NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

MODELING DYNAMIC TACTICAL BEHAVIORS IN COMBATXXI USING HIERARCHICAL TASK NETWORKS

by

Michael J. Donaldson

June 2014

Thesis Advisor: Imre Balogh
Second Reader: Kirk Stork

This thesis was performed at the MOVES Institute

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

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	3. REPORT TYPE AND DATES COVERED
	June 2014	Master's Thesis

4. TITLE AND SUBTITLE

MODELING DYNAMIC TACTICAL BEHAVIORS IN COMBATXXI USING HIERARCHICAL TASK NETWORKS

5. FUNDING NUMBERS

6. AUTHOR(S)

Michael J. Donaldson

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

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

N/A

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. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Defining accurate representations of group behaviors in simulations is an expensive, time-consuming task. One reason for this is that previously produced behaviors are often not reusable within other scenarios or simulations. Using Hierarchical Task Networks (HTNs) to model military behaviors is a promising technique for addressing this problem. HTNs provide a methodology for linking tactical behaviors, and offer a potential system for representing the military decision-making process at the tactical level. This thesis investigates the use of HTNs within the COMBATXXI model. COMBATXXI provides military planners a detailed representation of combat operations, and supports analysis efforts by providing insights into the effectiveness of weapon systems, unit organizations, and tactics. The use of HTNs within COMBATXXI is a relatively new concept; many aspects of HTN implementation have not been researched in depth. Work in this thesis involved development and testing of HTNs capable of executing a security formation behavior, and coordinating the execution of other ground combat related behaviors. The HTN-controlled behaviors were demonstrated in a simulated version of a United States Marine Corps live fire training range. The composable and dynamic aspects of these behaviors eased the scenario development process and added tactical realism to the test scenario.

14. SUBJECT TERMS

Affordance, Automated Behavior, Automated Planning, COMBATXXI, Composable Behavior, Dynamic Behavior, Hierarchical Task Network, HTN, Military, Model, Simulation, Tactical Behavior

15. NUMBER OF PAGES 209

16. PRICE CODE

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

NSN 7540-01-280-5500

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

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

MODELING DYNAMIC TACTICAL BEHAVIORS IN COMBATXXI USING HIERARCHICAL TASK NETWORKS

Michael J. Donaldson
Major, United States Marine Corps
B.S., United States Air Force Academy, 2003

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS, AND SIMULATION (MOVES)

from the

NAVAL POSTGRADUATE SCHOOL
June 2014

Author: Michael J. Donaldson

Approved by: Imre Balogh
Thesis Advisor

Kirk Stork
Second Reader

Christian Darken
Chair, MOVES Academic Committee

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Defining accurate representations of group behaviors in simulations is an expensive, time-consuming task. One reason for this is that previously produced behaviors are often not reusable within other scenarios or simulations. Using Hierarchical Task Networks (HTNs) to model military behaviors is a promising technique for addressing this problem. HTNs provide a methodology for linking tactical behaviors, and offer a potential system for representing the military decision-making process at the tactical level. This thesis investigates the use of HTNs within the COMBATXXI model. COMBATXXI provides military planners a detailed representation of combat operations, and supports analysis efforts by providing insights into the effectiveness of weapon systems, unit organizations, and tactics. The use of HTNs within COMBATXXI is a relatively new concept; many aspects of HTN implementation have not been researched in depth. Work in this thesis involved development and testing of HTNs capable of executing a security formation behavior, and coordinating the execution of other ground combat related behaviors. The HTN-controlled behaviors were demonstrated in a simulated version of a United States Marine Corps live fire training range. The composable and dynamic aspects of these behaviors eased the scenario development process and added tactical realism to the test scenario.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1 The Need for Dynamic Behaviors **1**
1.1 Military Doctrine 2
1.2 Composable Behaviors 4
1.3 Hierarchical Task Networks 5
1.4 Focus of Research 6
1.5 Research Questions 7
1.6 Benefits of Study 8

2 Automated Behavior History and Evolution **11**
2.1 Reactive Planning Techniques 11
2.2 Classical Planning Techniques 18
2.3 Hierarchical Task Network Planning 25

3 Dynamic Behavior in Current Applications **31**
3.1 Killzone 2 31
3.2 PlannedAssault 35
3.3 Command Ops 41
3.4 Tesla Tactics Language and Behavior Validation 50

