers by the war game. Unfortunately, as discussed in chapter one, none of the simulations currently fielded are suitable for this purpose. Therefore, a simulation that addresses the needs of brigade staffs in a field environment is needed. Chapter one identified the following characteristics for such a simulation.

- The simulation must be run in a period of one to three hours
- The simulation must be easy to use, requiring no special training of any type.
- The host for the simulation must be a smaller machine, such as a PC.
- The simulation must not require specially trained technical support staff.
- New terrain databases must be quickly and easily built.

From these requirements two additional characteristics can be inferred. For the simulation to be run in less than three hours, scenario initialization, the building of units and assigning them orders, must be simple and fast. Building and initializing scenarios should be simple enough that it can be done during COA development and not detract from time allocated to the war game. Preferably, the time required to initialize a scenario should be less than thirty minutes.

Requiring no technical support staff has deeper implications. For a simulation to be useful, it must evolve with conditions on the battlefield. As new weapons and organizations are deployed, the simulation must have the ability to swiftly incorporate these new entities. A simulation must therefore be capable of being upgraded by non-technical users. This is not to say the simulation must give users the ability to easily author upgrades, but rather, upgrades should be constructed in such a manner that users can install them.

Chapter two reviewed the context in which a new simulation could be employed as a real time decision support tool. Here too, there can be found an implied requirement for the

new simulation. The methodology of the war game calls for the battle to be explored in terms of action/reaction/counter-action at critical points. As the war game proceeds the staff needs the ability to modify the plan on the fly in order to more fully explore the COA. Thus the simulation must be fully interactive, allowing the staff to move forward and backward in time and quickly analyze several variants of the COA at each critical point.

The requirements enumerated above provide a gross specification for the required simulation. This thesis presents a prototype simulation named Staff Simulation (STAFFSIM) that aims to meet these requirements and thus be a useful decision support tool for brigade staffs. STAFFSIM meets many of the requirements by the adoption of a software component architecture. The remainder of this chapter will discuss how STAFFSIM is constructed with software components and the advantages of doing so. The first step is to understand the advantages of programming with reusable software components.

B. SOFTWARE COMPONENTS

What exactly is a software component? Intuitively a component is something that is one part of a greater whole. Unfortunately, a more precise definition is needed if the concept is to be completely understood. One such definition is provided below.

> **A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [26].**

This definition implies two fundamental characteristics of software components.

First, independent deployment implies that a component is in fact a stand-alone entity. In this context stand-alone means that a components internal implementation is independent of other components. The only external dependencies the component needs to function properly are defined in the component's interface. This allows a component to be used by a wide variety of different systems. In order for a system to be built using components of this nature only the requirements specified in the interface must be met. If a component depends upon another component in any other fashion it is not capable of independent deployment. Furthermore, because a component is a unit, it is deployed as a whole; it can not be split or partially deployed. Just as one of the components of a stereo system cannot be cut in half and then used, so too with software components; it is an all or nothing proposition.

Components are meant for third party composition [26]. Because a component is deployed as a single unit, it is fully encapsulated. Thus third parties cannot access a component's internal implementation. Therefore, for a component to be composable by third parties it must have a detailed user interface. The interface syntactically defines what the component provides and what it requires.

The use of components has several distinct advantages over object oriented software, the most obvious being software reuse [26]. In a perfect world the army would have a library of software components that implement approved combat models. Simulation designers could then use these components off the shelf again and again. With reuse comes refinement and ultimately software developers could expect off the shelf components to achieve superior quality. Furthermore, the army could make such a component library open-source, or in other words, make the source code for each component freely available to developers. If software components were open-source they could benefit from the intellectual insights and experiences of a much broader base of developers. Thus combining component architecture with open source code offers the opportunity for superior quality software that is easily reusable.

To further understand the benefits of software components to STAFFSIM, it is necessary to understand the different parts of a high-resolution combat simulation. First, these simulations rely on several databases: one for terrain, one for weapon to vehicle hit and kill probabilities, another for weather effects and so forth. They also include sets of algorithms to handle movement, sensing, detection, and engagements. On top of these functionalities there is usually a visualization of the simulation, such as a map display with unit icons. The simulation may also include some type of graphical user interface (GUI) for interactive play. In currently fielded simulations all of these are inseparable and are thus "stove-pipe" solutions.

The functionalities described above could easily be thought of as individual components. Thus a simulation could be composed of components such as a terrain model, a weather model, a GUI interface, and the simulation itself which encapsulates all the required algorithms. When built in this manner the simulation inherits all the advantages of component design. Furthermore, as more components are written a user could pick and chose from among several components that provide the same functionality. Thus, users could easily

tailor the simulation to their purpose. STAFFSIM aims to provide this kind of composability in order to meet many of the requirements specified above. Keeping the advantages of component design in mind, we will now discuss the component architecture of STAFFSIM.

C. STAFFSIM COMPONENTS

1. The Components

STAFFSIM is composed of seven independent software components: BattleSim, Flora, MessageCenter, SimBuilder, OverlayMaker, ExecutiveOfficer and Draftsman. These components and their interactions are depicted in figure 3.1. Each of these meets the definition of a component given above. In STAFFSIM's case independence means that each of these components executes its function completely without dependence upon, or knowledge of, the other components.

Of STAFFSIM's seven components four were developed as part of this thesis, two were used off the shelf and the last exists in concept only. The two components imported off the shelf are Flora and MessageCenter. These components are used as is and are integrated into STAFFSIM using only their defined user interface. The unimplemented component is Draftsman. A brief description of each component and its purpose follows.

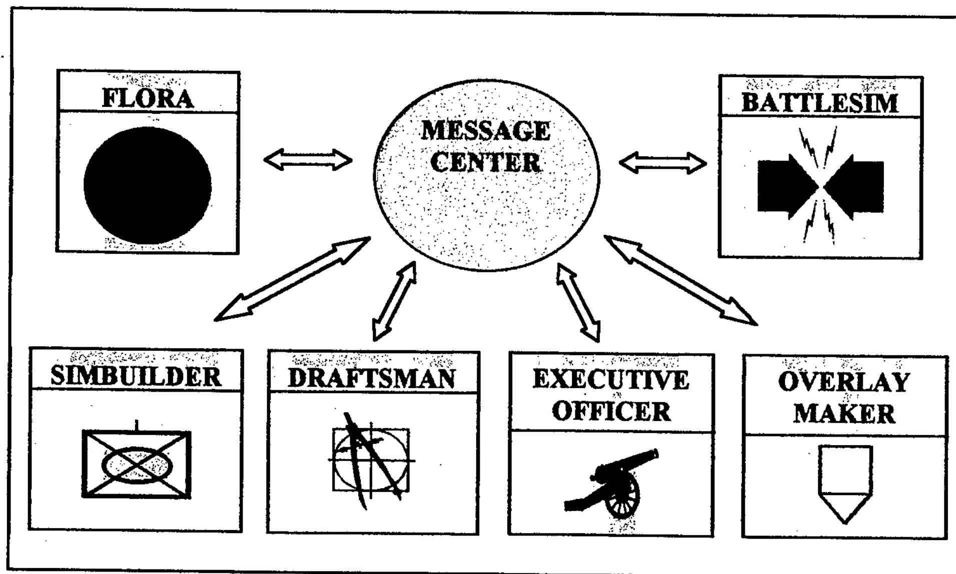


Figure 3.1: STAFFSIM Components

The core component of STAFFSIM is the simulation, BattleSim. Its function is to provide the combat models that will reduce the cognitive workload imposed by the war game. BattleSim provides no other services. BattleSim itself provides no visualization of the simulation or any kind of direct user interface. These services are separate functions that have nothing to do with simulating a combat action, and are best provided by separate components.

The next component is Flora. Flora is perhaps the best demonstration of the software component concept. Introduced by Norbert Schrepf [28], Flora is a simple map display tool that is used to visualize the simulation. In addition to displaying maps, Flora can accurately position unit icons on a map. To do so Flora specifies a message interface. If Flora receives a properly formatted message, it can take the information in that message and represent it on the map.

Flora is a good demonstration of the power of a component architecture. Flora does not know of and does not depend on any other components. Flora was added to STAFFSIM without modification. Thus Flora is a perfect example of software reusability, one of the advantages of software components discussed above. Figure 3.2 presents a screen shot of Flora displaying a 1:500,000-scale map.

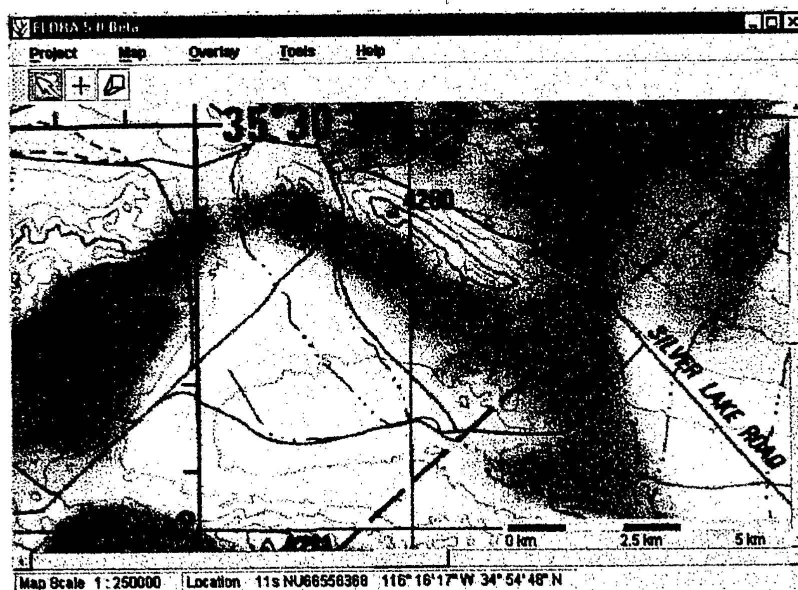


Figure 3.2: Flora

SimBuilder provides a user interface for unit construction to the simulation. SimBuilder allows the user to populate the simulation with the appropriate mix of units. Like

the rest of the components, SimBuilder is stand-alone. The user specifies the units to build and SimBuilder sends the appropriate messages. SimBuilder and BattleSim are mutually exclusive in purpose; they do not depend on each other's existence in any way. The SimBuilder user interface is depicted in Figure 3.3.

ExecutiveOfficer provides the user an interface for commanding units. This allows the user to reach into the simulation and give units orders. On user demand, ExecutiveOfficer creates orders that are then passed to the simulation where they are executed. ExecutiveOfficer can be run stand-alone or it can work with Flora to provide a more intuitive point and click interface. Figure 3.4 shows ExecutiveOfficer configured to build movement orders for a unit.

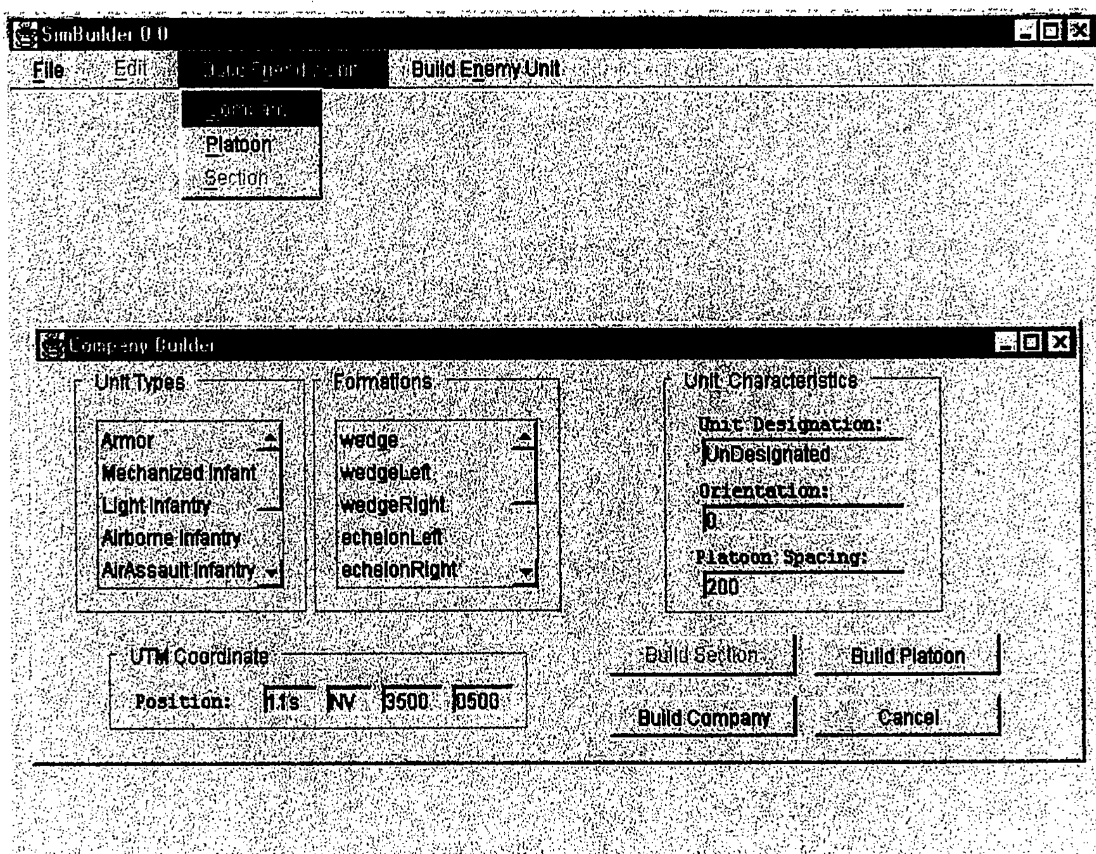


Figure 3.3: SimBuilder Displaying the CompanyBuilder Panel

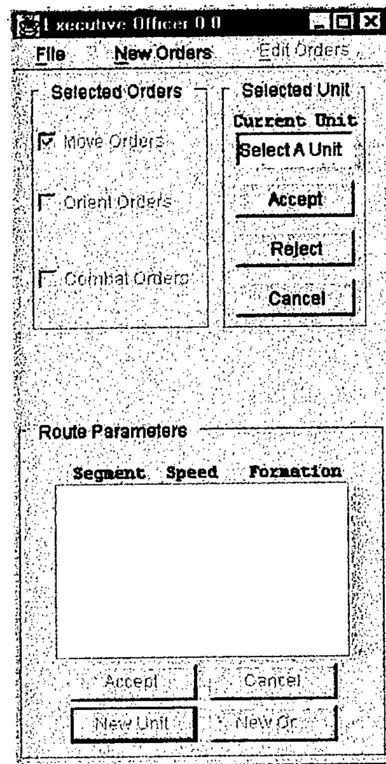


Figure 3.4: ExecutiveOfficer Configured to Build Movement Orders

Draftsman is a tool for drawing military graphics. It interacts with Flora in the same manner as ExecutiveOfficer and may be operated independently of the other components. The last component is OverlayMaker, a by-product of Flora. Flora does not have a well-defined input/output interface; instead Flora can display messages which are objects of a specified type. Because Flora is used off the shelf a local adapter is required; Overlay Maker is that adapter. OverlayMaker receives messages from other components such as BattleSim and ExecutiveOfficer. If those messages contain information that should be depicted on the map display, OverlayMaker builds the correct message objects and forwards them to Flora. This arrangement implies that OverlayMaker have some knowledge of the internal implementation of Flora.

The discussion of STAFFSIM components thus far has made the claim that each component is independent and this is indeed true. In fact, each component can be compiled and run without the others present. To be effectively composed into an application, however, some degree of knowledge about, and communication with the other components is required.

Just like a combat brigade, STAFFSIM is greater than the sum of its parts. The required communication is achieved through two distinct mechanisms, the MessageCenter and component interfaces.

2. Component Communication

The real utility of software components is the ability to combine them into a system. In order for a component to be part of such a system it must be able to communicate with the other components in the system. Simple communication however, is not good enough. A component must be able to communicate without losing its ability to stand-alone or to be composed into other, completely different, systems. STAFFSIM components meet this requirement by structuring their communications with component interfaces and by passing messages through the MessageCenter.

The MessageCenter[28] resembles a multicast IP address found in computer networks. System components send all their messages to the MessageCenter. When the MessageCenter receives a message it simply re-broadcasts it to all registered listeners. The listeners then act on the message if appropriate, or ignore it otherwise.

The MessageCenter de-couples software components. The MessageCenter relieves each component from the requirement of holding a reference to all other components. Thus each component is completely unaware of the existence of other components. The MessageCenter itself is not necessarily aware of all the components either. A component that only sends messages does not register with the MessageCenter and thus the MessageCenter is unaware of its presence. Even registered components are "known" in a very generic sense. The MessageCenter only knows that registered components have a method named *handleNewMessage(ModEvent event)* to which it forwards all messages it receives.

This architecture implies two modes of component operation. Components wishing to receive message traffic simply register their existence with the MessageCenter. They can then send and receive message traffic. Components that do no wish to receive message traffic simply do not register. If a component does not register it does not receive messages. However, it can still send messages, since registration is not required to be a message sender.

The format of messages in this system is extremely simple. A message is a Java object with only two fields. The first is a reference to the message originator or source, of

type Object. The second field is the message itself, also a Java object. Since all Java objects are a subclass of class Object, any object can serve as a message or a message sender. The message can be as simple as the String 'HELLO' or as complex as a Hash Table filled with Vectors of combat units. The flexibility of this arrangement allows component developers much freedom in structuring their component interfaces.

The second piece of the communication infrastructure is the component interface. Schrepf's MessageCenter allows components to communicate with great ease. However, the real question is whether one component can interpret the meaning of another's message. It is easy to be abstract but at some point the details of syntax must be defined. Message syntax is defined in the component interface. A component interface is a collection of Java interfaces. Each of the Java interfaces in the component interface defines one message format. For example, BattleSims component interface might contain separate Java interfaces defining message formats for new units, movement orders or orient orders. Figure 3.5 illustrates this concept.

The software component in figure 3.5 has a single entry point for messages. When a message is received it is examined to determine if it meets the requirements for any of the message interfaces. In this example, the message is first examined to determine if it meets the requirements of the new unit interface. If the requirements for a new unit are met, the message is read and the appropriate new unit is created. If the message is not a new unit message it is examined against the orient order and move order interfaces in turn. If the message does not implement any of the interfaces it is discarded.

If a component wishes to send a message that another component can understand it simply instantiates a message object that implements one of the message receivers' message interfaces. This arrangement promotes a great degree of flexibility. Programmers can build message objects that suit their specific needs. The only requirement is for the message interface to be implemented by the message object. Thus two programmers can encapsulate the same message information in two entirely different message objects. On the receiving side both of these messages will be understood in the same way.

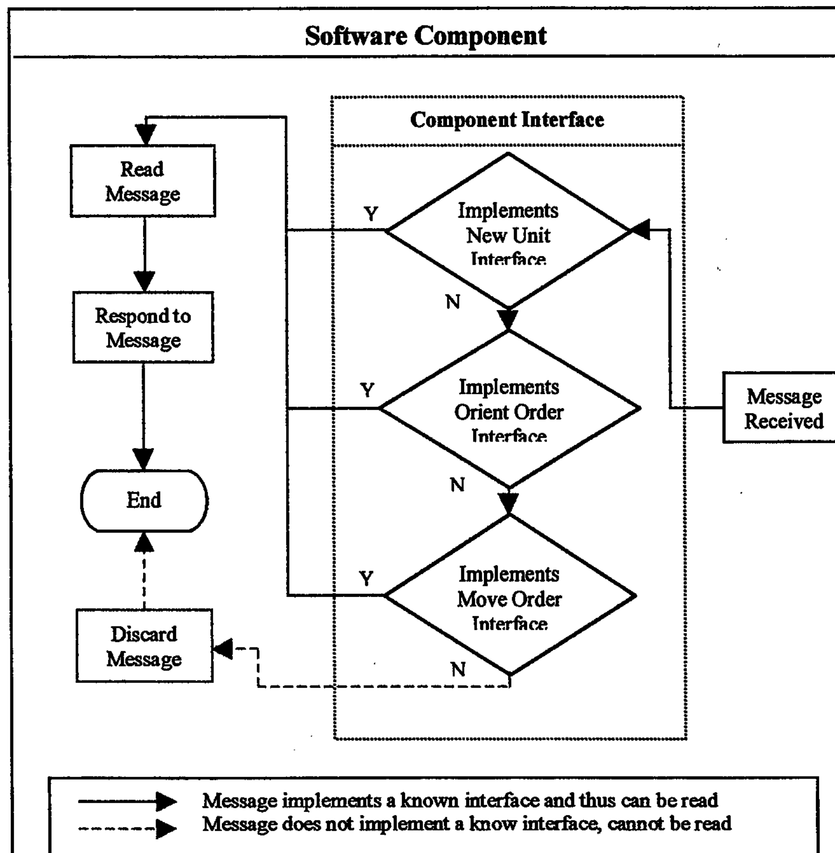


Figure 3.5: Component Interface Flow Chart Diagram

Figure 3.6 illustrates this concept. Components A and B both desire to send component C a message instructing C to change the state of a database. In order for C to understand the message both A and B must compose their message as an object that implements component C's change database interface. Observe however, that A and B instantiate the required message objects but that these objects are not the same. In fact they are of completely different types. Each one is suited to its own needs but can still be interpreted by C.