4 Military Doctrine for Simulations **55**
4.1 Mission Command and Doctrinal Terminology 56
4.2 Offensive Operations Overview 58
4.3 Attack Overview 62
4.4 Case Study: Conduct of an Attack 64
4.5 Additional Considerations for Implementing Military Doctrine in Simulations 71

5 Challenges Inherent to Scripted Scenarios **75**
5.1 Range 400 Overview 75
5.2 Training Objectives and Performance Evaluation 78

5.3 Description of a Typical Range 400 Exercise Force 79
5.4 Building a Scripted Scenario of Range 400 81
5.5 Scenario Security Formations 82
5.6 Security Formation Creation Process 85
5.7 Advanced Considerations for Security Formations 88
5.8 Challenges to Overcome 90

6 Dynamic Behavior Development and Demonstrations 93
6.1 Scoping a Behavior to Develop 93
6.2 Security Formation Behavior. 94
6.3 Affordances 98
6.4 Urban Patrol Demonstration 99
6.5 Dynamic Range 400 Demonstration 103

7 Recommendations and Conclusions 111
7.1 Recommended Improvements to Thesis Products. 112
7.2 Expanding HTN Usage in COMBATXXI. 116
7.3 Steps Toward a Mission Planning Capability 118

Appendices

A Technical Discussion of Affordances 119
A.1 Functionality 119
A.2 Features 121

B Range 400 Example Attack 125

C Scenario Initialization Files 131
C.1 Range 400 jump_start.py 131
C.2 r400AffordanceInitialization.py 132
C.3 Urban Patrol jump_start.py 136
C.4 urbanPatrolAffordanceInitialization.py 137

D Affordance Files **145**
D.1 affordanceUtilities.java 145
D.2 changeAffordanceValidBoolean.xml 154
D.3 processAffordances.xml 155
D.4 stopProcessAffordancesHTN.xml 158
D.5 templateAffordanceInitialization.py 159

E Core Security Formation Behavior Files **161**
E.1 securityFormationLibrary.py 161
E.2 masterSecurityFormation.xml 164
E.3 unitSecurityFormation.xml 168
E.4 simpleUnitSecurityFormation.xml 170
E.5 basicIndividualMove.xml 171

F Modifiers to Security Formation Behavior **173**
F.1 removeUnitFromAffSecForm.xml 173
F.2 securityFormation_byPhaseAffordanceMod.xml 174
F.3 securityFormation_byProcessingUnit.xml 176

G Modifiers to SquadMove Behavior **179**
G.1 SquadMove_MoveDesignatedUnits.xml 179
G.2 SquadMove_OtherUnitsInArrayList.xml 179

References **181**

Initial Distribution List **185**

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

Figure 2.1 Basic Finite State Machine for a Non-Player Character, from [10] 12

Figure 2.2 Finite State Machine Complexity, from [11] 13

Figure 2.3 Common Embedded Behavior Approach, from [20] 14

Figure 2.4 Using Embedded Information to Find Ambush Locations, from [18] 15

Figure 2.5 BT Composite Tasks, from [10] 17

Figure 2.6 BT Execution Order, from [9] 17

Figure 2.7 A* Algorithm for Navigation and Planning, from [11] 19

Figure 2.8 Comparison of Behaviors in No One Lives Forever 2 and First Encounter Assault Recon, from [11] 22

Figure 2.9 Example Actions used in First Encounter Assault Recon, from [11] 23

Figure 2.10 Example of Hierarchical Task Network Task Decomposition, from [25] 26

Figure 3.1 AI Architecture, from [29] 33

Figure 3.2 Squad AI, from [29] 33

Figure 3.3 Killzone 2 Task Organization Algorithm, from [29] 33

Figure 3.4 Advanced Embedded Behavior Considerations, from [20] 34

Figure 3.5 Killzone 2 Manual Annotations, from [30] 35

Figure 3.6 AI Techniques Considered During PlannedAssault Development, from [31] 36