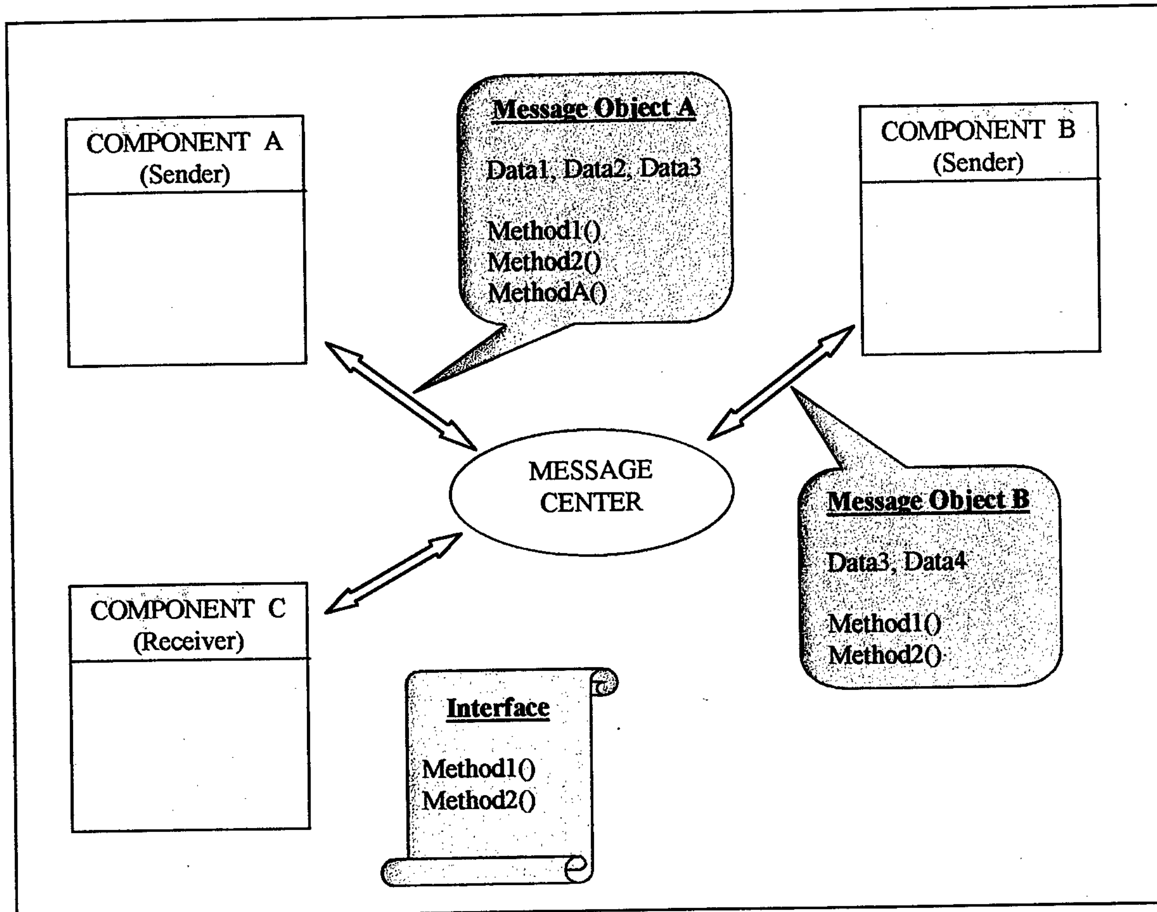


Figure 3.6: STAFFSIM Component Communication

D. SUMMARY

STAFFSIM is an interactive simulation composed from seven independent software components. Each component is a stand-alone application. When linked together through the MessageCenter the components form a complete simulation even though each individual component knows very little about its peers. Composition of the simulation from reusable components gives the simulation developer the ability to pick and choose from the highest quality components when building the simulation. It also allows the simulation to be quickly upgraded. The only real requirements for a new or even completely rewritten component to operate as part of the simulation are the component interfaces. Hence the system can be quickly upgraded with improved components that are easily added to the simulation by the user. Thus, the component architecture of STAFFSIM supports the aforementioned

requirements for a new simulation. The core component of STAFFSIM is BattleSim. BattleSim itself is constructed from software components and is discussed in the next chapter.

IV. BATTLE SIMULATION

A. INTRODUCTION

BattleSim provides the STAFFSIM package with a Discrete Event Simulation (DES) of combat between military vehicles. Like STAFFSIM, BattleSim utilizes a component model. However, the context of the components is significantly different. Each STAFFSIM component is designed to provide a single, basic functionality to the simulation. A BattleSim component however, is designed to provide a single functionality to an entity within the simulation. For example, in STAFFSIM, SimBuilder provides the simulation the functionality of building units. In BattleSim, a component such as *BasicMover* provides an entity in the simulation the ability to move.

BattleSim adopts the component model introduced by Arent Arntzen in his thesis, "Software Components for Air Defense Planning"[27]. Arntzen's concept calls for a component to provide an entity with basic services to facilitate the easy composition of components within a container. For example, to model a vehicle such as a tank, a group of components is added to a container. Each component provides a separate functionality such as moving, sensing, or shooting. When components are added to a container in Arntzen's system, the container takes on all the properties of the added components. Thus, the container becomes a tank. Figure 4.1 depicts this arrangement.

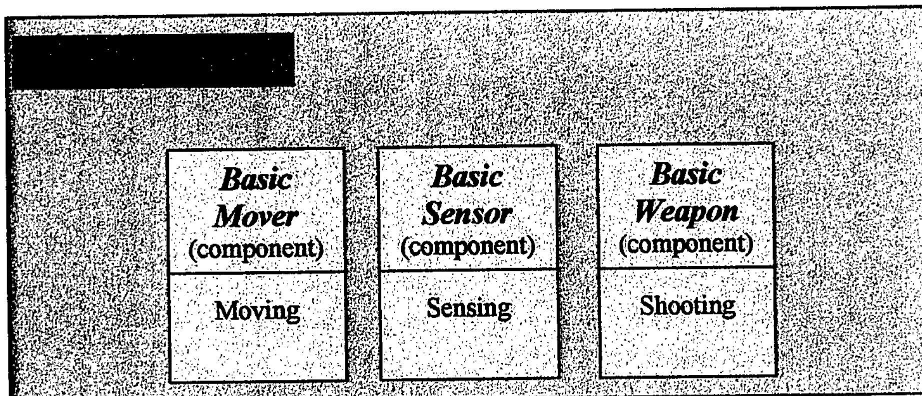


Figure 4.1: Component composition Within a Container

In the tank example, the only component that has a physical location and that can move is the *BasicMover*. Because the container, or tank, takes on the properties of all of its

components, the tank has a location and can move. Thus, the tank delegates its movement properties to it's *BasicMover* component. When a sensor is added to the container the sensor gets its location from the container, thus from the *BasicMover*. Furthermore, when the *BasicMover* moves, the container and all of its components move as well.

It is important to understand that within this arrangement components do not provide overlapping functionality. The *BasicMover* can move but can not sense and shoot. The *BasicSensor* can sense but not move and shoot. The *Vehicle*, however, can move, sense, and shoot.

The functionality that allows composition through containers is embedded within Arntzen's *BasicModComponent*. The functionality of moving, sensing and shooting is included in *BasicMover*, *BasicSensor* and *BasicWeapon*. These components build on *BasicModComponent* and are the basic building blocks BattleSim. These components are discussed in detail below.

In addition to components that are used to build entities such as military vehicles BattleSim has a second fundamental type of component. These components broker the interactions between opposing entities and between entities and the environment. The use of 'broker' or neutral entities has the primary advantage of ensuring opposing entities obtain only as much information about each other as their sensor capabilities and the environment permit.

In BattleSim the 'broker' entities are the *Registrar* and the *Mediator*. These components handle such tasks as determining the outcome of engagements, deciding who can see whom, determining line of sight and so on. The general division of components and their functionality is shown in figure 4.2. The following section describes the basic components in detail and explains their interactions with the 'broker' components.

Broker Components	Player Components
• Registrar	• BasicMover
• Mediator	• BasicSensor
	• BasicWeapon
	• FireControl

Figure 4.2: BattleSim Component Types

B. BUILDING BLOCK COMPONENT MODELS

There are four interfaces that define the component framework of BattleSim: *Mover*, *Sensor*, *Weapon*, and *FireDirection*. Each of these interfaces has a default implementation, *BasicMover*, *BasicSensor*, *BasicWeapon* and *FireControl* respectively. The following sections discuss these components and the combat models they implement.

1. Mover and BasicMover

The *Mover* interface specifies the baseline functionality required for a component to provide the position and movement functions within BattleSim. The actual functionality is provided in the *BasicMover* component. BasicMover extends *BasicModComponent* and thus is composable by container. In order to provide the functionality specified in the *Mover* interface it is implied that *BasicMover* must model movement in some manner. *BasicMover* models movement in a smooth linear fashion called a smooth linear mover [29].

The event graph for the smooth linear mover is shown in Figure 4.3. When a vehicle desires to move it schedules a *StartMove* event. When the *StartMove* event takes place an *EndMove* event is scheduled at a time in the future equal to the time required to complete the move.

The smooth linear model is depicted graphically in Figure 4.4. The smooth linear mover is simple. When the mover begins to move it instantaneously accelerates to cruising velocity. During the move it maintains a constant cruising velocity. When the end point of the move is reached it instantaneously decelerates to zero velocity. In effect, the smooth linear mover does not model acceleration.

Although the movement model is fairly general it will certainly not suit all needs. Changing the movement model is a relatively simple task. To change the movement model developers must simply sub-class *BasicMover* or re-implement the *Mover* interface. As stated above, *BasicMover* provides all the functionality required to operate as a *Mover* within BattleSim. When the subclass is written it must overwrite the methods listed below.

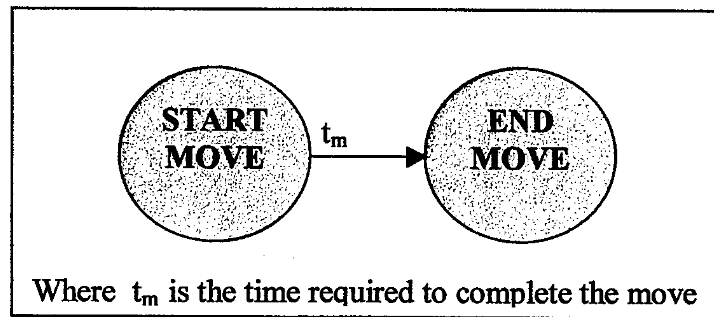


Figure 4.3: Event Graph Snippet for Movement Event Scheduling

- *calcMoveTime()* Calculates the time require to complete the move
- *getCurrentPos()* Calculates and returns the vehicles current position
- *calcMoveDistance()* Calculates and returns the distance to be moved

The code within these functions implements the algorithms for the movement model. Overwriting these functions in the sub-class allows the introduction a new movement model.

An example of a different movement model is one that provides for constant linear acceleration. Although BattleSim does not currently implement a constant linear accelerator the concept is depicted graphically alongside the smooth linear mover in Figure 4.4.

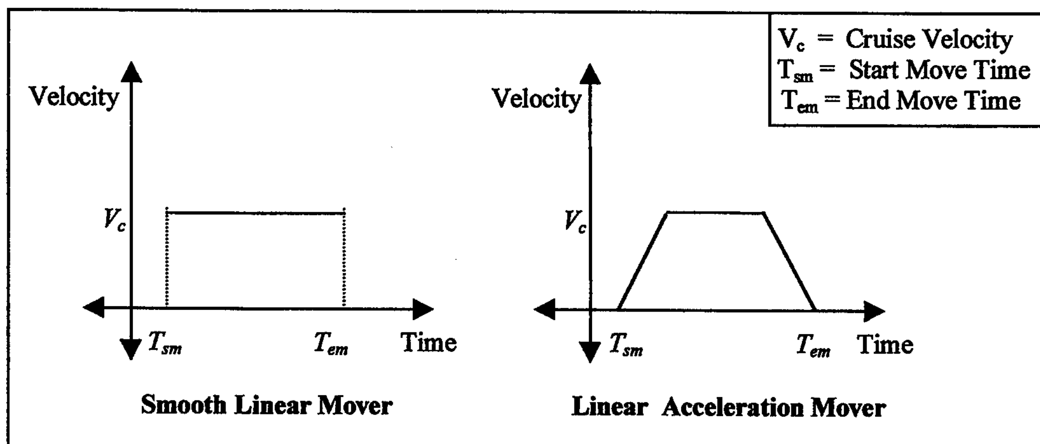


Figure 4.4 Smooth Linear and Linear Acceleration Movement Models

2. Sensor and BasicSensor

The sensing and detecting models in BattleSim are more complicated than the movement model. Before the sensing model is explored in detail it is important to understand the role of some of the other components in the system. Thus far we have discussed the

construction of a vehicle by adding various components to a container. The vehicle constructed in this manner has no information about any other vehicles in the simulation other than that provided by any sensors on the vehicle. The vehicle does not know where other vehicles are until the sensor detects them, but the sensor cannot detect them because it does not know where they are. The simulation is thus in a proverbial 'catch 22' situation.

This problem is solved with the introduction of the *Registrar*. The *Registrar* is a singleton component (i.e. each instance of BattleSim has only one registrar). The purpose of the registrar is two-fold. First, the registrar monitors all the vehicles in the simulation and begins the detection sequence when one vehicle can potentially detect another. The detection sequence determines when vehicles detect each other's presence based on the environment and the capabilities of the each vehicle's sensors.

The second function of the *Registrar* is to instantiate a *Mediator* to handle the resolution of the detection sequence. Once the detection sequence has begun the *Registrar* has completed its task. One *Mediator* is instantiated for each detection sequence that occurs. Once instantiated, the *Mediator* handles all interactions between two vehicles. The *Mediator,* however, is a one way component. The *Mediator* handles a detection sequence for a vehicle pairing where one vehicle is the detecting vehicle and the other is the detected vehicle. A second *Mediator* handles interactions in the opposite direction. This is a different detection sequence and is handled by a different *Mediator*. Figure 4.5 illustrates this concept.

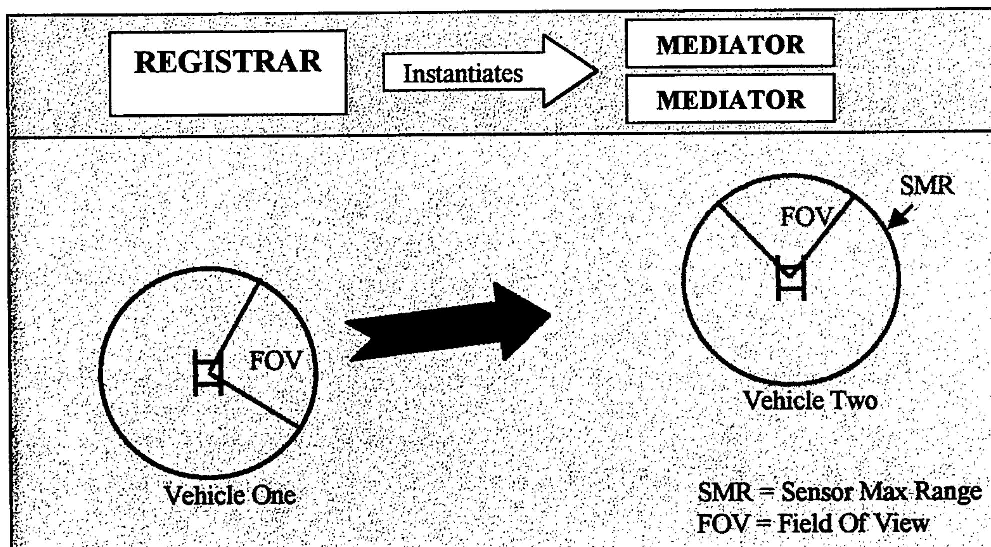


Figure 4.5: Interplay of Vehicle Sensors, the Registrar and the Mediators

In Figure 4.5 the action begins when vehicle one publishes a *StartMove* event. The *Registrar* listens for, and hears the *StartMove* from vehicle one and makes a series of decisions. First, the *Registrar* determines if vehicle two will enter the maximum range circle of vehicle one's sensor. If vehicle two will enter the maximum range then the *Registrar* determines when and schedules an *EnterRange* event for that time. The *EnterRange* event is the beginning of the detection sequence. The *Registrar* will also instantiate a mediator to handle the rest of this detection sequence. For this newly instantiated mediator, vehicle one is the detecting vehicle and vehicle two is the detected vehicle. The *Registrar* will also determine if vehicle one will enter the maximum range of vehicle two during its move. If so, a second mediator is established. For this mediator the detecting vehicle would be vehicle two while the detected vehicle would be vehicle one.

The detection sequence mentioned above begins when a vehicle publishes a *StartMove* event. The *StartMove* event may or may not cause the moving vehicle to enter into the sensor range of another vehicle. If the moving vehicle will enter the sensor range of another vehicle, an *EnterRange* event is scheduled to occur at the time of entry. An *ExitRange* event may be scheduled as well. The *ExitRange* event is not scheduled in cases where the moving vehicle stops within the sensor range of the detecting vehicle.

Once one vehicle has entered the sensor range of another, the *Mediator* takes over. The publishing of an *EnterRange* event causes the *Mediator* to determine if the moving vehicle will enter the field of view (FOV) of the sensing vehicle. If it does, then *EnterFOV* and potentially *ExitFOV* events are scheduled. When the *EnterFOV* event takes place the *Mediator* checks for entry into the sensing vehicles line of sight (LOS). If the target vehicle will enter the sensing vehicles LOS then *EnterLOS* and potentially *ExitLOS* events are scheduled as well. Once one vehicle has entered another's LOS it is time to calculate when detection will take place.

The mediator determines time to detection based on the detection algorithms resident in the detecting sensor. Detection in BattleSim means that one vehicle has seen another but cannot necessarily see it well enough to determine what or who, it is. When a *Detection* event takes place the *Mediator* schedules a *Classify* event. *Classify* means that the detecting vehicle can determine what type of vehicle it is observing in terms of tracked vehicle or wheeled vehicle or fixed position. Classification is an intermediate step on the road to being able to

fully identify what has been observed. When the *Classify* event occurs the *Mediator* schedules an *Identify* event. The *Identify* event represents full knowledge of the detected vehicle to include status as friend or foe and vehicle nomenclature such as T-80 or HMMWV. The detecting sensor once again provides the times to classify and identify. The algorithms to compute these times are similar to those for the time to detection.

Figure 4.6 depicts the event graph for the detection sequence. The event graph shown in the figure is a scaled back representation. Due to the complexity of event scheduling and interrupting a full event graph would be impossible to show on a single sheet of paper. The graph shown in figure 4.6 allows the reader to grasp the basic flow of event scheduling without becoming inextricably mired in detail.

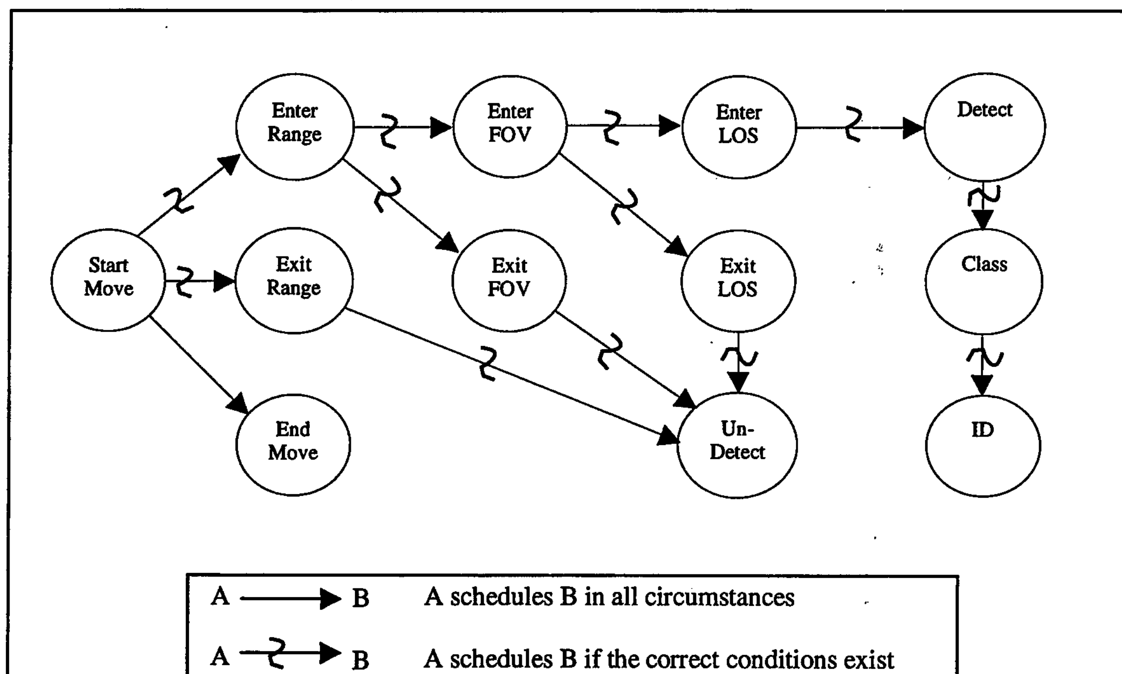


Figure 4.6: Event Graph of the Detection Sequence

Figure 4.7 shows an example of what the detection sequence means to entities in the simulation. The circle in figure 4.7 represents the maximum range of a vehicles sensor. The white pie slice is the sensor's field of view. In BattleSim this field of view is not necessarily the sensor's physical field of view but is usually a sensor's assigned sector of search. The gray areas within the field of view are dead space, areas the sensor cannot see into due to an obstruction of some type.

The action starts when vehicle A starts to move. Vehicle A is the detected or target vehicle while vehicle B is the sensing vehicle. When vehicle A enters the sensor range of vehicle B an *EnterRange* event is published. At this time A is within the sensor range of B but is not within the area that B's sensor is searching. When A enters the search area of B an *EnterFOV* event is published. In this particular example as A enters B's field of view it is also in B's line of sight (LOS), thus an *EnterLOS* event is published at this point as well. Once A has entered B's LOS, detection is possible. Therefore at some point further along

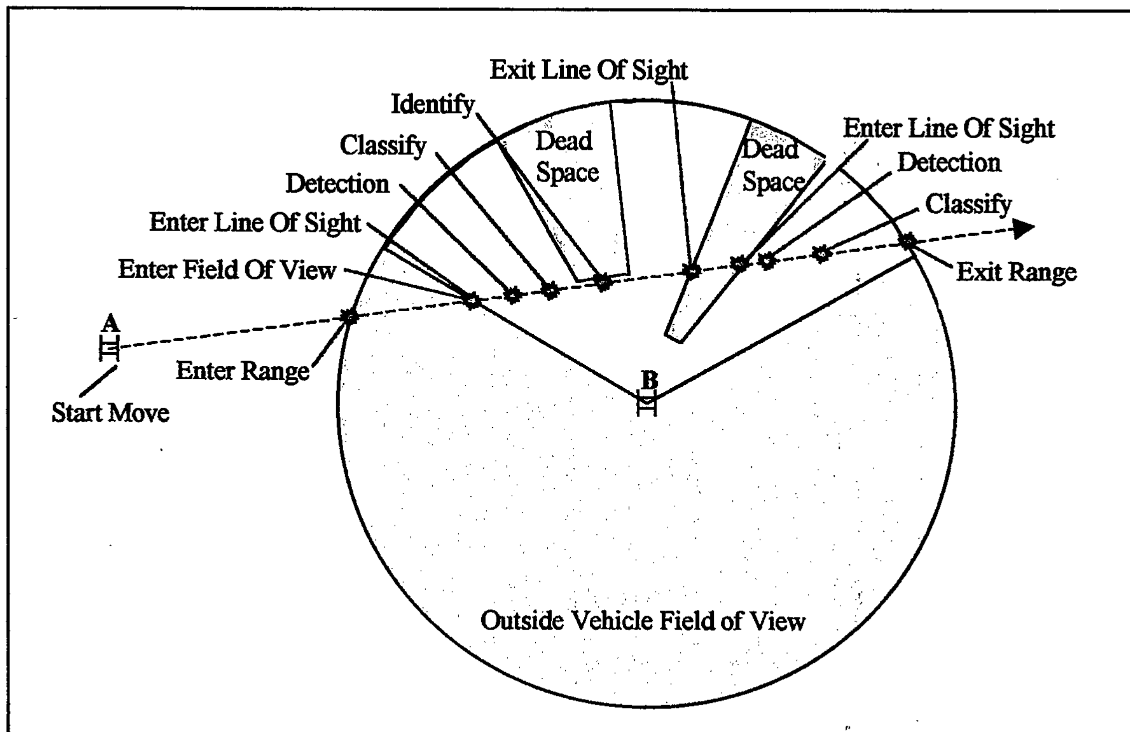


Figure 4.7: Detection Sequence Model

the move path B will detect A. *Classify* and *Identify* events follow. In this example A enters dead space and thus an *ExitLOS* event occurs. When A emerges from the dead space it is once again entering B's LOS and thus the sequence of events repeats itself, circumstances permitting. Finally, A exits the sensor range of B prompting an *ExitRange* event.

Like *BasicMover* the detection model provided in *BasicSensor* may easily be replaced with a more sophisticated one. The model that is easily replaced is the one that actually determines when one vehicle detects another. As with *BasicMover* sub-classing *BasicSensor* guarantees the new model will work within the system. However, when sub-classing *BasicSensor* the following methods must be overwritten.

- *getTimeToDetection(),* Calculates time until sensor detects target vehicle
- *getTimeToClassify(),* Calculates time from detection until target is classified
- *getTimeToIdentify(),* Calculates time from classification until identification
- *getRightLimit(),* Returns sensors right limit azimuth
- *getLeftLimit,* Returns sensors left limit azimuth
- *inFieldOfView(),* Determines if passed location is within the sensors FOV

The code within these methods is the implementation of the detection algorithm.

3. Weapon and BasicWeapon

Of the four basic building blocks *BasicWeapon* is the simplest. The only functionality encompassed in *BasicWeapon* is the ability to shoot. Included within the ability to shoot is the concept of ammunition availability. An integral part of each weapon is the ammunition on hand for the weapon to fire. When the ammunition is expended, the weapon will no longer fire.

Firing a weapon is a much more involved process than simply loading it and pulling the trigger. Combat scenarios usually present decisions such as what target to shoot at or simply deciding whether or not to shoot. To help the soldiers manning the weapons make smart decisions under the stress of battle the army has developed fire control measures. These control measures include trigger lines, sectors of fire and weapon control status. These concepts are implemented in BattleSim but not in *BasicWeapon*. A *BasicWeapon* simply shoots when it is told to do so.

The functionality that *BasicWeapon* does not implement is obviously very important. The decision of when to shoot and at whom to shoot requires a level of intelligence not normally embedded within weapons themselves. The capability to make these decisions is found in the weapons operator or in an automated fire control system. In BattleSim this functionality resides in *FireControl*, which the next section discusses in detail.

4. FireControl

FireControl is the last of the basic building block components. A *FireControl* component is added to a container representing a vehicle in the same manner as other components. The *FireControl* is essentially the vehicle's brain, deciding whom to engage and

when. The *FireControl* links the sensor to the weapon thus creating a weapon system. In order to control fires in a manner resembling a military vehicle *FireControl* implements many of the fire control measures found in military fire planning manuals.

Figure 4.8 shows the event graph for shooting at a target. The *Detect, Classify* and *Identify* events are lifted from the sensing and detection event graph shown in figure 4.6. Shooting is obviously directly linked to sensing. A target cannot be shot until it is detected. Depending upon the weapon control status, a *Detect, Classify* or *Identify* event can trigger a *NewTarget* event. If the weapon control status is restrictive then a *NewTarget* event will not be generated until the target vehicle is positively identified. In a permissive environment a *NewTarget* event is triggered as soon as a target is detected.

A *NewTarget* event ultimately results in the addition of the detected vehicle to the *FireControls* target queue. Once added to the target queue the detected vehicle will be engaged. Once the detected vehicle moves to the top of the queue the *FireControl* orders a weapon to shoot at it. This is represented by a *Fire* event. At this point the detected vehicle is removed from the target queue regardless of the outcome of the engagement. This is done because neither the fire control nor the weapon can assess the results of the engagement. Therefore, from the *FireControls* perspective the target has been handled.

The result of an engagement is received by the shooting platform via its sensor. The *Mediator* determines the result of the engagement and informs the sensor. If the shot was a miss, the sensor notifies the *FireControl* and the detected vehicle is re-added to the target queue. If the shot hit, no further action is required by the *FireControl.*

Figure 4.8 shows the event graph for the engagement sequence. If the weapon control status is weapons free then the *Detect* event causes a *NewTarget* event to be scheduled. At the other end of the spectrum if the weapon control status is weapons hold, then a *NewTarget* event will not be scheduled until the target is positively identified as signified by a *Detect* event in the figure.

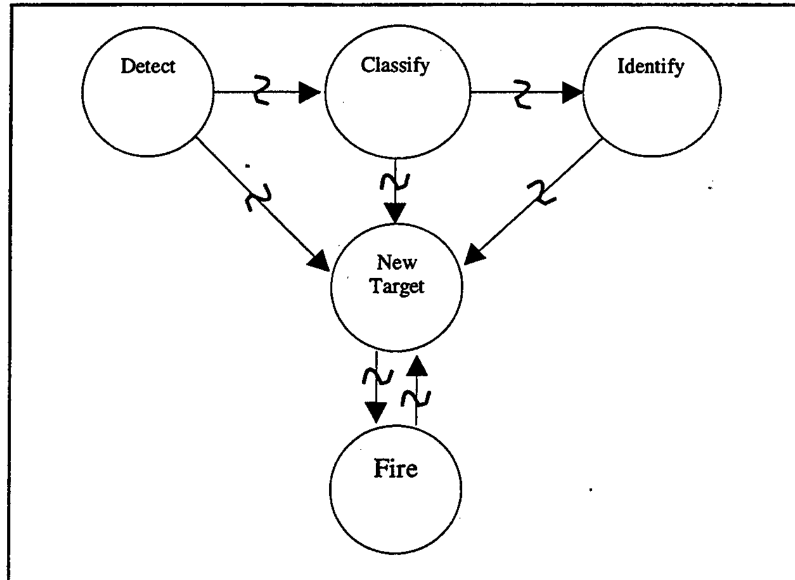


Figure 4.8: Event Graph of the Engagement Sequence

The event graph fails to depict the full detail of the *FireControls* decision making. In order to give the reader a better understanding of the complexity of the decision a flow diagram of the algorithm is provided as Figure 4.9. The decision cycle begins when the *FireControl* receives notification of a new target. If the new target is in sector, the current weapon control status (WCS) for in sector targets is checked. If the WCS allows engagement, the *FireControl* checks to ensure the new target can be ranged and is inside the user set trigger line. If the target is in range and within the trigger line then it is added to the target queue. Once the target is added to the queue, the *FireControl* pops the first target off the queue and orders a weapon to shoot at it. When the firing weapon receives the order to shoot it checks to ensure ammunition is available. If ammunition is available a shot is fired and the *FireControl* pops the next target off the queue. Targets are prioritized in the queue based on the danger they pose to the detecting vehicle. The most dangerous targets are always first in the queue.

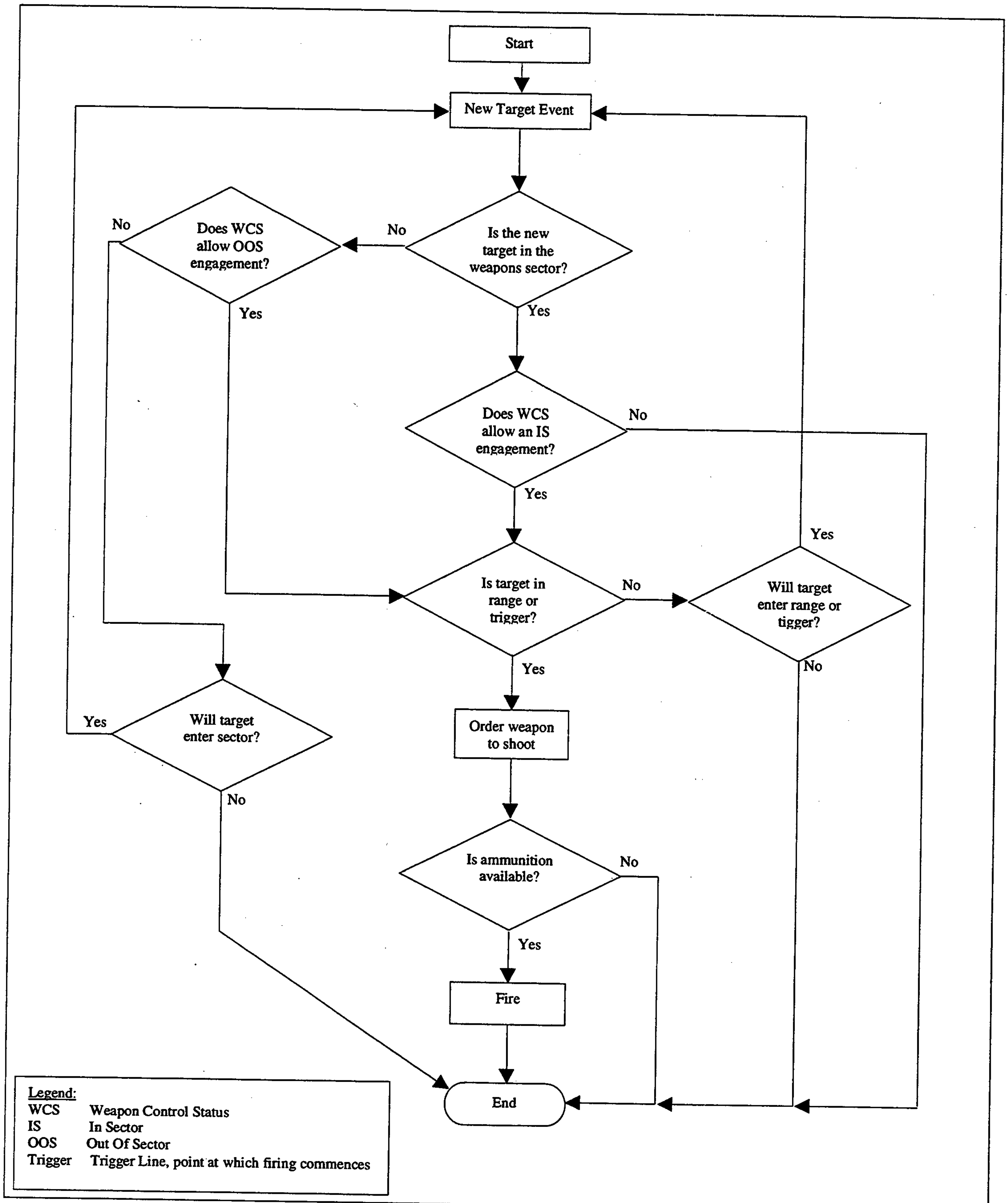


Figure 4.9: FireControl Decision Flow Chart

C. COMPONENT CONTAINERS

Containers are the mechanism by which components are composed into complex entities. STAFFSIM uses two types of containers. The first type allows the composition of components into vehicles using Arntzen's *ModContainer*. The second allows the aggregation of vehicles into units. In this case the container, or unit, should not inherit the properties of its component vehicles and thus *ModContainer* is not used. The following sections describe these two approaches to component containers.

1. Vehicles

The *Vehicle* interface is the primary container in BattleSim. When a component is added to a *Vehicle* container the container takes on all of its properties. A property in this context is any method that meets the following criteria.

- The method name begins with the word 'get'.
- The method has no arguments.
- The method has a non-void return type.

These criteria are the same as those used by Java Beans. For example, suppose a component with the method *public double getMaximumSpeed()* is added to a vehicle named *tank1*. The vehicle now has a property named maximumSpeed. This property is accessed with the following call.

tank1.getProperty("maximumSpeed");

Thus *tank1* now has a property called maximum speed. The functionality to make this happen is all included in Arntzen's *BasicModComponent* and *BasicModContainer* classes. In the example above, the container class must either sub-class *BasicModContainer* or implement the *ModContainer* interface. The component added to the container must extend *BasicModComponent* or implement the *ModComponent* interface.

Arntzen's work allows a container to assume the properties of components that are added to it. Significant benefits could be gained if a component in the container could also assume the properties of all the other components. Arntzen's component system currently does not support this functionality. To illustrate these concepts consider the following situation.

Suppose a *Sensor* is added to a *Vehicle*. The sensor's job is to detect targets and report that information to the vehicle. Part of detecting a target is being able to report where the target is. To do this the sensor must first know its own position. As discussed above, the sensor has no concept of its own position. However, the vehicle does know its position from its *Mover* component. Therefore, the problem is one of access to information that is already available. The *Sensor* cannot get its position from the *Mover* because the *Sensor* does not even know the *Mover* exists.

There is a simple solution to this problem. In BattleSim each component has a parent property. This property is a reference to the container in which the component resides. Thus, for the *Sensor* to get its location it simply queries its parent. Once the *Sensor* knows its location it can accurately report the position of targets it detects.

The benefits of this arrangement are two-fold. First, the amount of code is significantly reduced. Only one component must incorporate a specific property. Other components that need this property can get it from their parent. Thus, there is a single source for each property. The second benefit flows directly from single sourcing of properties. Single sourcing eliminates potential conflicts between components that would otherwise implement the same functionality. For example, what happens if the *Sensor* and the *Mover* both implement a position property? The potential problem arises that although in a physical sense these components are located in the same spot their position properties might not be the same. The natural question is then, who is right? How does the vehicle determine who is right? The introduction of the parent property eliminates this source of potential errors.

The vehicle container takes an additional task upon itself. When a resident component publishes an event, the container intercepts that event and changes the source of the event to be the container. The purpose here is simple. In order to appear to other containers as single entity events originating in the container must be sourced as if they originated from the container, not a resident component. Thus containers intercept their component events and change the source field from the component to the container.

The interception of resident component events has an associated disadvantage. Messages inbound to a component are sent to the components parent container instead. The container must then interpret the message and decide which component it is for. The introduction of the code required doing this limits the reusability of the container.

Consider the following example. A container designed to model a combat vehicle might have resident components that represent sensors, weapons and a mover. While this container could then be used to represent a tank, a self-propelled artillery piece or even a navy ship (depending upon the components) it could not represent a machine tool on a factory floor. The machine tool might have components such as a control unit, spindle, and tool tips. The vehicle container could in fact include these components but could not handle their message traffic because it does not contain the code to route inbound messages to the proper component.

2. Units

The purpose of the *Unit* interface is to allow aggregation of entities. Military organizations typically group men or vehicles into units and then units into larger units and so on. The *Unit* interface models this military hierarchy. Grouping entities into units also allows the user to interface with a single unit as opposed to ten to twenty vehicles that composed that unit.

For example, in Janus for a user to order ten vehicles to move from point A to point B the user must individually order each unit, a tedious and needlessly time-consuming task. A second approach available in Janus is to command just one unit to make the move and have the other nine mirror the movement of the first. This approach is certainly more time efficient but results in the units moving in a manner that poorly models the behavior of military units. BattleSim offers a different approach. The *Unit* interface allows the user to order a *Unit* to move and the container ensures that the vehicles move in a manner consistent with military movement techniques.

The purpose of the *Vehicle* interface is to allow the grouping of components to form a single entity. Obviously, the purpose of the *Unit* interface is significantly different. Because a unit has no need to assume the properties of its component vehicles nor to intercept and re-source message traffic a different aggregation technique is used. The technique is much simpler and requires much less overhead. A unit is simply a collection of vehicles. The functionality provided by *BasicModComponent* and *ModContainer* are not needed and therefore they are not used.

As an example consider BattleSim's *Platoon* class which implements *Unit.* To build a platoon vehicles are added to the platoon object up to a limit of six. The platoon object monitors message traffic from its component vehicles but does not re-source the messages. In BattleSim the basic entity is a single vehicle. Therefore the *Registrar* and the *Mediators* know and understand how to interact with vehicles but not with units. Thus the source for all messages must be a vehicle, the *Registrar* or a *Mediator*. The power provided by aggregating vehicles into units is speed of user interface as discussed above.

D. COMPONENT INTERACTIONS

1. Introduction

A variety of BattleSim components have been introduced and their purpose discussed. At this point is useful to take a step back and review primary players and how they fit into the bigger picture of a BattleSim simulation. Remember that BattleSim simulates the fighting between two brigade sized combat units. Within the simulation the primary entity is a single vehicle. Thus BattleSim models the brigade level fight as the interaction of hundreds of combat vehicles. Figure 4.10 diagrams this concept.

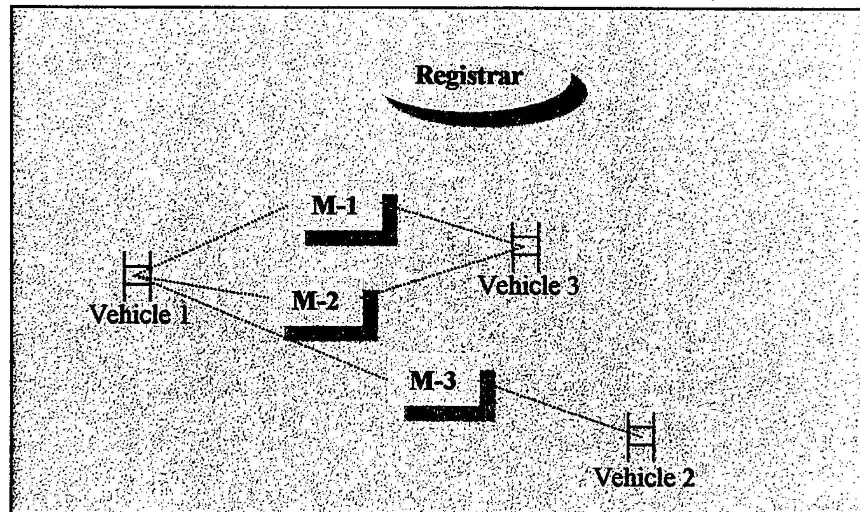


Figure 4.10: BattleSim Component Interactions

Figure 4.10 can be considered to be a snapshot in time of a BattleSim simulation. The basic entities in play are vehicles, mediators and the registrar. The *Registrar* and the *Mediators* are neutral entities while vehicles two and three oppose vehicle one. The diagram

depicts three mediators. M-1 is handling the interactions between vehicles one and three where vehicle one is the sensing vehicle and vehicle three is the target vehicle. M-2 is also handling interactions between vehicles one and three. In this case however the sensing and target roles are reversed. M-3 is handling interactions between vehicles one and two with vehicle one the sensing vehicle and vehicle two the target vehicle. Vehicle one has not entered sensor range of vehicle two and therefore no mediator has been instantiated to handle interactions in the reverse direction.

The interaction of these entities results in a battle. Vehicles move, sense, shoot, kill or are killed. Each of these discrete activities is represented in BattleSim by one or more events. Events are the primary means of inter-entity communication. When an event occurs the source entity notifies other interested entities. For example, if vehicle one in Figure 4.10 fires at vehicle three, vehicle one schedules a *Fire* event. When the *Fire* event takes place vehicle one notifies all other interested entities that it is firing at vehicle three. One of these interested entities is *Mediator* one. When *Mediator* one is notified of the *Fire* event it decides the outcome of the engagement. In this way publishing an event is very much like passing a message. In this case however, each message represents a physical occurrence on the battlefield at a specific time. The event passing mechanisms of BattleSim are discussed in depth below.

2. Event Handling

a. Listeners

BattleSim adopts the listener pattern developed as part of Arntzen's Modkit component framework. Modkit listeners are very similar to Java Bean's listeners. The basic concept is that any entity that wishes to receive events published by another entity simply registers to do so. Each entity keeps a list of registered listeners. When an entity publishes an event it notifies all of its registered listeners. Thus listeners are able to track what an entity is doing and respond to another entities actions.

In BattleSim it is very important to place restrictions on who can listen to whom. Recall that a mediator handles all interactions between two entities. The mediator thus listens to both of the entities. The entities do not listen to the mediator nor are they allowed to listen to each other.

In a physical sense these restrictions are intuitive. It is unrealistic for an entity to receive the events of an enemy entity. Receiving such information implies that an entity knows precisely what an enemy entity is doing under all circumstances and at all times. Essentially this state of affairs would be akin to riding in the enemy vehicle observing all of its activities and listening to all of its communications. Thus, opposing entities are not allowed to register as listeners to each other. Entities can not register as listeners to the registrar or the mediators as well. The registrar and mediator publish information in their events that is meant for one side or the other but not both. For example, if one vehicle enters another's LOS the mediator publishes and *EnterLOS* event. Only the detecting vehicle receives this information. The detected vehicle has no way of knowing when it enters or exits another vehicles LOS. Therefore, entities, or vehicles, cannot register as a listener to the *Registrar* or the *Mediators.*

The *Registrar* has no listening restrictions; it listens to all the message traffic outbound from vehicles. This enables the *Registrar* to initiate the detection sequence as required. The mediators are restricted in who they listen to. *Mediators* listen to the *Registrar* and the components they mediate. There is no real need for mediators to listen to each other or to other vehicles. If the *Mediators* listened to vehicles they do not mediate they would have to filter their inbound message traffic to eliminate messages that do not concern them. This would introduce wasteful inefficiencies in the code.

b. Event Classes

Events in BattleSim are objects. When a listener is notified of an event the listener receives a reference to the event object. Thus the listener has access to all the information in the event. The information passed in events is critical to the simulation. Entities use the information they receive via events to properly respond to the event. For example, a *StartMove* event contains who started moving, how fast they are moving and where they are going. When the *Registrar* receives this event it uses the information to determine if and when the moving vehicle will enter sensor range of other entities.

Figure 4.11 shows the BattleSim event hierarchy. The baseline event is the *BasicModEvent* introduced in Modkit. *BasicModEvent* provides the basic event functionality but very little information. *GenericModEvent* expands upon *BasicModEvent* but provides no information that is not general to all events. The last tier of events provides the specific

information required for entities to properly react to an event. Listed under this tier are the specific events that are passed between entities.

c. Event Scheduling

BattleSim is an event driven simulation; thus each event has a specific time that it will take place. When an event takes place, the simulation clock is advanced to that event's time. Unfortunately, the Modkit component architecture upon which BattleSim is constructed has no notion of time or of a continuously advancing clock. In order to schedule events and have them occur at a future time this shortcoming must be addressed.