Figure 3.7 Planner Methods, after [31] 38

Figure 3.8 Task Types, States and Relations, from [31] 38

Figure 3.9 PlannedAssault Unit Selection, from [32] 39

Figure 3.10 PlannedAssault Mission Selection, from [32] 40

Figure 3.11 PlannedAssault Example Scheme of Maneuver 41

Figure 3.12 Command Ops Screenshot, from [33] 42

Figure 3.13 Command Ops Unit Hierarchy, from [33] 44

Figure 3.14 Example Orders and Associated Parameters, from [33] 45

Figure 3.15 Orders Traffic During Gameplay, from [33] 46

Figure 3.16 Validation Task Information Flow, from [37] 51

Figure 3.17 Tesla Tactical Template, from [40] 53

Figure 4.1 Four Types of Offensive Operations, from [41] 58

Figure 4.2 Movement to Contact Task Organization, from [41] 59

Figure 4.3 Frontal Attack, Flanking Attack, and Penetration Examples, from [41] 61

Figure 4.4 Tactical Control Measures for a Basic Attack, from [44] 66

Figure 5.1 Notional Enemy Forces on R400, from [47] 76

Figure 5.2 Range 400 Company Commander Brief Sheet, from [47] 77

Figure 5.3 Range 400 Training & Readiness Tasks 1, from [47] 78

Figure 5.4 Range 400 Training & Readiness Tasks 2, from [47] 79

Figure 5.5 Pre-Operation Security T&R Task, from [46] 84

Figure 5.6 Post-Operation Security T&R Task, from [46] 84

Figure 5.7 Waypoints Needed for a Scripted Security Formation 87

Figure 6.1 Current Hierarchy of Security Formation HTNs 95

Figure 6.2 Three Examples of Security Formations 98

Figure 6.3 Patrol Route and Affordances 100

Figure 6.4	Removing a Unit from a Security Formation (From Left to Right)	100
Figure 6.5	Bounding Past a Stationary Squad	101
Figure 6.6	Reinforcing an Existing Security Formation	101
Figure 6.7	Using Adapter HTNs to Modify the Functionality of Core HTNs	103
Figure 6.8	Comparison of Required Behavior Mechanisms for Scripted and Dynamic Scenarios	104
Figure 6.9	Unit Security Formation - Scripted (Left) and Dynamic (Right) .	105
Figure 6.10	Guardian Angel Security Formation - Scripted (Left) and Dynamic (Right)	105
Figure 6.11	Small Unit 360 Security Formation - Scripted (Left) and Dynamic (Right)	106
Figure 6.12	180 Security Formation (No Casualties) - Scripted (Left) and Dynamic (Right)	107
Figure 6.13	180 Security Formation (With Casualties) - Scripted (Left) and Dynamic (Right)	108
Figure 7.1	Proposed Hierarchy of Security Formation Mechanisms	114
Figure 7.2	Analysis of Subunit Capabilities	115
Figure A.1	Depiction of the processAffordances HTN Structure	122

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

A* A-star

ACE air combat element

ACR advanced concepts and requirements

ADSO Australian Defence Simulation Office

AI artificial intelligence

BFTB Battles from the Bulge

BT behavior tree

CE command element

CNA Center for Naval Analysis

COA course of action

COMBATXXI Combined Arms Analysis Tool for the 21st Century

COTA Conquest of the Aegean

DOD Department of Defense

DS direct support

EINSTein Enhanced ISAAC Neural Simulation Tool

FAC forward air controller

FCL final coordination line

F.E.A.R. First Encounter Assault Recon

FiST fire support team

FLOT forward line of troops

FM	Field Manual
FO	forward observer
FPS	first person shooter
FSM	finite state machine
GCE	ground combat element
GOAP	goal-oriented action planning
GS	general support
GUI	graphical user interface
GCV	ground combat vehicle
HBR	human behavior representation
HMMWV	high mobility multipurpose wheeled vehicle
HTN	hierarchical task network
IA	immediate action
ISAAC	Irreducible Semi-Autonomous Adaptive Combat
IV	inter-visibility
LAV	light armored vehicle
LCE	logistics combat element
LD	line of departure
LOA	limit of advance
LOS	line of sight
M&S	modeling and simulation

MAGTF	Marine Air Ground Task Force
MCAGCC	Marine Corps Air Ground Combat Center
MCCDC	Marine Corps Combat Development Command
MCDP	Marine Corps Doctrinal Publication
MCRP	Marine Corps Reference Publication
MCWP	Marine Corps Warfighting Publication
MIM	minimal idea model
MOUT	military operations in urban terrain
MOVES	Modeling, Virtual Environments, and Simulation
MSM	minimal system model
NOLF2	No One Lives Forever 2
NPC	non-player character
NPS	Naval Postgraduate School
OAD	Operations Analysis Division
OODA	observe, orient, decide, act
PDDL	Planning Domain Definition Language
PL	phase line
RCPA	relative combat power analysis
RDA	research, development, and acquisition
RO	radio operator
RTS	real time strategy

R400	Range 400
R401	Range 401
S&T	Science & Technology
SAM-K	suppress, assess, move, kill
SAT	systems approach to training
SE	systems engineering
SHOP	Simple Hierarchical Ordered Planner
SHOP2	Simple Hierarchical Ordered Planning 2
SITS	Scenario Integration Tool Suite
SMAW	shoulder-launched multipurpose assault weapon
SME	subject matter expert
SSM	synthetic system model
STO	science and technology objective
STRIPS	Stanford Research Institute Problem Solver
T&R	Training and Readiness
TCM	tactical control measure
TRAC-WSMR	TRADOC Analysis Center - White Sands Missile Range
TTECG	Tactical Training Exercise Control Group
TTP	tactics, techniques, and procedure
U.S.	United States
USA	United States Army

USAF United States Air Force

USMC United States Marine Corps

USN United States Navy

V&V verification & validation

VBS2 Virtual Battlespace 2

WWII World War II

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

First and foremost I must thank my wonderful wife Kristen and our amazing son Cole for supporting me during this long process. You both are everything to me, and I could not have done this without you.

The rest of my family also deserves a great deal of credit. Thank you to my parents, Jim and Signe, for instilling in me the importance of education. I truly appreciate the efforts that my parents and brother Corey put into correcting my occasional tendency to butcher the English language! Thank you to my brother Spencer for spending your time deployed while I "relaxed" in Monterey. Your devotion to your men served as a source of motivation, and reminded me of this work's true purpose - supporting the warfighter.

Last but not least, I must thank my amazing thesis team. I cannot show enough appreciation for the impact that Dr. Imre Balogh had on my time at the Naval Postgraduate School. You taught me to open my mind to new topics, helped me link this work to my infantry background, and ultimately enabled me to enjoy the thesis process. To Imre, Kirk, and everyone else who devoted time to shaping this thesis, I will forever be in your debt. I sincerely hope that the information in this work proves useful in advancing the field of modeling and simulation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
The Need for Dynamic Behaviors

The rapid advancement of technology has greatly increased the capabilities of modeling and simulation (M&S) technologies. Understanding the potential of M&S is made difficult because of the large number and varying types of models in existence. In 1996, Dr. Joan Roughgarden described a three-tiered classification system for models. The first tier involved a minimal idea model (MIM) in which the details of a real-world system were replaced by approximations. The second tier consisted of a minimal system model (MSM) that contained elements which accurately replicated some aspects of the real world system. The final tier involved a synthetic system model (SSM) which attempted to replicate nearly all physical aspects of a real world system [1, pp. 35-36].

Numerous MIMs and MSMs have proven useful for providing important knowledge of real-world systems. A major advantage of these models is that they are relatively simple to develop in comparison to SSMs. If constructed properly, MIMs and MSMs can provide insights about the system being modeled. However, abstracting too many real life details can result in limitations on the amount of useful information a model can provide. Additionally, this can prevent a model from being reusable in other scenarios and contexts.

For one to properly implement actual military behaviors into simulations, it is necessary to consider the type of operations being conducted, and the level of detail provided by the simulation. Only then is it possible to decide which military behaviors must be modeled explicitly (in a detailed manner) and which behaviors can be modeled implicitly (abstracted out of the model). As technology has progressed, simulations have gained the ability to represent increasingly accurate versions of reality. To avoid problems caused by the excessive use of implicit modeling, organizations have sought to employ simulations that model the real world at a detailed level. Combined Arms Analysis Tool for the 21st Century (COMBATXXI) is one model that the military has built for such use.

COMBATXXI is a "joint, high-resolution, closed-form, stochastic, discrete event, entity level structure analytical simulation" [2]. It is currently used by the United States

Army (USA) and United States Marine Corps (USMC) in the "research, development, and acquisition (RDA) and advanced concepts and requirements (ACR) M&S domains" [2]. COMBATXXI has the ability to model numerous aspects of combat operations, including ground forces, fixed-wing and rotary-wing aircraft, combat service support units, and future forces. Detailed terrain maps provide data for simulated military operations in all types of theaters, including amphibious environments for USMC forces. COMBATXXI provides military planners a detailed representation of combat operations. This information supports the analysis of combat operations by providing insights into the effectiveness of weapon systems, unit organizations, and tactics.