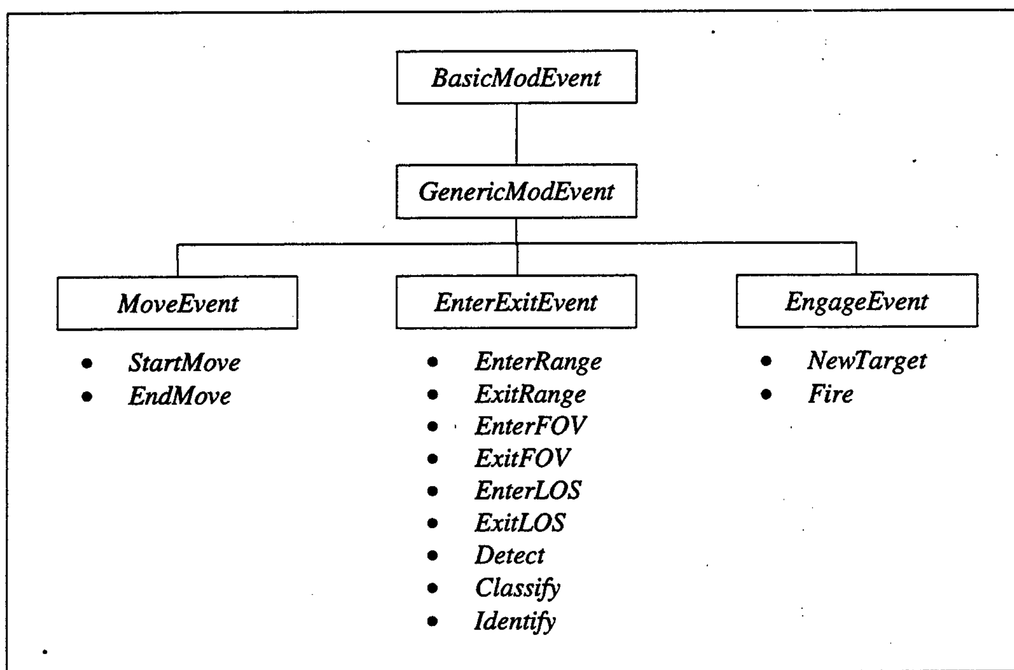


Figure 4.11: BattleSim Event Hierarchy

To handle event scheduling BattleSim uses the event scheduling facilities provided in Simkit [30]. Simkit has built-in event scheduling facilities and a clean interface for scheduling and executing events. The definition of an event in Simkit is provided in the SimEvent interface. Simkit provides a SimEvent abstract factory that creates SimEvents from parameters provided by the user. To schedule an event the user provides the parameters to the abstract factory and receives a SimEvent in return. The SimEvent is then passed to the Simkit Scheduler where it is added to an event queue. When the event takes place the Simkit Scheduler notifies the originating object via a callback. The process of scheduling an event is depicted in Figure 4.12.

The primary problem encountered when scheduling events is that Simkit understands and handles *SimEvents* while BattleSim uses *ModEvents*. The solution to this problem is the *SimkitAdapter* class. The *SimkitAdapter* takes a *ModEvent*, converts it to a *SimEvent*, and sends that *SimEvent* to Simkit for scheduling. When the Simkit scheduler determines that the event has occurred it sends it back to the *SimkitAdapter*. The adapter converts the *SimEvent* back into a *ModEvent* and sends it to the originating component. When the originating component receives the event it understands that the event is taking place now and responds accordingly. Part of the originating component's response is to notify its listeners.

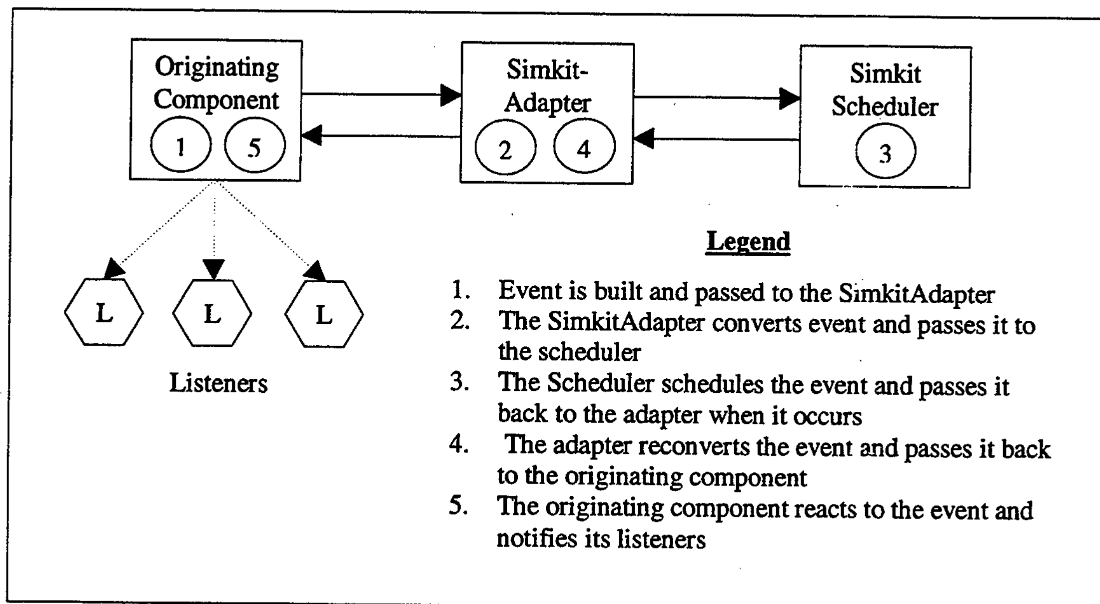


Figure 4.12: BattleSim Event Scheduling

E. SUMMARY

BattleSim provides STAFFSIM a discrete event simulation of vehicle to vehicle battle. The entities in BattleSim are designed with a component architecture in order to maximize component reuse and improve overall efficiency. The components within BattleSim implement simple models for the real world interactions between combat vehicles. These models can be replaced without discarding the component and coding a new component. To introduce a new combat model the existing component is simply sub-classed.

Sub-classing of components in this manner increases the potential for reuse while simultaneously reducing the coding effort required to implement new models.

Recalling the motivations for developing a new simulation discussed in chapters one and two it is now time to evaluate STAFFSIM against those requirements. The primary tool for evaluating STAFFSIM will be its ability to meet the specified requirements while war gaming a typical scenario from the NTC. The following chapter will introduce such a scenario and evaluate STAFFSIMs ability to function as required by army staffs in the field.

V. STAFFSIM IMPLEMENTATION

A. INTRODUCTION

Now that STAFFSIM and BattleSim have been presented it is appropriate to evaluate the simulation in terms of the requirements developed in earlier chapters. The baseline requirements enumerated for STAFFSIM in chapters one through three are listed below.

- The simulation must be hosted on a single personal computer.
- Generation of new terrain models from NIMA products must be fast and easy.
- The time required to generate a new scenario must be less than thirty minutes.
- The simulation must run to completion in less than three hours.
- No specially trained support staff must be required to operate or maintain the simulation.
- Once an upgrade to the simulation is developed and ready for fielding, specially trained support staff must not be required to install it.

These requirements along with how well STAFFSIM supports the war gaming process form the primary yardstick against which to evaluate the concept of simulation support for the war game.

The best measure of STAFFSIM's utility from the operational standpoint is to use it as it was designed. In short, develop a scenario and determine STAFFSIM's ability to assist a staff in achieving better synchronization in the war game. Unfortunately, STAFFSIM does not yet feature the full functionality required to do this. However, the base architecture is complete and does allow trial scenarios to be run and evaluated. The remainder of this chapter discusses one such scenario and STAFFSIM's performance during the trial. The goal of this thesis was to build a simulation that serves as a proof of concept for the operational use of a simulation to support real-time decision-making. This chapter demonstrates a simulation's ability to improve staff synchronization during the war game. Thus, as a proof of concept, STAFFSIM achieves its stated purpose. Additionally, STAFFSIM's ability to meet the requirements reviewed above is discussed.

B. SCENARIO DEVELOPMENT

As discussed earlier the National Training Center (NTC) is the army's premier maneuver training center. At the NTC, brigade staffs face the most challenging series of scenarios possible short of actual combat. To evaluate STAFFSIM, a typical NTC scenario has been replicated. Arguably the most demanding of the NTC scenarios is the full Motorized Rifle Regiment (MRR) attack. In this scenario the rotational brigade is usually allowed 48 hours to prepare a deliberate defense. During the planning and preparation phase the brigade staff conducts the staff planning process to include war gaming. As the brigade completes planning and preparation, the OPFOR attacks in order to penetrate the defense and destroy the defending friendly brigade. During the attack the OPFOR faithfully replicates the doctrine, tactics and equipment of a full MRR.

The first step to war gaming the trail scenario is to initialize STAFFSIM with the opposing orders of battle and courses of action. The time required to initialize a scenario is one of the primary performance criteria against which STAFFSIM is evaluated. In order to fully understand what is required to initialize a scenario the opposing orders of battle and courses of action are presented below.

1. Order of Battle

The opposing orders of battle define the units that compose the attacking OPFOR regiment and defending friendly brigade. The units depicted in the order of battle diagrams represent aggregations of combat vehicles. Although STAFFSIM models combat between individual vehicles, the simulation map display depicts unit icons as shown in the order of battle diagrams. Staff officers are trained to represent men and equipment in this manner. Thus STAFFSIM's user interface uses these icons because their meaning and function is intuitively obvious to the target audience.

a. Opposing Orders of Battle

The OPFOR regiment is organized for combat with four motorized rifle battalions (MRB), each consisting of three motorized rifle companies (MRC) and an anti-tank platoon. STAFFSIM supports company and platoon size units thus the MRCs are the units seen in the simulation. The numbers underneath the icons in the diagrams are the unit "slant" reports. The slant report is simply a shorthand method of annotating the strength of a

unit on a vehicle basis. For example, the slant of 40/116/9 for the MRR means that the MRR is equipped with 40 tanks, 116 infantry-fighting vehicles and nine anti-tank vehicles. Figure 5.1 depicts the OPFOR order of battle.

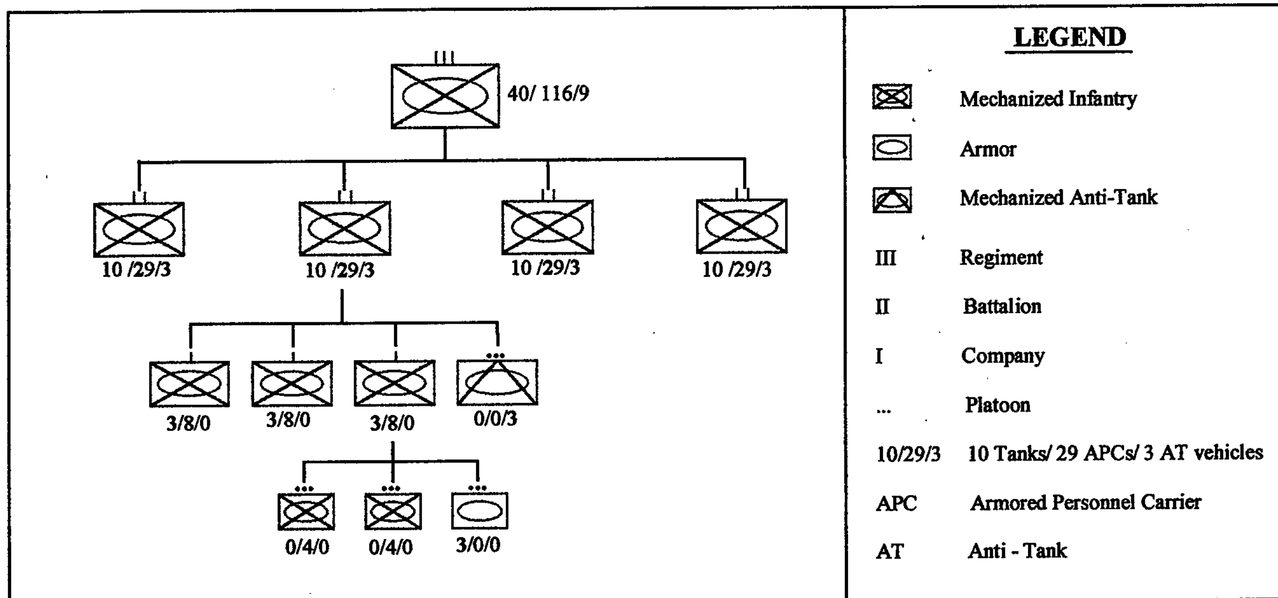


Figure 5.1: OPFOR Order of Battle

The friendly force is organized into two battalions, one with four companies, the other with two. A seventh company is held as the brigade reserve. The friendly slant figures represent tanks and infantry fighting vehicles only. Figure 5.2 presents the friendly order of battle.

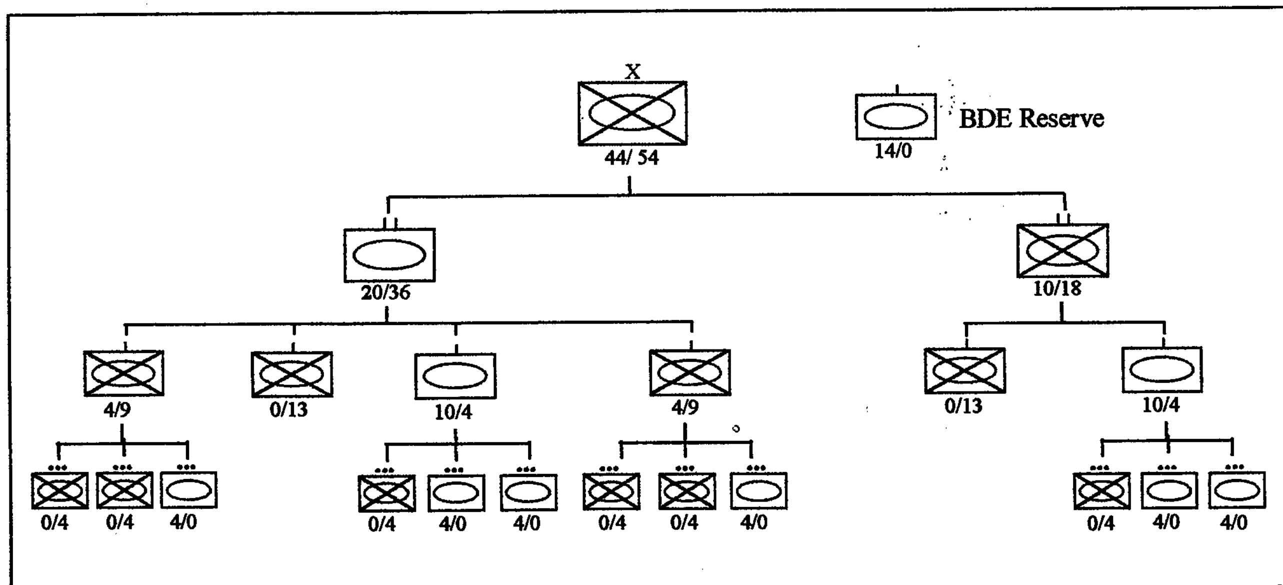


Figure 5.2: Friendly Forces Order of Battle

b. Order of Battle Input to STAFFSIM

STAFFSIM provides two techniques for unit construction, file input or unit construction with the graphic users interface (GUI). Either technique is accomplished using SimBuilder.

Unit construction boils down to the specification of a list of properties for the unit being built. Table 5.1 lists the properties that must be supplied to build platoons and companies. Although properties are specified for units, STAFFSIM takes these properties and uses them to construct both the unit and its component vehicles. In order to build a company the specified parameters must be supplied as well as up to five platoons.

SimBuilder Unit Construction Properties	
Platoon	**Company**
• Force Identifier	• Force Identifier
• Designation	• Designation
• Vehicle Type	• Position
• Number of Vehicles	• Formation
• Position	• Distance Between Sub-Units
• Formation	• Orientation
• Orientation	• Unit Type
• Distance Between Vehicles	
• Vehicle Rate of Fire	
• Vehicle Field of View	
• Vehicle Ammunition Load	

Table 5.1: Unit Properties

Figure 3.3 (page 28) depicts SimBuilder's CompanyBuilder panel. The company and platoon builder panels are simple point and click interfaces that allow rapid specification of the desired units. While the SimBuilder GUI is intuitive and fast, importing units by file can be much faster, once the unit file is built. Building unit files for scenario initialization can take some time but is a one-time exercise. Once one unit file exists, others can be rapidly created from the original in less than half the time required by the GUI interface. A full discussion of specific times is presented in later sections.

2. Courses of Action

While planning the defense the brigade staff develops several potential courses of action the enemy could pursue as well as several potential friendly courses of action. During the war game each friendly course of action is fought against each enemy course of action.

The analysis during the war game is used to synchronize the friendly course of action and ultimately leads to the selection of one friendly course of action. For the trial scenario a single friendly and single enemy course of action are presented below.

a. OPFOR Course of Action

The OPFOR course of action has the regiment attacking in advanced guard formation. The leading elements are the combat reconnaissance patrols (CRP) followed by the forward security element (FSE). The FSE is a company sized unit and is followed by the battalion sized advanced guard main body (AGMB). The mission of these forces is to gain intelligence, find or create potential weak spots in the enemy defense, or, if necessary, to fix a portion of the defending enemy force. Following the AGMB is the regimental main body. The main body seizes key terrain and attempts to defeat and penetrate the defending force to allow the regimental second echelon to seize the regiment's main objective.

The course of action is depicted in Figure 5.3 and has the regiment placing its main attack in the northern half of the zone. The FSE attacks in the south to both deceive the friendly force as to where the main attack will occur and to fix friendly forces defending in the south. The AGMB attacks in the north attempting to find a weakness in the defense or failing that, to create a weak point in the defense. The regimental main body follows the AGMB to complete the destruction of the defenders in the north and to create a penetration of the enemy defense. The regimental second echelon follows behind the main body with the mission of securing the regimental objective. This course of action is one of several the OPFOR could potentially pursue. For purposes of brevity it is the only COA war gamed in this discussion.

b. Friendly Course of Action

The friendly force defends in sector with two task forces abreast and a tank company in reserve. The brigade expects the brunt of the enemy attack to fall on only one battalion task force. Each defending battalion is prepared to counterattack into the flank of the MRR if it is not attacked. The battalion in the north defends with three companies forward and one in battalion reserve. The southern battalion defends with both of its companies forward. Figure 5.4 shows the friendly course of action. The arrows labeled 'A' and 'B' in the figure depict the planned counterattack axis mentioned above. If the enemy attacks in the north, the southern battalion will counterattack into the flank or vice versa. The

brigade reserve will be committed as a last resort.

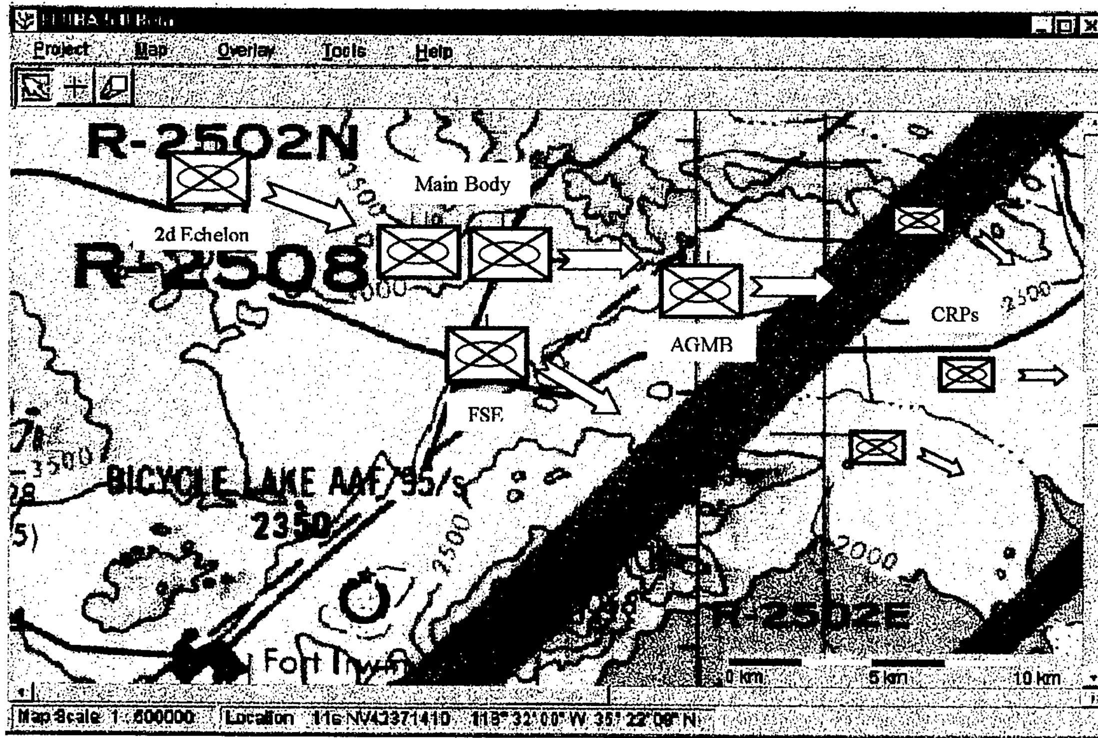


Figure 5.3: OPFOR Course of Action

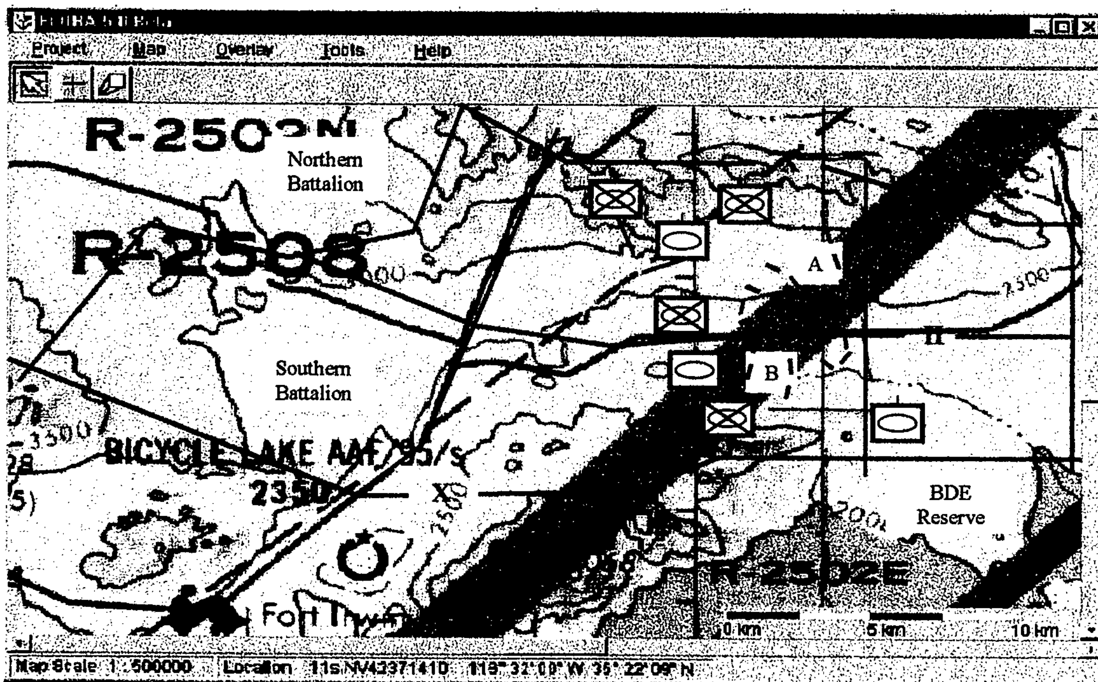


Figure 5.4: Friendly Course of Action

c. *Course of Action Input to STAFFSIM*

STAFFSIM's ExecutiveOfficer component allows users to rapidly assign movement orders to any unit in the simulation. Units can be assigned orders on the fly without even stopping the simulation. However, it is preferable to pause the simulation before assigning orders to units.