Unfortunately, this knowledge does not come without a cost. Developing the inputs for running a simulation in COMBATXXI requires a significant investment in time and money. Building the input scenario for a recent Army study of a new ground combat vehicle (GCV) took nine trained analysts approximately seven months to complete [3]. Even an entry level tutorial focused on a mechanized infantry platoon requires an estimated twelve hours to program.

One of the reasons for such a long development time is that high-resolution simulations require entity behaviors to be programmed in a highly detailed manner. Additionally, these behaviors are often not reusable in other scenarios, or even by other entities in the same simulation. There is no simple solution to such problems; obtaining high-resolution insights from a computer simulation will always require a great deal of work. Creating mechanisms to ease this process first requires a basic understanding of military operations.

1.1 Military Doctrine

The United States Armed Forces differ from civilian businesses due to their manner of their funding, their large number of personnel, and the chaotic nature of their intended operating environment – warfare. Training a workforce in a consistent and effective manner is a serious issue for organizations that have a large numbers of employees with a high personnel turnover rate. To accomplish this task in an efficient manner, the Army and the Marine Corps adopted a set of procedures from the systems engineering (SE) field that are known as the systems approach to training (SAT) [4].

According to the SAT manual, "the SAT was created to manage the instructional process

for analyzing, designing, developing, implementing, and evaluating instruction. ... It is a dynamic, flexible system for developing and implementing effective and efficient instruction to meet current and projected needs" [4, p. iii]. Originally intended as a method of instruction for formal schools, the SAT process was applied to training and operations throughout the Marine Corps.

A Marine Air Ground Task Force (MAGTF) is divided into four main categories: the ground combat element (GCE), the air combat element (ACE), the logistics combat element (LCE), and the command element (CE). Each MAGTF branch has a series of SAT documents called Training and Readiness (T&R) manuals. These documents provide a detailed explanation of operational tasks in which Marines are required to maintain proficiency. Additionally, T&R manuals specify tasks for individuals and units at all levels, and arrange these tasks in a hierarchical manner. Whether one is a regimental commander, or a machine gun team leader, the T&R manuals contain detailed descriptions, instructions, and performance standards for the tasks that one's unit must be able to accomplish.

Literature concerning military actions is not limited to the USMC T&R manuals. There exists a great number of military publications that specify how units should execute tactical actions in scenarios ranging from counterinsurgency operations, to full scale high intensity conflict in urban or amphibious settings. Examples of doctrinal military publications pertinent to this thesis are listed below:[1]

1. Marine Corps Doctrinal Publication (MCDP) 1 "Warfighting"
2. MCDP 1-0 "Operations"
3. Marine Corps Warfighting Publication (MCWP) 3-11.2 "Marine Rifle Squad"
4. MCWP 5-1 "Marine Corps Planning Process"

While no written document can perfectly illustrate the dynamics of actual combat, the above listed publications provide detailed descriptions of the basic behaviors that military units are expected to perform. Potentially of even more importance, the listed publications provide insights into how military members are trained to think. Understanding the principles and the actions in these publications, as well as in the USMC T&R manuals, can

[1] Similar information can be found in United States (U.S.) Army Field Manual (FM) 3-0 "Operations" and FM 3-21.8 "The Infantry Rifle Platoon and Squad."

play a vital role in programming agent behaviors that are representative of actual military maneuvers.

1.2 Composable Behaviors

Defining accurate representations of group behaviors in simulations is an expensive, time-consuming task. One of the main reasons for this is that previously produced behaviors are often not reusable within other scenarios or other simulations. The concept of composable behaviors has the potential to create "an easy to use, adaptive and flexible framework for simulating group behaviors" [5].

Reusable components help systems users by enabling them to leverage previous work for the purpose of easing future work. There are many ways to define reusability; the two meanings most relevant to this thesis involve template based reuse and direct reuse. Template based reuse is where previously developed components are used as guides to build new components. The use of templates can substantially reduce the effort required to build a new component; however, a new component still needs to be created. Direct reuse is when a component can be directly utilized in a scenario without the need to create a new version. To support a range of contexts, this approach uses parameters that allow the component to adjust to different contexts. The goal of this work is to support direct reuse.