When a unit is assigned orders it immediately begins execution of the orders. If a unit is currently executing orders and is assigned new orders, the current orders are canceled in favor of the new ones. The ExecutiveOfficer allows staff officers to explore courses of action by assigning units new orders as unanticipated situations arise. The ability to stop the simulation, analyze options and assign new orders allows for rapid course of action refinement and more complete analysis.

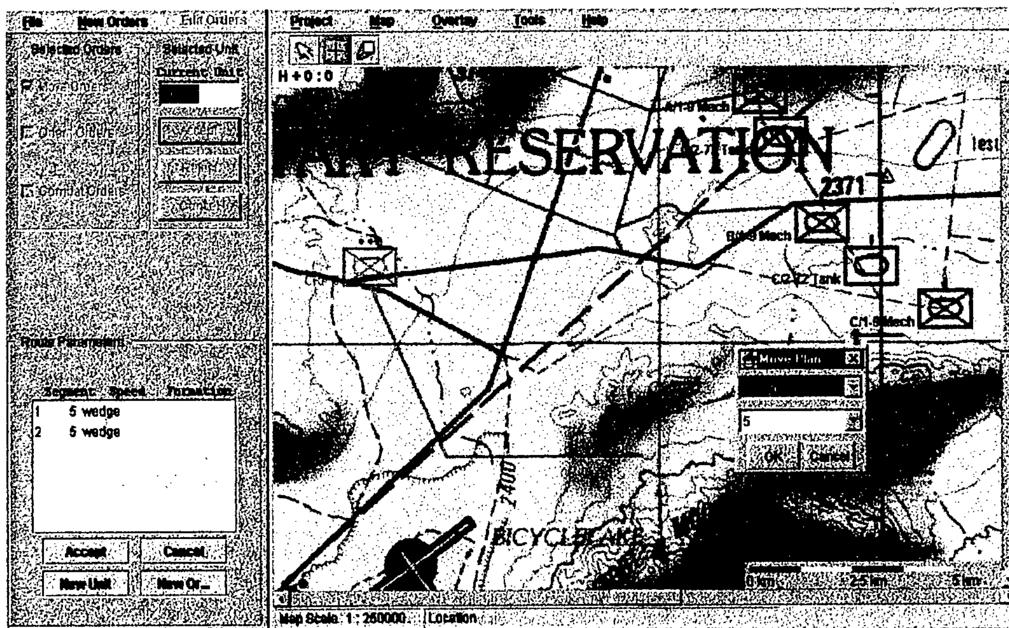


Figure 5.5: Assigning Unit Orders with ExecutiveOfficer

Figure 5.5 depicts the ease with which units can be assigned orders using ExecutiveOfficer. In the figure the user has selected CRP3 (highlighted in light red) and is assigning move orders. The line extending south from the unit and then to the east is the route the unit is being directed to follow. Route segments are added simply by clicking a desired destination on the map. For each route segment, the user can set the speed for the segment and the unit's movement formation. These parameters are input via the Move Plan

dialog box shown in the lower right hand corner of the map. Once the user is satisfied with the assigned orders ExecutiveOfficer is used to task the selected unit and the orders are executed. Assigning orders to units in this manner is fast and efficient allowing user interaction without substantially impacting the time required to run the simulation.

C. SCENARIO EXECUTION

Once the simulation has been initialized it is time to begin the war game. Figure 5.6 depicts the scenario as the first units of the MRR, the combat reconnaissance patrols, begin to enter the defending battalions' sectors. As the scenario unfolds the forward security element (FSE) attacks in the south followed by the Advanced Guard Main Body (AGMB) attacking in the north. As more and more enemy units enter the picture it becomes more and more difficult for the staff officers to completely visualize the potential options open to both sides not to mention conducting any type of analysis. To complicate the matter even further, as opposing units come into direct fire range the staff begins to spend large amounts of precious time debating outcomes. The debate about outcomes often eclipses any attempt at objective analysis and thus sidetracks the war game from its true purpose. In order to illustrate how STAFFSIM avoids unproductive debate and allows the staff to focus on synchronization; we will focus on the efforts of the AGMB and the main body to penetrate the brigade's defense.

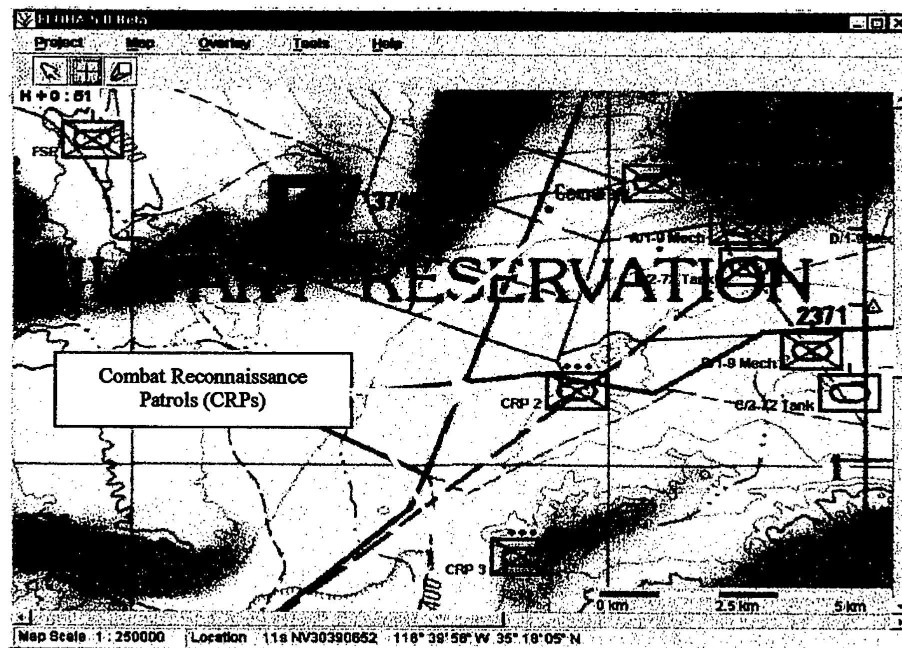


Figure 5.6: CRPs Enter Sector and Make Contact

As the scenario continues the AGMB moves into sector and attacks the battalion in the north. At this point in the typical war game the only visualization available to staff officers is one enemy icon representing the AGMB next to two icons representing the friendly companies. The icons themselves are typically oversized and usually obscure the map. As the AGMB moves into direct fire range, the assembled staff officers must envision the situation and think through time, space, unit capability relationships to eventually arrive at an outcome for the engagement. While doing this they must also consider the impact of combat multipliers such as artillery support, air support and obstacles. They must also evaluate the utility of things such as intelligence collection plans, planned decision points and reserve dispositions. Given the multitude of factors, the staff must consider and the complex relationships that must be though through it is easy to see how a staff can be sidetracked from true analysis. When the staff finally reaches a consensus about the outcome, in this case, say, the AGMB is destroyed for the loss of one friendly company, the staff moves on to the next critical event. No real analysis has occurred and synchronization has not been improved.

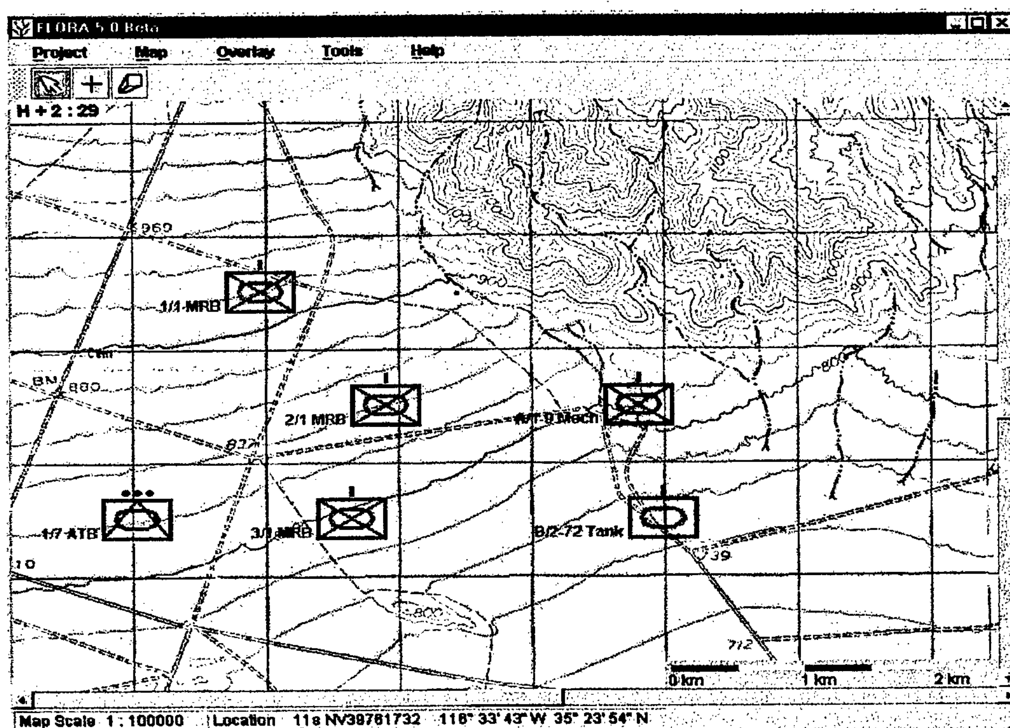


Figure 5.7: AGMB Assaults

Figure 5.7 shows the visualization STAFFSIM provides to the staff and STAFFSIM provides the outcomes. By providing an accurate visualization and showing within that visualization the time, space, unit capability relationships, the staff can analyze the situation and better synchronize the plan. From STAFFSIM's visualization it is easy to observe that the OPFOR has the opportunity to mass the AGMB against just the northern defending company. Given the terrain in the vicinity of the defense the potential exists for an attacker to achieve a significant local superiority. Furthermore, since the simulation provides the outcomes the staff can easily evaluate the defending company's ability to defeat the AGMB. If the probability of success is to low the staff can modify the plan as necessary to ensure that either the AGMB cannot mass against a single company or that if it does, the defending company is properly resourced to succeed. Analysis such as this allows the staff to ensure unit plans are feasible, properly synchronized and that all units have been assigned missions within their capabilities.

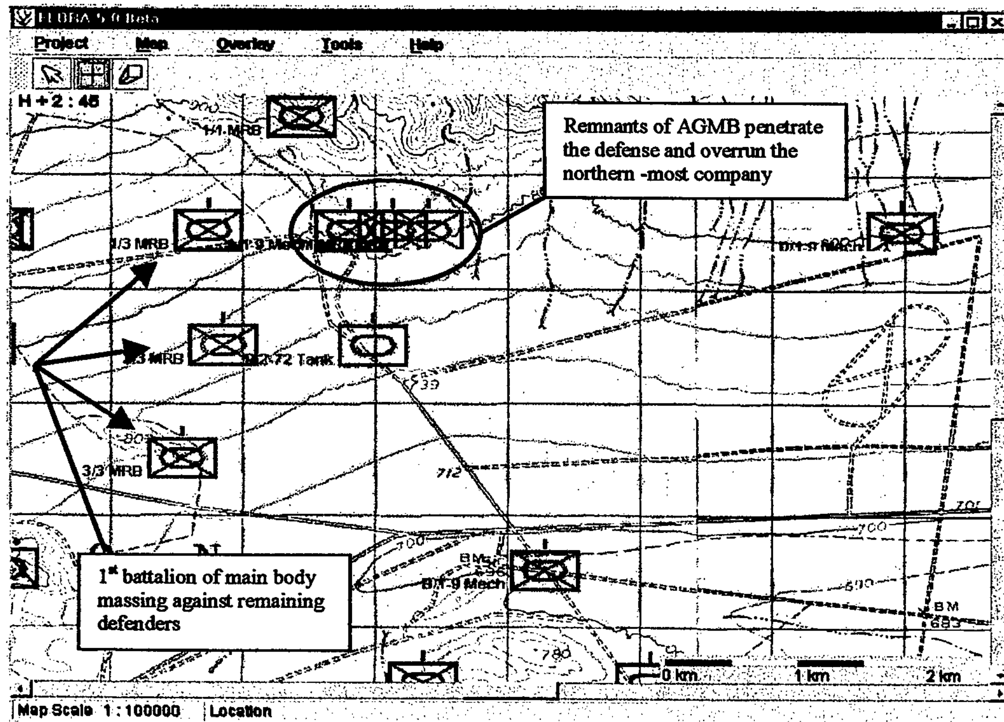


Figure 5.8: AGMB Penetrates the Defense

Returning to the typical war game, the AGMB has been destroyed at the cost of the northernmost defending company. Given the loss of the northernmost company, the northern battalion's reserve company would probably be committed to reinforce the surviving

company forward. The separation in time between the OPFOR's AGMB and main body is thirty minutes. The staff would normally judge that as sufficient time to move the reserve forward and re-establish a two-company defense before the OPFOR's main body arrives.

It is human nature to abstract events into discrete occurrences separated by time. However, to do so misrepresents the actual time space relationships in play on the battlefield. The attack of the AGMB and loss of the northernmost company take place over time, not at a discrete time. While that attack is occurring the main body is steadily closing the thirty-minute gap. By the time the defending force realizes that it must reinforce the defense the main body will have arrived and begun its attack. Thus the defenders plan is already becoming desynchronized as the attackers can mass up to two battalions on a single company. Worse still, the reserve company will arrive too late to influence the action and will itself have to fight two enemy battalions. In essence, the attackers have created a situation where they can mass against the defending companies one at a time, achieving overwhelming force ratios in each instance.

A simulation can prevent the kind of errors in calculating time distance relationships discussed above. Returning to STAFFSIM's visualization of the scenario, Figure 5.8 shows the scenario as the attack of the AGMB plays through and the main body arrives. It can clearly be seen that as the attack of the AGMB culminates in the destruction of the northernmost company the leading battalion of the main body has arrived and is massing against the remaining defending company. Furthermore, the remnants of the AGMB, about company size in strength, have penetrated the defense. STAFFSIM's visualization clearly shows that the enemy is succeeding in massing against single companies. Furthermore, the time space relationships discussed above are shown, they do not have to be thought through by the staff. Figure 5.8 also reveals that as the reserve company moves forward to reinforce the defense it can not possibly make it in time. Even if the reserve could make it in time it would have to fight the remnants of the AGMB in order to assume the positions of the destroyed company.

At this point the typical war game is usually hopelessly off track. Failure to understand time space relationships and focusing on outcomes and not analysis has combined to mislead the staff as to the feasibility of their course of action. By the time the two battalions of the main body have completed their attack the staff typically concludes that the

two defending companies and the battalion reserve company have been destroyed and that a unit the size of a motorized rifle company has penetrated the defense. The penetration of the defense triggers a counterattack by the southern battalion and the brigade reserve that successfully destroys the penetrating units and the regimental 2d echelon ending the war game. The staff would usually conclude that the COA is feasible and could even recommend to the commander that the brigade implement it.

Figure 5.9: Main Body Penetrates the Defense

Figure 5.9 shows that a simulation can reveal a far different picture. The remnants of the AGMB have swung to the south in order to provide flank protection for the main body of the regiment. Although the AGMB is only company size in strength it can potentially disrupt counterattacks from the south long enough for additional forces to arrive. The two battalions of the main body are continuing the attack to the east. The northern battalion's reserve company is still alive but will almost certainly be wiped out by the advancing main body in short order. The battalion defending in the south is counterattacking with one company and the brigade reserve is enroute. The enemy's second echelon battalion has not been committed and is following closely behind the main body. Once again the defenders are desynchronized and committing companies piecemeal in a failing effort to save a lost battle. The simulation has revealed several flaws in the plan that must be corrected if it is to stand a reasonable chance of success.

On the battlefield at the NTC, results generally resemble those depicted in the simulation. Defending brigades never go into battle with a plan they believe has a poor

chance of succeeding. Yet almost invariably they lose, many times losing very badly. Why then do brigades fail? As discussed in chapter two, invariably part of reason for failure is a poorly synchronized plan. Brigades do not intentionally enter battle with poorly synchronized plans, usually they expend a great deal of time and effort trying to ensure their plan is sound. Unfortunately, given the mental complexity of war gaming most staffs fail to realize they are doing a poor job of synchronizing their plan.

This scenario has demonstrated some of the pitfalls of war gaming that a simulation can remedy. Time space relationships and their relevance to unit capabilities are very difficult to fully think through. A simulation can visualize these relationships for a staff providing them insight into what is and what is not possible in a given situation. A simulation can also remedy the natural tendency for staffs to discuss outcomes of battles, as opposed to analyzing situations. The underlying combat models in a simulation provide probable outcomes eliminating the need for any discussion of outcomes at all. This allows the staff to focus their cognitive energies on analysis and synchronization. A simulation can also give the staff the opportunity to experiment with several different solutions to a given problem. Analysis of this nature not only gives the staff a better understanding of the problem but can help to ensure workable solutions are selected for implementation.

This section has demonstrated the ability of simulation to assist the staff during war gaming. STAFFSIM is a prototype of the kind of simulation needed. STAFFSIM's set of features is limited to vehicle on vehicle combat and thus many of the combat multipliers found on modern battlefields have not been discussed. As combat multipliers are added to the scenario the complexity of the analysis increases dramatically. The additional cognitive workload that increased complexity places on the staff can be eased by simulation as well. Simulation can allow staff officers to focus on finding and analyzing solutions to the current tactical problem as opposed to wasting time thinking through details that are best presented visually by a computer. The following section addresses the ability of STAFFSIM to meet the requirements for a simulation presented in earlier chapters.

D. STAFFSIM VERSUS SIMULATION REQUIRMENTS

The first requirement for the new simulation is that it can run on a personal computer typically found in a brigade or battalion headquarters. STAFFSIM was developed on an Intel

based personal computer with one processor using Microsoft's Windows 98 operating system. The CPU clock speed was 400 megahertz with 128 MB of random access memory (RAM). The memory footprint for the version run in the trial scenario was just more than 102 MB. A breakdown of the memory figure is useful in understanding what is actually required by STAFFSIM to execute a scenario. Table 5.2 provides a memory breakdown for STAFFSIM.

STAFFSIM MEMORY REQUIRMENTS	
Program Component	**Memory Required**
Source Code	1.56 megabytes
DTED Elevation Data	2.88 megabytes
1:500,000 Mapping	3.56 megabytes
1:250,000 Mapping	9.97 megabytes
1:100,000 Mapping	31.30 megabytes
1:50,000 Mapping	52.48 megabytes

Table 5.2: Memory Requirements

From a memory standpoint STAFFSIM's requirements are not extensive and can be easily supported by most modern PCs. In the event memory becomes an issue it is easy to scale back STAFFSIM's requirements. For example, in the trial scenario the 1:50,000 mapping was not used at all. Furthermore, for all map scales two to three times the map area required was included in the mapping database. Efficient selection of the mapping required for a given scenario could reduce the total memory required to less than 40 megabytes.

The second key issue concerning utilization of a PC is speed. When run on a 400 megahertz system the simulation was sluggish. Although the trial scenario runs in less than three hours the slow pace renders STAFFSIM unusable for real-time analysis in its current configuration. However, no reasoned approach to optimizing the code has been attempted. Furthermore, STAFFSIM is only meant as a proof of concept. Given that the underlying concepts are sound, a professionally coded simulation can almost certainly meet the requirements for real-time use. In fact, STAFFSIM itself, once properly optimized stands the chance of being responsive enough for real-time use.

The second criterion requires fast and easy generation of new terrain models for the simulation. The terrain model used by STAFFSIM consists of two components, mapping and terrain elevation data. STAFFSIM imports terrain elevation data directly from the Digital Terrain Elevation Data (DTED) CD-ROMs produced by NIMA. The time required to read in and initialize the elevation model for a one-degree DTED square is less than five minutes. STAFFSIM uses DTED level one data. The mapping component of the terrain model is more complicated.

STAFFSIM uses NIMA ARC Digitized Raster Graphics as the source for the mapping used by Flora. STAFFSIM does not, however, physically generate the image files used by Flora. A third party application is used to generate the actual mapping image files. Generation of the mapping for the trial scenario required less than five hours.

Does a composite time of five hours to generate a complete terrain model from scratch meet the requirement for fast and easy terrain generation? Recalling from chapter one the time and expense required to generate new terrain models for some of the existing simulations it is easy to see that STAFFSIM is faster and simpler. The real question is however, is five hours fast enough? From an operational standpoint, it probably is. Even the fastest deploying troops do not expect to see themselves thrust into combat any faster than twenty-four hours, probably more. For forward-deployed troops, the area where they will potentially fight is well known and thus the mapping can be prepared ahead of time. Thus, five hours is almost certainly fast enough.

Scenario generation time is of critical importance for real time use. The staff planning process has been shown to be an intense, time critical effort where every minute counts. Scenario generation time in excess of thirty minutes cannot be supported. STAFFSIM has the ability to generate scenarios in less than thirty minutes, when run on two machines. The time required populating the simulation for the trial scenario using the GUI was thirty-eight minutes or roughly two minutes per company. To create the same unit files from scratch using a text editor such as Notepad required less than two hours. Importing units from text files is the preferred method. The two-hour time requirement is a one-time expense. Once one unit file exists, it is a simple matter of cut and paste to modify that file for a different scenario. Depending on the amount of changes that must be made a new unit file can be

prepared in less than ten minutes. Once the unit file is imported, the simulation requires less than three minutes to process it and build the required units.

The second piece of scenario initialization is input of the course of action into the simulation. Using the ExecutiveOfficer GUI the course of action for the trail scenario required twenty-eight minutes to input. Twenty-eight minutes is somewhat slow and must be improved if real time use is to be feasible. The course of actions in the trial scenario required extensive COA input for only one side. If the trail scenario had required extensive COA input for both sides the thirty-minute limit would have been exceeded. However, if the friendly and enemy COAs are built on separate computers, the files can then be merged and run on the same machine. In this manner the thirty-minute limit can be achieved.

The final two criteria are that the simulation is easily upgraded and need no special support staff. Hosting the simulation on a single PC eliminates many of the reasons support staff are required for the currently fielded simulations. STAFFSIM does not require any hardware setup, running of cables, or specialized software installation and initialization. STAFFSIMs user interface is designed to be intuitive to the target audience and does not require any special training beyond reading a users manual. STAFFSIM and all supporting software can be downloaded over a network and installed simply by following a one or two page instruction sheet.

E. SUMMARY

Simulation support for real-time decision making is achievable using STAFFSIM STAFFSIM can provide valuable support to the Army's staff planning process, particularly to course of action analysis. It can successfully visualize a course of action for the staff, relieve the staff from the difficult task of envisioning complex time, space, unit capability relationships and provide probabilistic outcomes to engagements. The use of a simulation in this manner can focus staffs on synchronizing courses of action, prevent time wasting debate about outcomes and speed the course of action analysis process. The end result is a better plan that is more fully synchronized and thus better positions a unit for success on the battlefield.

As a prototype war game simulation, STAFFSIM has demonstrated the fundamental utility of simulation to the war gaming process. It has been shown how a simulation supports

the war game by helping to achieve the results discussed above. STAFFSIM itself has several shortcomings that prevent its immediate application by unit staffs. While STAFFSIM's current prototype implementation does not have the run-time performance needed for operational use, it successfully demonstrates the feasibility of the underlying architecture.

VI. CONCLUSIONS

A. CONCLUSIONS

The use of a simulation in the war gaming step of the Military Decision-Making Process can reduce the cognitive workload on staff officers in three important ways.

- The simulation visualizes the battlefield situation with respect to time for the staff officers. Officers will no longer have to envision in their mind's eye the precise sequencing of events and spatial relationships between units.
- The simulation relieves the officers from the tedious and difficult task of mentally thinking through complex time, space, unit capability relationships. The simulation will demonstrate these relationships allowing officers to rapidly and accurately assess what is and is not possible with respect to time, distance and unit capabilities.
- The simulation provides the outcomes to all engagements. The combat models embedded within the simulation determine the most probable outcome for unit on unit engagements. Thus the subjective decisions arrived at in current war games can be replaced with objective results based on tested and accepted combat models.

These factors combine to significantly reduce the mental workload imposed on staff officers during the war game. By reducing mental workload and replacing subjective outcome decisions with objective combat models a simulation can allow the staff to focus on analyzing the course of action. True course of action analysis as opposed to simple discussion of outcomes will result in better synchronized battle plans which in turn will better position friendly forces for success on the battlefield.

The latest generation of personal computers are now powerful enough and have enough storage space to run high resolution combat models. The Army's current set of high-resolution combat simulations was designed well over a decade ago. At that time it was unthinkable to use computer simulation in a real time decision making process. The complexity of the systems required to run the simulations and the support staff required to do so prevented their use in anything but fixed site simulation centers.

It is now possible to implement simulations using high-resolution combat models on personal computers. If properly designed, these simulations can be used in real time decision making. STAFFSIM is a proof of concept demonstrating how simulation can be used to improve course of action analysis in the Military Decision-Making process. Use in real time environments requires that the interface to the simulation be consistent with the training and doctrine of the target audience.

B. FUTURE WORK

The work completed on STAFFSIM thus far constitutes the base architecture for the simulation. STAFFSIM's run-time performance needs to be enhanced by optimizing its code. Furthermore, STAFFSIM models only vehicle on vehicle combat. These factors combine to suggest four areas where significant future work is required; implementation of acceptable high-resolution combat models, addition of the all the Battlefield Operating Systems (BOS) to the simulation, improvement of the systems performance, and experimentation in the field.

1. High Resolution Combat Models

The only combat model in STAFFSIM that is a standard army model is its use of the Janus line of sight algorithm. STAFFSIM is ready to have the basic army models plugged into its components, as described in chapter four. For example, the Army's Acquire model for sensing and detection could be added to the BasicSensor component. Incorporation of these models is important for the following reason. These models have been extensively tested and are accepted as valid throughout both the tactical and simulations communities within the Army. An attempt to use models other than currently accepted ones could result in dismissal of the entire concept of simulation support for real time decision making based not on its merits but on the use of untested models.

2. Battlefield Operating Systems

In its current state STAFFSIM does not model indirect fires, dismounted infantry, close air support, chemical munitions, command and control, engineers or army aviation. When the brigade staff analyzes a course of action all of these factors must be carefully considered. For a simulation to be useful to a brigade staff it must model all the elements and

capabilities of the brigade. Thus, it is important for STAFFSIM to include these factors as development continues.

3. System Performance

STAFFSIM's speed of execution needs improvement. Real time use by a group of assembled staff officers requires a crisp response from the user interface and speed of execution from the simulation. In order to improve overall performance, STAFFSIM should be profiled to determine where the bottlenecks exist. Once the bottlenecks have been identified, general solutions that preserve the architecture can be implemented. Additionally, the degree of complexity and multitude of independent tasks accomplished by the simulation suggest that a threading model may help improve performance. The results of profiling the simulation may suggest certain tasks or even components that are candidates for their own thread. Performance improvements in the simulation should lead to a more responsive GUI as code bottlenecks that prevent timely execution of the Java event thread are eliminated.

4. Field Experimentation

Once the improvements discussed above have been accomplished STAFFSIM must be tested in a field environment. Only field tests can truly determine the feasibility of simulation support for the Military Decision-Making Process.

C. SUMMARY

This thesis contends that the time has arrived for the use of simulation to support real time decision-making. Currently, the Army uses a wide variety of simulations at the tactical level to train troops on a multitude of tasks. As computers have become smaller and more powerful it has become possible to operate complex simulations on personal computers. Simulations run on PCs can deploy with tactical units and be used by unit staffs in the field. This thesis presents a prototype of one such simulation designed for use by brigade staffs to analyze courses of action. It has been demonstrated that unit staffs continually have difficulty conducting course of action analysis resulting in less than optimal unit performance at the Army's combat training centers. This thesis has demonstrated that a complex, high-resolution

simulation can be run on a single PC and that such a simulation can most probably improve staff performance of course of action analysis.

APPENDIX A: SELECTED IMPLEMENTATION CODE LISTINGS

Table of Contents for the Code Listings

I. BASE INTERFACES 80

A. MOVER.JAVA 80
B. SENSOR.JAVA 81
C. WEAPON.JAVA 83
D. FIREDIRECTION.JAVA 84

II. INTERFACE IMPLEMENTATIONS 85

A. BASICMOVER.JAVA 85
B. BASICSENSOR.JAVA 92
C. BASICWEAPON.JAVA 97
D. FIRECONTROL.JAVA 100

```java
/**
 * Author: Bill Bohman
 * Originated: 30 Nov 98
 * Version: 0.0
 * Updates:
 * To Do:
**/

package StaffSim;

import modkit.*;
import modutil.spatial.*;

public interface Mover {

    public void setStartPos(Coor3D setValue);    // Position from which current move started or
    public Coor3D getStartPos();                 // if not moving the current position

    public void setInitialPos(Coor3D initPos);   // Position at simulation time equal to zero
    public Coor3D getInitialPos();

    public void setEndPos(Coor3D newEndPos);     // Position at which current move will end

    public Coor3D getCurrentPos();               // Current Position at simtime when method called

    public Coor3D getVelocity();                 // true velocity, direction and speed

    public double getCurrentSpeed();             // current speed, magnitude only, no direction
    public double getMaxSpeed();                 // Movers maximum attainable speed (kph)

    public Coor3D getDirection();                // unit vector in direction of move, cartesian

    public double getAzimuth();                  // current direction referenced from grid north

    public void setFinalAzimuth(double value);   // azimuth mover will assume at the end of
    public double getFinalAzimuth();             // current move or if stationary, current azimuth

    public double getStartTime();                // time current move started

    public double getEndTime();                  // time current move will end

    public String getName();                     // retrieve the Mover's name

    public boolean isMoving();                   // is Mover currently moving

    public void stopMove(double delay);          // stop current move at current simtime + delay

    public double calcMoveDistance(double time, double speed);          // how far can be moved

    public void moveTo(Coor3D destination, double spd, double delay);  // results in scheduling
                                                                        // of a move event

    public void addModEventListener(ModEventListener eavsDropper);     // add a listener

    public void generateMoveEvent(String evtName, double delay, double speed, double prior);

} // end interface Mover
```

```
/**
 * Author: Bill Bohman
 * Originated: 30 Nov 98
 * Version: 0.0
 * Updates:
 * To Do:
**/

package StaffSim;

import modkit.*;
import modutil.spatial.*;
import java.util.*;

public interface Sensor {

   public ModComponent getParent();                       // component that owns this sensor

   public double getSensorMaxRange();                     // sensors maximum range in kilometers

   public void setSensorOrientation(double orientTo);     // center of sensors search area
   public double getSensorOrientation();                  // referenced to grid north, generates
                                                          // a sensor changed orientation event

   public void adjustSensorOrientation(double orientTo);  // changes sensor orientation w/o
                                                          // generating a sensor changed
                                                          // orientation event

   public double getSensorFieldOfView();                  // width of sensor's fov in radians

   public boolean inFieldOfView(Coor3D tgtVehPos);        // is target in the sensors area of
                                                          // search
   public boolean inFieldOfView(Coor3D tgtVehPos, Coor3D snsVehPos);

   public double getSensorLeftLimit();                    // angle from sensor to specified
   public double getSensorRightLimit();                   // limit in radians, referenced to
                                                          // grid north

   public Coor3D getSensorVelocity();                     // speed & direction sensor is moving
   public Coor3D getSensorLocation();                     // sensor's current location, these
                                                          // two properties are retrieved from
                                                          // the parent, they are not resident
                                                          // in the sensor

   public Vector getTrackList();                          // list of 'Target'(s) sensor is
   public void printTrackList();                          // is tracking

   public boolean isTracking(Vehicle tgtVeh);             // is specified vehicle being tracked
   public boolean isDetected(ModComponent target);

   public void setActive(boolean onOff);                  // is sensor searching or not
   public boolean getActive();

   public void addDetection(Target target);               // add Target to the tracklist
   public void removeDetection(Target target);            // remove traget from track list

   public double getTimeToDetection();                    // returns time in hours until the
   public double getTimeToClassify(Target tgt);           // specified event, these methods
   public double getTimeToIdentify(Target tgt);           // specify the detection algorithms
                                                          // used by the sensor
```

```
public void targetClassified(Target ghost);            // notification to the sensor that
   public void targetIdentified(Target ghost);            // these events have occurred
   public void targetChangedVelocity(Target ghost);
   public void targetMissed(Target ghost);

   public String getName();                                // get sensors name

   public void addModEventListener(ModEventListener eavsDropper);  // add a listener

} // end interface Sensor
```

```
/**
 * Author: Bill Bohman
 * Originated: 8 Jan 99
 * Version: 0.0
 * Updates:
 * To Do:
**/

package StaffSim;

import modkit.*;
import java.util.*;

public interface Weapon{

   public void setWeaponOrientation(double orientation);  // direction weapon is facing in
   public double getWeaponOrientation();                  // radians reference to grid north

   public double getWeaponLeftLimit();                    // angle to specified limit in
radians
   public double getWeaponRightLimit();

   public double getWeaponFieldOfView();                  // width of fov in radians

   public double getWeaponMaxRange();                     // weapon max range in kilometers

   public int getAmmoAvailable();                         // number of rounds on hand

   public String getWeaponName();                         // weapons name

   public ModComponent getParent();                       // component that owns the weapon

   public void shoot(Target targetToShoot, Sensor detectingSensor);  // tell weapon to fire
                                                                     // generates a 'Fire'
event

} // end interface Weapon
```

```java
/**
 * Author: Bill Bohman
 * Originated: 14 Jun 99
 * Version: 0.0
 * Updates:
 * To Do:
**/

package StaffSim;

public interface FireDirection {

   public void engageTarget(Sensor detectingSensor, Target newTgt);  // determines if and when
   public void engageTarget(Sensor detectingSensor);                 // to engage a new target

   public Weapon selectFiringWeapon(Target target, double range);    // determines which weapon
                                                                     // to shoot at the target

} // end interface FireDirection
```

```
/**
 * Authors: Bill Bohman
 * Originated: 20 Oct 98
 * Version: 0.3
 * Updates: 1) 4 Nov 98  --> Converted from simkit.smd.coordinate to
 *                           modkit.modutil.spatial.Coor3D
 *          2) 7 Nov 98  --> included capability to move to several waypoints in sequence by
 *                           the inclusion of the 'path' private data memeber and by
 *                           overloading the moveTo() method.
 *          3) 14 Nov 98 --> swithched over to SimkitAdapter technique, discarded SimModEvent
 *                           and MyBasicModEvent()
 *          4) 30 Nov 98 --> made BasicMover implement the mover interface
 *          5) 7 Jan 99  --> added 'classification' data member
 *
 * To Do: 1) Fix Interrogator
 *        2) replace console i/o error checking in the constructor(), generateSimModEvent(),
 *           and setCurrentSpeed() with GUI dialog boxes
 *
 * Notes: 1) This Mover generates the following events, event priority is included in
 *           parenthesis. For priority low numbers = high priority
 *                        InitialPlacement (0.0)          EndMove  (3.0)
 *                        StartMove  (0.0)                EndSegment  (3.0)
 *                        StartSegemnt  (0.0)
 *        2) This Mover is a smooth linear mover, i.e. when this mover begins to move it
 *           instantaneously jumps to cruising speed (as set by user), moves the required
 *           distance and direction (also as set by user) then instaneously stops. This mover
 *           has no ability to accelerate or decelerate
 *        3) To use this class as the base for a more complex mover the following methods need
 *           to be over ridden, calcMoveTime(), getCurrentPos(), calcMoveDistance()
**/

package StaffSim;

import StaffSim.*;                   // for Class CoordConverter --> for getting elevations
import StaffSim.Shared.*;
import StaffSim.Events.*;
import StaffSim.Shared.Terrain.*;
import simkit.*;                     // for Schedule etc.
import modkit.*;                     // for BasicModComponent etc.
import modutil.spatial.*;            // for Coor3D
import thistle.flora.coord.*;
import java.util.*;                  // for Vector

public class BasicMover extends BasicModComponent implements Mover{

  private Coor3D startPos,     // position from which moves begin or position when stationary
                 endPos,       // position at end of a move
                 initialPos,   // first position in the simulation, used for reset
                 velocity;     // direction and speed of movement
  private double maxSpeed,     // maximum allowable speed
                 moveTime,     // time required to complete currrent move
                 startTime,    // time current move started, if stationary, time last move ended
                 endTime,      // time current move will end
                 finalAzimuth;// direction vehicle is to be pointing after last move
  private Vector path;         // sequence of waypoints that define a path of movement
  private static int identity;// for unique naming
  private static SimkitAdapter sa;


  static{
     sa = new SimkitAdapter();
     identity = 0;
  } // end static initializations

//----------------------------------------------------------------------------------------------
// Constructors
//----------------------------------------------------------------------------------------------

  public BasicMover(String name, Coor3D position, double mSpeed){
     super(new String(name + identity++ + " "), true);  // allow self introspection
     startPos = new Coor3D(position);               // set to user provided value
     initialPos = new Coor3D(position);             // set to user provided value
```

```
    endPos = new Coor3D(-1.0, -1.0, -1.0);          // set for consistency/error checking
    velocity = new Coor3D(0.0, 0.0, 0.0);           // start as not moving thus no velocity
    setMaxSpeed(mSpeed);                            // set to user supplied value
    setMoveTime(0d);                                // set for consistency/error checking
    setStartTime(0d);                               // set for consistency/error checking
    setEndTime(0d);                                 // set for consistency/error checking
    setFinalAzimuth(0d);
    setParent(null);                                // if added to container, container sets
    setVerbose(false);
    path = new Vector();                            // allocate memory for the vector
    addModPropertySource(this);                     // property source for self
    addModEventListener(this);                      // listen to own events, thus will hear own
                                                    // scheduled events when they occur and can
                                                    // then take the appropriate action
    generateMoveEvent("InitialPlacement", 0.0, 0.0); // notify listeners of existence
  } // end constructor                                // and initial location

  public BasicMover(String name, Coor3D position, double mSpeed, double orient){
    this(name, position, mSpeed);
    setFinalAzimuth(orient);
  } // end constructor

//-----------------------------------------------------------------------------------------
// Properties
//-----------------------------------------------------------------------------------------

  public void setStartPos(Coor3D location){startPos = location;}          // StartPos
  public Coor3D getStartPos(){return new Coor3D(startPos);}

  public void setInitialPos(Coor3D location){                             // InitialPos
    initialPos = location;
    startPos = location;
  } // end setInitialPos

  public Coor3D getInitialPos(){return new Coor3D(initialPos);}

  public void setEndPos(Coor3D location){endPos = location;}              // EndPos
  public Coor3D getEndPos(){return new Coor3D(endPos);}

  public void setVelocity(Coor3D vel){velocity = vel;}                    // Velocity
  public Coor3D getVelocity(){return new Coor3D(velocity);}

  public void setMaxSpeed(double mSpeed){                                 // MaxSpeed
    mSpeed = checkLessThanZero(mSpeed, "maxSpeed");
    maxSpeed = mSpeed;
  } // end setMaxSpeed

  public double getMaxSpeed(){return maxSpeed;}

  public void setCurrentSpeed(double cSpeed){                             // CurrentSpeed
    if (isMoving()){
      cSpeed = checkSpeed(cSpeed);
      // retrieve current direction and multiply by new speed
      Coor3D direction = getDirection();
      setVelocity(new Coor3D(direction.getX() * cSpeed,
                             direction.getY() * cSpeed,
                             direction.getZ() * cSpeed));
      // recalculate move parameters
      moveTo(getEndPos(), cSpeed, 0.0);
    } // end if
    else{
      System.out.println(getName() + " cannot change speed because " +
                         getName() + " is not currently moving");
    } // end else
  } // end setCurrentSpeed

  public double getCurrentSpeed(){
    // current speed is embedded in velocity & must be extracted
    Coor3D temp = getVelocity();
```

```
   return Math.sqrt(temp.getX() * temp.getX() +
                    temp.getY() * temp.getY() +
                    temp.getZ() * temp.getZ());
} // end getCurrentSpeed

public Coor3D getDirection(){
   // current direction is embedded in velocity and must be extracted     // direction
   // retrieve X, Y and Z components of velocity
   double xComponent = endPos.getX() - startPos.getX();
   double yComponent = endPos.getY() - startPos.getY();
   double zComponent = endPos.getZ() - startPos.getZ();

   // calculate magnitude of velocity vector
   double magnitude = Math.sqrt(xComponent * xComponent +
                                yComponent * yComponent +
                                zComponent * zComponent);

   // calculate direction (in unit vector form) of velocity vector
   Coor3D direction = new Coor3D(xComponent / magnitude,
                                 yComponent / magnitude,
                                 zComponent / magnitude);
   return direction;
} // end getDirection

public double getAzimuth(){                                               // azimuth
   // azimuth is embedded in the velocity vector
   if (getVelocity().norm() == 0){
      return getFinalAzimuth();
   } // end if
   else {
      return Math.atan2(getVelocity().getX(), getVelocity().getY());
   } // end else
} // end getAzimuth

public void setFinalAzimuth(double facing){finalAzimuth = facing;}        // final azimuth
public double getFinalAzimuth() {return finalAzimuth;}

public Coor3D getCurrentPos(){                                            // CurrentPos
    // if not moving current position is StartPos
    if (!isMoving()){
       return startPos;
    } // end if
    // if we are moving calculate the time since move began, then
    // calculate distance covered since move began, then add distance
    // covered to start position to get current position
    else {
       Coor3D deltaMove = (Coor3D)velocity.scalarMul(Schedule.simTime() - startTime);
       Coor3D newPos = (Coor3D)startPos.add(deltaMove);
       FloraCoordinate currentPos = CoordConverter.getFloraPosition(newPos);
       newPos.setZ(ElevationManager.getElevation(currentPos.getLatLong()));
       return newPos;
    } // end else
 } // end getCurrentLocation

public void setMoveTime(double time){                                     // MoveTime
   time = checkLessThanZero(time, "moveTime");
   moveTime = time;
} // end setMoveTime

public double getMoveTime(){return moveTime;}

public void setStartTime(double time){                                    // StartTime
   time = checkTime(time, "startTime");
   startTime = time;
} // end setStartTime

public double getStartTime(){return startTime;}

public void setEndTime(double time){                                      // EndTime
   time = checkTime(time, "endTime");
   endTime = time;
```

```
   } // end setEndTime

   public double getEndTime(){return endTime;}

   public boolean isMoving(){                                          // Moving
      // isMoving is embedded in velocity, if velocity is not zero then
      // isMoving is true, else false
      if (getVelocity().norm() != 0){
         return true;
      } // end if
      else return false;
   } // end is moving

   public void setPath(Vector newRoute){                               // Path
      path.removeAllElements();
      path = newRoute;
   } // end setPath

   public Vector getPath(){return (Vector)path.clone();}

   // Move property allows a parent (ModContainer) to move by
   // setting the Move property of its Mover component
   public void setMove(Object[] params){
      Vector route = (Vector)params[0];
      Double temp1 = (Double)params[1];
      Double temp2 = (Double)params[2];
      double speed = temp1.doubleValue();
      double delay = temp2.doubleValue();
      moveTo(route, speed, delay);
   } // end setMove()

   public Vector getMove(){return (Vector)path.clone();}

//-----------------------------------------------------------------------------------------------
// Movement Methods
//-----------------------------------------------------------------------------------------------

   public double calcMoveTime(Coor3D start, Coor3D stop, double speed){
      double distance = start.distTo(stop);
      setMoveTime(distance / speed);
      return moveTime;
   } // end calcMoveTime

   public double calcMoveDistance(double time, double speed){
      return time * speed;
   } // end calcMoveDistance

   public void moveTo(Vector movePath, double spd, double delay){
      setPath(movePath);                                    // set the new route into path
      Coor3D nextWayPoint = (Coor3D)path.firstElement();    // get the 1st waypoint of the route
      path.removeElementAt(0);
      moveTo(nextWayPoint, spd, delay);                     // move to the 1st waypoint, see
   } // end moveTo                                          // handleStartMove for further moves

   public void moveTo(Coor3D destination, double spd, double delay){
      spd = checkSpeed(spd);                                // check for valid speed
      delay = checkLessThanZero(delay, "delay");            // check for valid delay

      double direction = Math.atan2(destination.getY() - getCurrentPos().getY(),
                                    destination.getX() - getCurrentPos().getX());

      // if vehicle is currently moving then there are several different cases to handle
      // --> Case 1) Vehicle is executing waypoints in it's path vector and has just completed
      //             an endSegment event and is beginning the next segment of it's route,
      //             therefore a startSegment must be scheduled
      // --> Case 2) Vehicle has been given an updated speed an/or destination in the middle
      //             of a move segment, therefore its current endMove event must be interrupted
      //             and a new endMove event must be scheduled

      if (isMoving()){
```

```
        setStartTime(Schedule.simTime());                          // start time = cur. time
        setStartPos(getCurrentPos());                              // start pos = cur. pos
        setEndPos(destination);                                    // endPos = destination
        setMoveTime(calcMoveTime(startPos, endPos, spd));          // set time to finish move
        setEndTime(Schedule.simTime() + moveTime);                 // set time move will end
        setVelocity((Coor3D)getDirection().scalarMul(spd));        // set velocity
        // vehicle is already moving therefore start a new segment
        generateMoveEvent("StartSegment", 0.0, spd, 0.0);
        return;
     } // end if isMoving()

     // else if not already moving we need to start moving
     else {
        setStartTime(Schedule.simTime() + delay);               // set time move begins
        setStartPos(getCurrentPos());                           // start pos = cur. pos
        setEndPos(destination);                                 // user sets destination
        setMoveTime(calcMoveTime(startPos, endPos, spd));       // set time to complete the move
        setEndTime(Schedule.simTime() + moveTime + delay);      // set time move will end
     } // end else                                              // in simkit
     generateMoveEvent("StartMove", delay, spd, 0.0); // generate 'StartMove' event
  } // end moveTo

  public void stopMove(double delay){
     if (isMoving()){
        generateMoveEvent("EndMove", 0.0, 0.0, 0.0);             // schedule EndMove event
     } // end if
  } // end stopMove()

//================================================================================================
// Utility Methods
//================================================================================================

  public void myDumpState(){
     System.out.print("\n" + getName() + " is at " + getCurrentPos());
     if (isMoving()){
        System.out.print(" moving to " + endPos + " at " + getCurrentSpeed() + " kph\n\n");
     } // end if
     System.out.println("\n\n");
  } // end dumpState

  public String toString() {return getName();}

  public double checkSpeed(double cSpeed){
     // ensure current speed is less than max speed and greater than zero
     while (cSpeed > maxSpeed || cSpeed < 0){
        if (cSpeed > maxSpeed){
           System.out.println("currentSpeed must be less than MaxSpeed");
           cSpeed = maxSpeed;
        } // end if
        if (cSpeed < 0){
           System.out.println("currentSpeed must be greater than zero");
           cSpeed = 0.0;
        } // end if
     } // end while
  return cSpeed;
  } // end checkSpeed

  public double checkLessThanZero(double numToCheck, String variableName){
     while(numToCheck < 0){
        numToCheck = Console.readDouble(variableName + "must be greater than " +
                                  "or equal to zero, enter a new value...");
     } // end while
     return numToCheck;
  } // end checkLessThanZero

  public double checkGreaterThanZero(double numToCheck, String variableName){
     while(numToCheck > 0){
        numToCheck = Console.readDouble(variableName + "must be less than " +
                                  "or equal to zero, enter a new value...");
     } // end while
     return numToCheck;
 } // end checkGreaterThanZero
```

```java
public double checkTime(double timeToCheck, String variableName){
    while(timeToCheck < Schedule.simTime()){
        System.out.println(variableName + "must be after current " +
                                         "simTime(), current simTime() is " +
                                         Schedule.simTime() + ", enter a new value");
        timeToCheck = Schedule.simTime();
    } // end while
    return timeToCheck;
} // end checkTime

public void printListeners(){
    for (Enumeration enum = listeners.elements(); enum.hasMoreElements();){
        ModEventListener ears = (ModEventListener)enum.nextElement();
        System.out.println(ears.toString());
    } // end for
} // end printListeners

//------------------------------------------------------------------------------------------
// Event Generators
//------------------------------------------------------------------------------------------

// for StartMove/StartSegment & EndMove/EndSegment Events
public void generateMoveEvent(String eventName, double delay, double speed, double
                              priority){
    Object[] params = new Object[8];
    params[0] = this;
    params[1] = new String("MoveEvent");
    params[2] = (getParent() == null ? this : getParent());
    params[3] = startPos;
    params[4] = endPos;
    params[5] = new Double(startTime);
    params[6] = new Double(endTime);
    params[7] = new Double(speed);
    sa.generateSimEvent(getName(), delay, params, priority, eventName);
} // end generateSimModEvent

// for Initial Placement Events
public void generateMoveEvent(String eventName, double delay, double priority){
    Object[] params = new Object[8];
    params[0] = this;
    params[1] = new String("MoveEvent");
    params[2] = (getParent() == null ? this : getParent());
    params[3] = getCurrentPos();
    params[4] = getCurrentPos();
    params[5] = new Double(Schedule.simTime());
    params[6] = new Double(Schedule.simTime());
    params[7] = new Double(getCurrentSpeed());
    sa.generateSimEvent(getName(), delay, params, priority, eventName);
} // end generateSimModEvent

//------------------------------------------------------------------------------------------
// Event Handlers
//------------------------------------------------------------------------------------------

public void handleInitialPlacement(MoveEvent evt){
    if (((BasicMover)evt.getSource()).equals(this)){             // if I generated this
        setStartPos(getCurrentPos());                               // event, update my
    } // end if                                                     // starting position
} // end handleInitialPlacement

public void handleStartMove(MoveEvent evt){
    if (((BasicMover)evt.getSource()).equals(this)){             // if I started moving
        Coor3D direct = getDirection();
        double spd = evt.getSpeed();                                // unwrap speed for move
        setVelocity((Coor3D)direct.scalarMul(spd));                 // set velocity vector
        setFinalAzimuth(getAzimuth());
        if(path.isEmpty()){                                         // if this is last leg...
            generateMoveEvent("EndMove", moveTime, spd, 3.0);       // schedule EndMove event
        } // end if
```

```
        else {                                                          // else...
           generateMoveEvent("EndSegment", moveTime, spd, 3.0);         // schedule endSegment
        } // end else
     } // end if
  } // end handleStartMove

  public void handleEndSegment(MoveEvent evt){
     if (evt.getSource().equals(this)){
        setStartPos(endPos);
        Coor3D nextWayPoint = (Coor3D)path.firstElement();
        path.removeElementAt(0);
        moveTo(nextWayPoint, getCurrentSpeed(), 0.0);
     } // end if
  } // end handleEndSegment

  public void handleStartSegment(MoveEvent evt){
     if (evt.getSource().equals(this)){
        Coor3D direct = getDirection();
        double spd = evt.getSpeed();                                    // unwrap the speed
        setVelocity((Coor3D)direct.scalarMul(spd));                     // set velocity vector
        setFinalAzimuth(getAzimuth());
        if(path.isEmpty()){                                             // if this is last leg...
           generateMoveEvent("EndMove", moveTime, spd,  3.0);           // schedule EndMove event
        } // end if
        else {                                                          // else...  schedule
           generateMoveEvent("EndSegment", moveTime,spd,  3.0);         // endSegment evt
        } // end else
     } // end if
  } // end handleStartSegment

  public void handleEndMove(MoveEvent evt){
     if (evt.getSource().equals(this)){               // if I finished moving
        Coor3D direct = getDirection();
        setFinalAzimuth(getAzimuth());
        setStartPos(endPos);                          // update startPos, startPos for next move
        setVelocity(new Coor3D(0.0, 0.0, 0.0));       // set velocity to zero
        setStartTime(Schedule.simTime());             // earliest possible time another move can
        setMoveTime(0.0);                             // start is the time the last move ended
        setEndTime(Schedule.simTime());
     } // end if
  } // end handleEndMove

} // end class BasicMover
```

```
/**
 * Authors: Arnold Buss & Bill Bohman
 * Originated: 7 Nov 98
 * Version: 0.1
 * Updates: 22 Nov 98 --> removed sensorLocation and sensorVelocity properties because those
 *                        properties are avialable in the parent property of BasicModComponent
 *                        which this class extends
 *          30 Nov 98 --> made BasicSensor implement the Sensor interface
 *
 * To Do: 1)
 *
 * Notes: 1) This sensor is a basic cookie cutter sensor. When a target enters the sensors
 *           range the mediator checks for and if necessary schedules Enter/Exit LOS events.
 *           When a target enters LOS time to detection is assumed to exponentially distributed
 *           with a mean time to detect of 5 mins or as set by the user.
 *        2) To use this class as the base for a more sophisticated sensor the following
 *           methods must be over written; getTimeToDetection(), getTimeToClassify(),
 *           getTimeToIdentify(), getRightLimit(), getLeftLimit, inFieldOfView()
**/

package StaffSim;

import StaffSim.Events.*;
import StaffSim.Shared.*;
import simkit.data.*;              // for Class RandomStream
import modkit.*;                   // for class BasicModComponent etc
import modutil.spatial.*;          // for class Coor3D
import java.util.*;                // for class Vector
import java.lang.reflect.*;        // for class Method

public class BasicSensor extends BasicModComponent implements Sensor{

   private double sensorMaxRange,      // maximum range at which a sensor can detect a target
                  sensorOrientation,   // direction the sensor is currently looking
                  sensorFieldOfView,   // width of the sensors field of view
                  sensorLeftLimit,     // left bound of sensors assigned sector of search
                  sensorRightLimit;    // right bound of sensors assigned sector of search
   private Vector trackList;           // list of all targets currently being tracked
   private String sensorName;          // identifying name of the sensor
   private boolean active,             // true = sensor is active, false = sensor is inactive
                   debug;              // for debugging, if true activates tracing
   private double meanTimeToDetect,    // mean time to detection after entering LOS
                  meanTimeToClassify,  // mean time to classify after detection occurs
                  meanTimeToIdentify;  // mean time to identify afetr classification occurs
   private static RandomStream rs;     // for exponential times to detection
   private static int identity;        // for unique naming

   static{
      rs = new RandomStream(RandomStream.STREAM_1);
      identity = 0;
   } // end static initializations


//===============================================================================================
// Constructors
//===============================================================================================

   public BasicSensor(double mRng, String id, boolean onOff, double orient, double fov) {
      this(mRng, id, orient, fov, onOff, 1.0/4.0, 1.0/8.0, 1.0/12.0);
   } // end constructor

   public BasicSensor(double mRng, String id, double orient, double fov) {
      this(mRng, id, orient, fov, true, 1.0/6.0, 1.0/12.0, 1.0/18.0);
   } // end constructor

   public BasicSensor(double mRng, String id, double orient, double fov, boolean onOff,
                      double mttd, double mttc, double mtti){
      super(new String(id + identity++ + " "), true);// allow self introspection
      setSensorMaxRange(mRng);                       // set user supplied values
      setSensorOrientation(orient);
```

```
      setSensorFieldOfView(fov);
      trackList = new Vector();                          // allocate memory
      setSensorName(id);                                 // set user supplied values
      setActive(onOff);                                  // set user supplied value
      debug = false;
      setMeanTimeToDetect(mttd);
      setMeanTimeToClassify(mttc);
      setMeanTimeToIdentify(mtti);
      addModEventListener(this);                         // listen to own events
   } // end constructor

//==============================================================================
// Properties
//==============================================================================

   public void setSensorMaxRange(double mRng) {                       // SensorMaxRange
      sensorMaxRange = checkGreaterThanZero(mRng, "sensorMaxRange");
   } // end set maxRange

   public double getSensorMaxRange() {return sensorMaxRange;}

   public void setSensorOrientation(double facing){                   // SensorOrientation
      sensorOrientation = facing;
      //System.out.println("Setting " + this + " orientation to " + facing);
      setSensorLeftLimit(facing - getSensorFieldOfView() / 2);
      setSensorRightLimit(facing + getSensorFieldOfView() / 2);
      generateEnterExitEvent("ChangedFieldOfView");
   } // end setOrientation()

   public void adjustSensorOrientation(double facing){
      sensorOrientation = facing;
      setSensorLeftLimit(facing - getSensorFieldOfView() / 2);
      setSensorRightLimit(facing + getSensorFieldOfView() / 2);
   } // end setOrientation()

   public double getSensorOrientation(){return sensorOrientation;}

   public void setSensorFieldOfView(double fov){                      // SensorFieldOfView
      sensorFieldOfView = fov;
      setSensorLeftLimit(getSensorOrientation() - fov / 2);
      setSensorRightLimit(getSensorOrientation() + fov / 2);
      generateEnterExitEvent("ChangedFieldOfView");
   } // getOrientation()

   public double getSensorFieldOfView(){return sensorFieldOfView;}

   private void setSensorLeftLimit(double ll){                        // SensorLeftLimit
      sensorLeftLimit = ll;
   } // end setSensorLeftLimit

   public double getSensorLeftLimit(){return sensorLeftLimit;}

   private void setSensorRightLimit(double rl){                       // SensorRighLimit
      sensorRightLimit = rl;
   } // end setSensorRightLimit

   public double getSensorRightLimit(){return sensorRightLimit;}

   public Coor3D getSensorLocation(){                                 // Location
      if (getParent() != null){
         return (Coor3D)getParent().getProperty("CurrentPos");
      } // end if
      return null;
   } // enf getSensorLocation

   public Coor3D getSensorVelocity() {                                // Velocity
      if (getParent() != null){
         return (Coor3D)getParent().getProperty("Velocity");
      } // end if
      return null;
   } // end getSensorVelocity
```

```
public void setTrackList(Vector newList) {trackList = newList;}          // TrackList

   public Vector getTrackList() {return trackList;}

   public void setSensorName(String id) {sensorName = id;}               // SensorName

   public String getSensorName() {return sensorName;}

   public void setActive(boolean onOff){                                 // Active
      if (active == false && onOff == true){
         generateGenericModEvent("ActivateSensor");
      } else if (active == true && onOff == false){
         generateGenericModEvent("DeactivateSensor");
      } // end else if
      active = onOff;
   } // end setActive

   public boolean getActive() {return active;}

   public void setMeanTimeToDetect(double mtd){meanTimeToDetect = mtd;}
   public double getMeanTimeToDetect(){return meanTimeToDetect;}

   public void setMeanTimeToClassify(double mtc){meanTimeToClassify = mtc;}
   public double getMeanTimeToClassify(){return meanTimeToClassify;}

   public void setMeanTimeToIdentify(double mti){meanTimeToIdentify = mti;}
   public double getMeanTimeToIdentify(){return meanTimeToIdentify;}

//==============================================================================================
// Utility Methods
//==============================================================================================

   public double checkGreaterThanZero(double numToCheck, String paramName){
      while(numToCheck < 0){
         numToCheck = Console.readDouble(paramName + " must be greater than zero, " +
                                         "enter a valid number...");
      } // end while
      return numToCheck;
   } // end checkGreaterThanZero


   public boolean isTracking(Vehicle tgtVeh){
      for (Enumeration enum = trackList.elements(); enum.hasMoreElements();){
         Target checkVeh = (Target)enum.nextElement();
         if (checkVeh.getName().equals(new String("Ghost-" + tgtVeh.getName()))){
            return true;
         } // end if
      } // end for
      return false;
   } // end isTracking()

   public void printTrackList(){
      System.out.println("Sensor " + getName() + " is tracking...");
      for (Enumeration enum = trackList.elements(); enum.hasMoreElements();){
         Target tempTarget = (Target)enum.nextElement();
         System.out.println("    " + tempTarget.getName());
      } // end for
   } // end printTrackList()

   public String toString() {return getName();}

   public void trace(String arg){System.out.println(arg);}

   public boolean inFieldOfView(Coor3D tgtVehPos){

      Coor3D snsVehPos = (Coor3D)(getParent().getProperty("CurrentPos"));
      double angle = Math.atan2(tgtVehPos.getX() - snsVehPos.getX(),
                                tgtVehPos.getY() - snsVehPos.getY());
      if (angle >= sensorLeftLimit && angle <= sensorRightLimit){return true;}
      if (angle >= -sensorRightLimit && angle <= -sensorLeftLimit){return true;}

      return false;
} // end inFieldOfView()
```

```
public boolean inFieldOfView(Coor3D tgtVehPos, Coor3D snsVehPos){
      if (debug) {trace("entering M1GunnersPrimarySight.inFieldOfView() w/args... " +
                        "\n   tgtVehPos = " + tgtVehPos + "\n   snsVehPos = " + snsVehPos);}

      double angle = Math.atan2(tgtVehPos.getX() - snsVehPos.getX(),
                                tgtVehPos.getY() - snsVehPos.getY());
      if (angle >= sensorLeftLimit && angle <= sensorRightLimit){return true;}
      if (angle >= -sensorRightLimit && angle <= -sensorLeftLimit){return true;}

      if (debug) {trace("returning false");}
      return false;
   } // end inFieldOfView()

//==========================================================================================
// Event Generators
//==========================================================================================

   public void generateGenericModEvent(String eventName){
      GenericModEvent newEvent = new GenericModEvent(this, eventName);
      notifyListeners(newEvent);
   } // end generateGenericModEvent

   public void generateEngageEvent(String eventName, Target tgt){
      EngageEvent newEvent = new EngageEvent(this, eventName, (Vehicle)getParent(), this, tgt);
      notifyListeners(newEvent);
   } // end generateGenericModEvent()

   public void generateEnterExitEvent(String eventName){
      if (debug){trace("entering BasicSensor.generateEnterExitEvent()");}
      EnterExitEvent newEvent = new EnterExitEvent(this, eventName, this, (Vehicle)getParent());
      notifyListeners(newEvent);
      if (debug){trace("exiting BasicSensor.generateEnterExitEvent()");}
   } // end generateEnterExitEvent
//==========================================================================================
// Event Handlers
//==========================================================================================

   public void handleActivateSensor(ModEvent evt){
      trackList.removeAllElements();                // ensure trackList is clear of all old
   } // end handleActivateSensor                    // tracks

   public void handleDeactivateSensor(ModEvent evt){
      trackList.removeAllElements();
   } // end handleDeactivateSensor

//==========================================================================================
// Detection Methods
//==========================================================================================

   public double getTimeToDetection(){
      return rs.exponential(meanTimeToDetect);
   } // end getTimeToDetect

   public double getTimeToClassify(Target tgt){
      return rs.exponential(meanTimeToClassify);
   } // end getTimeToDetect

   public double getTimeToIdentify(Target tgt){
      return rs.exponential(meanTimeToIdentify);
   } // end getTimeToDetect

   public void addDetection(Target target){
      if (!trackList.contains(target)){
         trackList.addElement(target);
         generateEngageEvent("NewTarget", target);
      } // end if
      else {
         System.out.println("Target " + target + " is already being tracked");
      } // end else
   } // end addDetection
```

```
public void removeDetection(Target target){
      if (trackList.contains(target)){
         trackList.removeElement(target);
      } // end if
      else {
         System.out.println("Sensor " + this + " is not tracking Target " + target);
      } // end else
   } // end removeDetection

   public boolean isDetected(ModComponent target){
      return trackList.contains(target);
   } // end isDetected

   public void targetChangedVelocity(Target ghost){
      generateEngageEvent("NewTarget", ghost);
   } // end targetChangedVelocity()

   public void targetMissed(Target ghost){
      generateEngageEvent("NewTarget", ghost);
   } // end targetMissed

   public void targetClassified(Target ghost){
      if (ghost.getDetectionStatus() == 3){
         generateEngageEvent("NewTarget", ghost);
      } // end if
   } // end targetClassified

   public void targetIdentified(Target ghost){
      if (ghost.getDetectionStatus() == 4){
         generateEngageEvent("NewTarget", ghost);
      } // end if
   } // end targetIdentified

} // end class BasicSensor
```

```
/**
 * Author: Bill Bohman
 * Originated: 8 Jan 99
 * Version: 0.0
 * Updates:
 * To Do:
**/

package StaffSim;

import StaffSim.Shared.*;
import simkit.*;
import modkit.*;
import modutil.spatial.*;
import java.util.*;

public class BasicWeapon extends BasicModComponent implements Weapon {

   private double weaponMaxRange;              // maximum engagement range for this weapon
   private int maximumBasicLoad,               // max # of rounds typically carried on vehicle
               ammoAvailable;                  // number of rounds currently on hand
   private double nextEngagementTime,          // is vehicle engaging at this time
                  timeToFire,                  // time to complete one engagement
                  weaponOrientation,           // azimuth of weapons center of sector
                  weaponFieldOfView,           // angular width of weapons sector of fire
                  weaponLeftLimit,             // radians, left limit of assigned sector
                  weaponRightLimit,            // radians, right limit of assigned sector
                  maxRateOfFire,               // rounds/minute
                  rateOfFire,                  // rounds/minute
                  roundsPerBurst;              // number of rounds expended each time wpn fired
   private String weaponName;                  // name of weapon for indexing kill tables
   private static int identity;                // for unique naming
   private static SimkitAdapter sa;            // for generating scheduled events
   static {
      sa = new SimkitAdapter();
      identity = 0;
   } // end static initializations

//=========================================================================================
// Constructors
//=========================================================================================

   public BasicWeapon(String name, double maxRng, int maxLoad, int load, double maxFire,
                      double typicalFire, double orient, double fov, String wpnName,
                      double rdsPerBurst){
      super(name, true);
      setWeaponMaxRange(maxRng);
      setMaxBasicLoad(maxLoad);
      setAmmoAvailable(load);
      setMaxRateOfFire(maxFire * 60.0);          // user inputs in rounds/minute, must convert
      setRateOfFire(typicalFire * 60.0);         // to rounds per hour
      setNextEngagementTime(0.0);
      setTimeToFire(1.0 / getRateOfFire());
      setWeaponOrientation(orient);
      setWeaponFieldOfView(fov);
      setWeaponName(wpnName);
      setRoundsPerBurst(rdsPerBurst);
      addModEventListener(this);
   } // end constructor

   public BasicWeapon(String wpnName){
      this(new String(" BasicWeapon-" + identity++ + " " ), 3.5, 40, 40, 360.0, 3.0, 0.0,
           Math.PI / 2, wpnName, 1.0);

      // name --> BasicWeapon-###
      // weaponMaxRange --> default to 3.5 kilometers
      // maximumBasicLoad --> default to 40 rounds, actual M1A1 capacity
      // ammoAvailable --> default to a full load
      // maxRateOfFire --> 360 rounds/hour = 6 rounds/minute
      // rateOfFire -> 180 rounds/hour = 3 rounds/minute
   } // end constructor
```

```
   public BasicWeapon(String name, int load, int rateOfFire, String wpnName){

      // note: rate of fire must be input in rounds per minute, is converted to rounds/hour
      this(name, 4.0, 40, load, 360, rateOfFire * 60, 0.0, Math.PI / 2, wpnName, 1.0);
   } // end constructor

   // this is constructor called by Vehicle Builder
   public BasicWeapon(String wpnName, int rateOfFire, int load, double maxRange,
                      double roundsBurst, double orientation, double fieldOfView){

      // note: rate of fire must be input in rounds per minute, is converted to rounds/hour
      this(new String("BscWpn-" + identity++), maxRange, 10000, load, rateOfFire * 60,
           rateOfFire * 60, orientation, fieldOfView, wpnName, roundsBurst);
   } // end constructor

   public BasicWeapon(String name, int rateOfFire, String wpnName, double maxRange,
                      double roundsBurst){

      // Note: rate of fire must be sent in in rounds per minute, is converted to rounds/hour
      this(name, maxRange, 40, 40, 360, rateOfFire * 60, 0.0, Math.PI / 2, wpnName,
           roundsBurst);
   } // end constructor

   public BasicWeapon(double maxRng, double orient, double fov, String wpnName){
      this(new String("BasicWeapon-" + identity++), maxRng, 40, 40, 360, 3, orient, fov,
           wpnName, 1.0);
   } // end constructor

//==========================================================================================
// Properties
//==========================================================================================

   public void setWeaponMaxRange(double maxRng) {weaponMaxRange = maxRng;}
   public double getWeaponMaxRange() {return weaponMaxRange;}

   public void setMaxBasicLoad(int maxLoad) {maximumBasicLoad = maxLoad;}
   public int getMaxBasicLoad() {return maximumBasicLoad;}

   public void setAmmoAvailable(int ammo) {ammoAvailable = ammo;}
   public int getAmmoAvailable() {return ammoAvailable;}

   public void setMaxRateOfFire(double maxRof) {maxRateOfFire = maxRof;}
   public double getMaxRateOfFire() {return maxRateOfFire;}

   public void setRateOfFire(double rof) {
      if (rateOfFire <= maxRateOfFire){
         rateOfFire = rof;
      } // end if
      else {
         rateOfFire = Console.readInt("Attempted to set rateOfFire > maxRateOfFire " +
                                      " re-enter rateOfFire here --> ");
      } // end else
   } // end setRateOfFire()

   public double getRateOfFire() {return rateOfFire;}

   public void setNextEngagementTime(double nextTime) {nextEngagementTime = nextTime;}
   public double getNextEngagementTime() {return nextEngagementTime;}

   public void setTimeToFire(double setValue) {timeToFire = setValue;}
   public double getTimeToFire() {return timeToFire;}

   public void setWeaponOrientation(double orient) {
      weaponOrientation = orient;
      setWeaponLeftLimit(orient - getWeaponFieldOfView() / 2);
      setWeaponRightLimit(orient + getWeaponFieldOfView() / 2);
   } // end setWeaponOrientation()

   public double getWeaponOrientation() {return weaponOrientation;}
```

```
   public void setWeaponFieldOfView(double fov) {
      weaponFieldOfView = fov;
      setWeaponLeftLimit(getWeaponOrientation() - fov / 2);
      setWeaponRightLimit(getWeaponOrientation() + fov / 2);
   } // end setWeaponOrientation()

   public double getWeaponFieldOfView() {return weaponFieldOfView;}

   public void setWeaponLeftLimit(double leftLim) {weaponLeftLimit = leftLim;}
   public double getWeaponLeftLimit() {return weaponLeftLimit;}

   public void setWeaponRightLimit(double rightLim) {weaponRightLimit = rightLim;}
   public double getWeaponRightLimit() {return weaponRightLimit;}

   public void setWeaponName(String wpnName) {weaponName = wpnName;}
   public String getWeaponName() {return weaponName;}

   public void setRoundsPerBurst(double rpb) {roundsPerBurst = rpb;}
   public double getRoundsPerBurst() {return roundsPerBurst;}

//==========================================================================================
// Utility Methods
//==========================================================================================

   public String toString() {return getName();}

//==========================================================================================
// Event generaters
//==========================================================================================

   public void generateEngageEvent(String sourceName, double delay, double prior,
                              String eventName, Sensor detectingSensor, Target ghost){
      Object[] eventParameters = new Object[6];      // build event object array
      eventParameters[0] = this;
      eventParameters[1] = new String("EngageEvent");
      eventParameters[2] = getParent();
      eventParameters[3] = detectingSensor;
      eventParameters[4] = this;
      eventParameters[5] = ghost;
      sa.generateSimEvent(getName(), delay, eventParameters, 0.0, "Fire");
   } // end generateEngageEvent

   public void shoot(Target tgt, Sensor detectingSensor){
      double delay;
      if (Schedule.simTime() >= nextEngagementTime){
         delay = timeToFire;
         setNextEngagementTime(Schedule.simTime() + timeToFire);
          if (ammoAvailable > 0){
            generateEngageEvent(getName(), delay, 0.0, "Fire", detectingSensor, tgt);
            ammoAvailable -= roundsPerBurst;
         } // end if
         else {
            System.out.println(this + " is out of ammunition");
         } // end if
      } // end if
      else {
         delay = nextEngagementTime - Schedule.simTime() + timeToFire;
         setNextEngagementTime(Schedule.simTime() + delay);
          if (ammoAvailable > 0){
            generateEngageEvent(getName(), delay, 0.0, "Fire", detectingSensor, tgt);
            ammoAvailable -= roundsPerBurst;
         } // end if
         else {
            System.out.println(this + " is out of ammunition");
         } // end else
      } // end else
   } // end shoot()

} // end class BasicWeapon
```

```
/**
 * Author: Bill Bohman
 * Originated: 8 Jan 99
 * Version: 0.0
 * Updates:
 * To Do:
**/

package StaffSim;

import StaffSim.Shared.*;
import modkit.*;
import modutil.spatial.*;
import java.util.*;

public class FireControl extends BasicModComponent implements FireDirection {

   private WeaponControl wpnCtrl;                // current weapons control status
   private TargetPriority tgtPriority;           // current target priorities
   private TreeSet masterTargetList;             // prioritized list of targets to engage
   private Vector targetsOnMasterTargetList;     // un-prioritized list of targets on MTL
   private static int identity;                  // for unique naming
   private static SimkitAdapter sa;              // for scheduling events

   static{sa = new SimkitAdapter();
          identity = 0;
   } // end static initializations

//==========================================================================================
// Constructors
//==========================================================================================

   public FireControl(String name){
      super(new String(name + "-" + identity++), true);
      wpnCtrl = new WeaponControl(0, 2, 2);   // wcsA = 1, wcsB = 2, triggerLine = 2.0
      tgtPriority = new TargetPriority(name);
      masterTargetList = new TreeSet(new TargetComparator());
      targetsOnMasterTargetList = new Vector();
      wpnCtrl.addModEventListener(this);
   } // end constructor

   public FireControl(){
      super(new String("FC-" + identity++), true);
      wpnCtrl = new WeaponControl(0, 2, 2);   // wcsA = 0, wcsB = 2, triggerLine = 2.0
      tgtPriority = new TargetPriority(getName());
      masterTargetList = new TreeSet(new TargetComparator());
      targetsOnMasterTargetList = new Vector();
      wpnCtrl.addModEventListener(this);
   } // end constructor

//==========================================================================================
// Properties
//==========================================================================================

   public void setWpnCtrl(WeaponControl setObj){wpnCtrl = setObj;}
   public WeaponControl getWpnCtrl(){return wpnCtrl;}

   public void setTgtPriority(TargetPriority setObj){tgtPriority = setObj;}
   public TargetPriority getTgtPriority(){return tgtPriority;}

   protected void setMasterTargetList(TreeSet tgtList){masterTargetList = tgtList;}
   public TreeSet getMasterTargetList(){return masterTargetList;}

   public void setTargetsOnMasterTargetList(Vector tgts){targetsOnMasterTargetList = tgts;}
public Vector getTargetsOnMasterTargetList(){return targetsOnMasterTargetList;}
```

```
//=========================================================================================
// Event Generators
//=========================================================================================

   public void generateEngageEvent(String eventName, double delay, Target tgt, Sensor ds,
                                   Weapon firingWeapon){
       Object[] evtParams = new Object[6];
      evtParams[0] = this;
      evtParams[1] = "EngageEvent";
      evtParams[2] = getParent();
      evtParams[3] = ds;
      evtParams[4] = firingWeapon;
      evtParams[5] = tgt;
      sa.interruptAll("NewTarget", evtParams);
      sa.generateSimEvent(getName(), delay, evtParams, 0.0, eventName);
   } // end generateScheduledEvent

//=========================================================================================
// Fire Control Methods
//=========================================================================================

   public void engageTarget(Sensor detectingSensor, Target newTgt){
      Coor3D wpnPos = (Coor3D)(getParent().getProperty("CurrentPos")); // retrieve own position
      double range = wpnPos.distTo(newTgt.getCurrentPos());            // and calculate range
      Weapon firingWeapon = selectFiringWeapon(newTgt, range);         // select a weapon

      if (wpnCtrl.inSector(newTgt.getCurrentPos(), wpnPos, firingWeapon)){
         if (wpnCtrl.engageInSector(newTgt, (Vehicle)getParent())){
            if (wpnCtrl.inRangeAndTrigger(newTgt, firingWeapon)){
               if(!targetsOnMasterTargetList.contains(newTgt)){   // if target is not already on
                  masterTargetList.add(newTgt);                   // the target list add it
                  targetsOnMasterTargetList.addElement(newTgt);
               } // end if
               Target targetToShoot = (Target)masterTargetList.first();
               masterTargetList.remove(targetToShoot);
               targetsOnMasterTargetList.removeElement(targetToShoot);
               firingWeapon = selectFiringWeapon(targetToShoot, range);
               firingWeapon.shoot(targetToShoot, detectingSensor);
            } // end if
            else {
               checkForTrigger(detectingSensor, newTgt, firingWeapon);
            } // end else
         } // end if
      } // end if
      else {
         if (wpnCtrl.engageOutOfSector(newTgt, (Vehicle)getParent())){
            if (wpnCtrl.inRangeAndTrigger(newTgt, firingWeapon)){
               if(!targetsOnMasterTargetList.contains(newTgt)){   // if target is not already on
                  masterTargetList.add(newTgt);                   // the target list add it
                  targetsOnMasterTargetList.addElement(newTgt);
               } // end if
               Target targetToShoot = (Target)masterTargetList.first();
               masterTargetList.remove(targetToShoot);
               targetsOnMasterTargetList.removeElement(targetToShoot);
               firingWeapon = selectFiringWeapon(targetToShoot, range);
               firingWeapon.shoot(targetToShoot, detectingSensor);
            } // end if
            else {
               checkForTrigger(detectingSensor, newTgt, firingWeapon);
            } // end else
         } // end if
         checkEnterSector(detectingSensor, newTgt, firingWeapon);
      } // end else
   } // end engageTarget()

   public void engageTarget(Sensor detectingSensor){
      if (masterTargetList.isEmpty()){                                 // if there are no
         return;                                                       // targets on the list
      } // end if
      Target nextTgt = (Target)masterTargetList.first();
      masterTargetList.remove(nextTgt);
      Coor3D wpnPos = (Coor3D)(getParent().getProperty("CurrentPos"));
```

```
double range = wpnPos.distTo(nextTgt.getCurrentPos());
      Weapon firingWeapon = selectFiringWeapon(nextTgt, range);

      if (wpnCtrl.inSector(nextTgt.getCurrentPos(), wpnPos, firingWeapon)){
         if (wpnCtrl.engageInSector(nextTgt, (Vehicle)getParent())){
            if (wpnCtrl.inRangeAndTrigger(nextTgt, firingWeapon)){
               targetsOnMasterTargetList.removeElement(nextTgt);
               firingWeapon.shoot(nextTgt, detectingSensor);
            } // end if
            else {
               masterTargetList.add(nextTgt);
               checkForTrigger(detectingSensor, nextTgt, firingWeapon);
            } // end else
         } // end if
      } // end if
      else {
         if (wpnCtrl.engageOutOfSector(nextTgt, (Vehicle)getParent())){
            if (wpnCtrl.inRangeAndTrigger(nextTgt, firingWeapon)){
               targetsOnMasterTargetList.removeElement(nextTgt);
               firingWeapon.shoot(nextTgt, detectingSensor);
            } // end if
            else {
               masterTargetList.add(nextTgt);
               checkForTrigger(detectingSensor, nextTgt, firingWeapon);
            } // end else
         } // end if
         checkEnterSector(detectingSensor, nextTgt, firingWeapon);
      } // end else
   } // end engageTarget()

   public Weapon selectFiringWeapon(Target target, double range){
      String targetClassification;
      String targetType;
      double maxPk = 0;
      int detectionStatus = target.getDetectionStatus();
      Vector weapons = ((Vehicle)getParent()).getWeapons();
      Weapon weaponOfChoice = (Weapon)weapons.firstElement();

      if (detectionStatus == 4){
         targetType = target.getTargetType();
         for (Enumeration enum = weapons.elements(); enum.hasMoreElements();){
            Weapon tempWeapon = (Weapon)enum.nextElement();
            String weaponName = tempWeapon.getWeaponName();
            double pk = KillTable.getPk(weaponName, targetType, "frontal", range);
            maxPk = Math.max(pk, maxPk);
            if (pk == maxPk) {
               weaponOfChoice = tempWeapon;
            } // end if
         } // end for
         return weaponOfChoice;
      } // end else if

      if (detectionStatus == 3){
         targetClassification = target.getClassification();
      } // end if
      else {
         targetClassification = "TANK";
      } // end else
      return weaponOfChoice;
   } // end selectFiringWeapon()


//==========================================================================================
// Utility Methods
//==========================================================================================

   public String toString() {return getName();}

   public double retrievePk(double[][] killMatrix, double targetRange){
      int xx = 0;
      double killRange = 0;
```

```
while(killMatrix[xx][1] != 0){
      killRange = killMatrix[xx][0];
      if (killRange >= targetRange){
         return killMatrix[xx][1];
      } // end if
      xx++;
   }// end while
   return killMatrix[xx - 1][1];
} // end retrievePk;

public void checkEnterSector(Sensor detectingSensor, Target tgt, Weapon firingWeapon){
   Coor3D firingVehPos = (Coor3D)getParent().getProperty("CurrentPos");
   Coor3D targetVehPos = tgt.getCurrentPos();
   Coor3D firingVehVel = (Coor3D)getParent().getProperty("Velocity");
   Coor3D targetVehVel = tgt.getTargetVelocity();
   double proxyTime = 0.0;
   double timeStep = .0083;
   double range = firingVehPos.distTo(targetVehPos);
   Coor3D tgtVehDeltaPos, snsVehDeltaPos, relativePos;
   while (range < detectingSensor.getSensorMaxRange()){
      if (firingVehVel.equals(new Coor3D(0, 0, 0)) &&
          targetVehVel.equals(new Coor3D(0, 0, 0))){
            return;
      } // end if
      if (wpnCtrl.inSector(targetVehPos, firingVehPos, firingWeapon)){
         generateEngageEvent("NewTarget", proxyTime, tgt, detectingSensor, firingWeapon);
         return;
      } // end if

      proxyTime += timeStep;                                        // increment time

      tgtVehDeltaPos = (Coor3D)targetVehVel.scalarMul(timeStep);    // determine how far veh
      snsVehDeltaPos = (Coor3D)firingVehVel.scalarMul(timeStep);    // can move in 1 timestep

      targetVehPos = (Coor3D)targetVehPos.add(tgtVehDeltaPos);      // update positions by
      firingVehPos = (Coor3D)firingVehPos.add(snsVehDeltaPos);      // their respective delta

      relativePos = (Coor3D)targetVehPos.sub(firingVehPos);         // determine new range
      range = relativePos.distTo(new Coor3D(0.0, 0.0, 0.0));
   } // end while
} // end checkEnterSector()

public void checkForTrigger(Sensor detectingSensor, Target tgt, Weapon firingWeapon){
   double criticalRange = Math.min(firingWeapon.getWeaponMaxRange(),
                          wpnCtrl.getTriggerLine());

   Coor3D firingVehPos = (Coor3D)getParent().getProperty("CurrentPos");
   Coor3D targetVehPos = tgt.getCurrentPos();
   Coor3D firingVehVel = (Coor3D)getParent().getProperty("Velocity");
   Coor3D targetVehVel = tgt.getTargetVelocity();

   double proxyTime = 0.0;
   double timeStep = .0083;
   double range = firingVehPos.distTo(targetVehPos);
   Coor3D tgtVehDeltaPos, snsVehDeltaPos, relativePos;

   while (range < detectingSensor.getSensorMaxRange()){
      if (firingVehVel.equals(new Coor3D(0, 0, 0)) &&         // if both vehicles are not
          targetVehVel.equals(new Coor3D(0, 0, 0))){          // moving no further events
            return;                                           // will occur
      } // end if
      if (range <= criticalRange){
         generateEngageEvent("NewTarget", proxyTime, tgt, detectingSensor, firingWeapon);
         return;
      } // end if

      proxyTime += timeStep;                                        // increment time
```

```
        tgtVehDeltaPos = (Coor3D)targetVehVel.scalarMul(timeStep);    // determine how far veh
        snsVehDeltaPos = (Coor3D)firingVehVel.scalarMul(timeStep);    // can move in 1 timestep

        targetVehPos = (Coor3D)targetVehPos.add(tgtVehDeltaPos);      // update positions by
        firingVehPos = (Coor3D)firingVehPos.add(snsVehDeltaPos);      // their respective delta

        relativePos = (Coor3D)targetVehPos.sub(firingVehPos);         // determine new range
        range = relativePos.distTo(new Coor3D(0.0, 0.0, 0.0));
      } // end while
    } // end checkForTrigger()
} // end class FireControl
```

APPENDIX B: ACRONYMS

- AAR – After Action Review
- ACDM – Accelerated Decision Making Process
- BBS – Brigade/ Battalion Battle Simulation
- BOS – Battlefield Operating System
- CALL – Center for Army Lessons Learned
- CBS – Corps Battle Simulation
- CMTC – Combat Maneuver Training Center
- COA – Course of Action
- CONUS – Continental United States
- CPX – Command Post Exercise
- CTC – Combat Training Center
- DTED – Digital Terrain Elevation Data
- DEC – Digital Equipment Corporation
- DES – Discrete Event Simulation
- GUI – Graphic users Interface
- JRTC – Joint Readiness Training Center
- Km – Kilometers
- MRB – Motorized Rifle Battalion
- MRC – Motorized Rifle Company
- MRR – Motorized Rifle Regiment
- NET – New Equipment Training
- NIMA – National Imagery and Mapping Agency
- NTC – National Training Center
- OC – Observer Controller
- OPFOR – Opposing Forces
- PC – Personal Computer

LIST OF REFERENCES

[1] U.S. Department of Defense, Department of the Army, *FM 100-5 Operations,* Government Printing Office, Washington, D. C. 1993.

[2] U.S. Department of Defense, Department of the Army, *FM 101-5 Staff Organization and Operations,* Government Printing Office, Washington, D. C. 1997.

[3] U.S. Army Command and General Staff College, *ST 100-9 The Command Estimate Process,* Government Printing Office, Washington, D. C. 1992.

[4] U.S. Army Command and General Staff College, *ST 101-5 Command and Staff Decision Processes* Government Printing Office, Washington, D. C. 1995.

[5] Long, Clyde L., *Synchronization of Combat Power at the Task Force Level: Defining a Planning Methodology*, Master's Thesis, U. S. Army Command and General Staff College, Fort Leavenworth, Kansas, 1989.

[6] National Training Center Brigade Combat Training Team, *Accelerated Tactical Decision Making Process,* Classroom training presented by the Brigade Combat Trainers as part of the NTC's pre-rotation training of visiting units, 1997.

[7] Titan Inc., *Janus 3.X/Unix Model User's Manual*, 1993.

[8] U.S. Army National Simulation Center, (Janus) "Information Paper." [http://www-leav.army.mil/nsc/famsim/janus/infopaper.htm], 3 Jan. 97

[9] U.S. Army National Simulation Center, "Available Janus Terrain." [http://www-leav.army.mil/nsc/famsim/janus/terrain.htm], 11 Feb. 99

[10] Defense Modeling and Simulation Office, "BBS – Brigade/Battalion Battle Simulation." [http://www.msrr.dmso.mil/msdocs/sof/BBS.htm], undated

[11] U.S. Army National Simulation Center, "Brigade / Battalion Battle Simulation Information Paper." [http://www-leav.army.mil/nsc/famsim/bbs/infopaper.htm], 11 Feb. 1998

[12] U.S. Army National Simulation Center, "BBS Terrain Information Paper." [http://www-leav.army.mil/nsc/famsim/bbs/terrain.htm], undated

[13] Defense Modeling and Simulation Office, "CBS – Corps Battle Simulation." [http://www.msrr.dmso.mil/msdocs/sof/CBS.htm], undated

[14] U.S. Army National Simulation Center, (CBS) "Information Paper." [http://www-leav.army.mil/nsc/famsim/cbs/infopaper.htm], 18 Sep. 98

[15] U.S. Army National Simulation Center, "Corps Battle Simulation Playboxes." [http://www-leav.army.mil/nsc/famsim/cbs/play.htm], undated

[16] Telephone conversation between MAJ Raymond Stienbart, Janus Team Chief at the National Simulation Center and the author, 12 Apr 1999.

[17] Telephone conversation between Mr. David Sargent, CBS Operations Research Analyst at the National Simulation Center and the author, 12 Apr 1999.

[18] U.S. Army Center for Lessons Learned, "Military Decision Making: Abbreviated Planning" [http://call.army.mil/call/newsltrs/95-12upd/table.htm], 1995

[19] U.S. Army Center for Lessons Learned, "National Training Center Trends 1st and 2nd Qtrs., FY 98" [http://call.army.mil/call/ctc_bull/98-14/intro.htm], 1998

[20] U.S. Army Center for Lessons Learned, "National Training Center Trends 3rdt and 4th Qtrs., FY 97" [http://call.army.mil/call/ctc_bull/98-4ntc/intro.htm], 1998

[21] U.S. Army Center for Lessons Learned, "National Training Center Trends Compendium 3QFY96 through 2QFY97" [http://call.army.mil/call/ctc_bull/97-17/intro.htm], 1997

[22] U.S. Army Center for Lessons Learned, "National Training Center Priority Trends 4QFY94 through 2QFY96" [http://call.army.mil/call/ctc_bull/ntc96pri/ntc96toc.htm], 1996

[23] U.S. Army Center for Lessons Learned, "JRTC Trends 4QFY97 & 1QFY98" [http://call.army.mil/call/ctc_bull/98-20/jrtctoc2.htm], 1998

[24] U.S. Army Center for Lessons Learned, "JRTC Trends Compendium 4QFY96 through 3QFY97" [http://call.army.mil/call/ctc_bull/98-7/table.htm], 1997

[25] U.S. Army Center for Lessons Learned, "Joint Readiness Training Center Priority Trends 4QFY94 through 3QFY96" [http://call.army.mil/call/ctc_bull/jrtc96pt/jr96pt.htm], 1996

[26] Szyperski, Clemens, *Component Software Beyond Object-Oriented Programming,* Addison Wesley Longman Limited, 1997

[27] Arntzen, A., *Software Components for Air Defense Planning,* Master's Thesis, Naval Postgraduate School, Monterey, California, September 1998.

[28] Schrepf, N, *Visual Planning Aid for Movement of Ground Forces in Operations Other Than War*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1999.

[29] Buss, A. H., *Simple Movement and Detection,* Class Notes, June 1998.

[30] Stork, K, *A Simulation Study of Countermeasure Effectiveness Against Anti-Ship Missles*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1996.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Capt. Steve Chapman, USN 1
   N6M
   2000 Navy Pentagon
   Room 4C445
   Washington, DC 20350-2000

4. George Phillips 1
   CNO, N6M1
   2000 Navy Pentagon
   Room 4C445
   Washingon, DC 20350-2000

5. Mike Macedonia 1
   Chief Scientist and Technical Director
   US Army STRICOM
   12350 Research Parkway
   Orlando, FL 32826-3276

6. National Simulation Center (NSC) 1
   ATTN:ATZL-NSC (Jerry Ham)
   410 Kearney Avenue --- Building 45
   Fort Leavenworth, KS 66027-1306

7. Michael Bailey 1
   Principal Analyst, Modeling and Simulation
   Marine Corps Combat Development Command (Code 56)
   3300 Russell Road
   Quantico, VA 22134

8. Dr. Michael Zyda, CodeCS/Zk 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

9. Dr. Arnold Buss, Code OR/Sb 1
   Operations Research Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

10. Senior Lecturer Bard Mansager, Code MA/Ma 1
   Mathematics Department
   Naval Postgraduate School
   Monterey, CA 93940-5000

11. William E. Bohman 1
   3155 Lookout Circle
   Cincinnati, OH 45208