Different meanings are also attributed to the term composability. Addressing the general notion of composability is a complex problem and well beyond the scope of this thesis; therefore, this research will refer to composability as the "capability to select and assemble simulation components in various combinations into valid simulation systems to satisfy specific user requirements" [6, p. 1]. Applied to the field of automated behaviors, composability refers to the process by which primitive behaviors are combined to form high-level composite behaviors. These composite behaviors are then executed by simulation entities. Within military simulations, the creation of such a capability promises to increase the realism of automated behaviors and to ease the scenario development process.

According to Mikel Petty, "the defining characteristic of composability is the ability to combine and recombine components into different simulation systems for different purposes" [6, p. 1]. The application of a composable behavior system with this degree of flexibility has numerous benefits to military simulations. However, achieving this goal first

requires a solid foundation, particularly in the creation of low-level supporting behaviors. As stated by Joshua Summers, "a primitive behavior is an action that is not decomposable into sub-actions and that is used, combined with other primitive behaviors, to develop composite behaviors" [7, p. 2]. The creation of a composable behavior system requires such primitive behaviors to be identified early in development processes, and coded in a manner that ensures maximum reusability.

An important goal of this thesis is to apply the concept of composable behaviors to military operations. It is important to recognize that military tasks are hierarchical in nature. Simple military behaviors, such as fire and movement, play vital roles in complex military behaviors such as flanking attacks and movements to contact. Additionally, many of the fundamental actions and mindsets of small military forces in one environment remain applicable to the operations of a larger force in a different environment. These factors create a direct tie to the concept of composable behaviors, which can be described informally, as building complex behaviors from simple ones in a systematic manner.

1.3 Hierarchical Task Networks

To be able to represent complex operations such as mechanized warfare or military operations in urban terrain (MOUT), it is first necessary to model simple behaviors that form the foundation for such complex maneuvers. One method to accomplish this is through the use of automated planning techniques known as hierarchical task networks (HTNs). In general, HTNs can be described as "a formalism for representing hierarchical plans" in which high-level tasks are decomposed into simpler tasks [8, p. 2]. This method can enable the accomplishment of high-level strategies through the coordinated actions of individual entities [8, p. 2]. Using the HTN methodology to model fundamental military behaviors is a promising technique for applying the concept of composable behaviors to military simulations. HTNs not only provide a method for linking tactical behaviors, they also offer a potential technique for representing the military decision-making process at the tactical level.[2]

Military leaders devise solutions to problems in one of three ways: they select a previously

[2] HTNs are appropriate for representing the human decision making process in models like COMBATXXI because this model represents the net effect of human decision making and not the actual process.

learned technique, they modify a previously learned technique, or they devise a unique solution. Simulating the decision-making process of military leaders is a complex task but, if feasible, would be of great value to simulations. Modeling military behaviors with HTNs provides a potential way to represent many aspects of this process at the tactical level. Their ability to do so lies in the fact that HTNs contain numerous paths for reaching a single goal state. The selection of which path to utilize depends upon an agent's current situation and the HTN parameters. This methodology, in combination with the concept of composable behaviors, holds the potential to simulate military actions in a more realistic manner.

Applied properly, agents executing HTNs of simple behaviors would choose the course of action best suited to their current situation. Because composable behaviors are being utilized, the actions caused by HTNs of simple behaviors would directly impact the automated decision-making conducted for complex tasks (also controlled by HTNs). The result of this would be the creation of a non-scripted action that is uniquely tailored to the given situation.

This concept can be illustrated by an example of a military unit conducting a standard attack on an enemy position. When conducting an attack, military members will often conduct fire and movement in order to close the distance between themselves and their enemy. If one element of the unit were to begin receiving heavy volumes of enemy fire, that specific element could automatically change its behavior from fire and movement, to attack by fire. This action would place additional amounts of suppression on the enemy force, thus allowing adjacent friendly elements to successfully close to within hand grenade range of the enemy force. This is an example of how the correct application of simple behaviors (fire and movement, attack by fire) could result in the emergence of more complex military behaviors (fix and flank). This concept could be achieved by modeling basic military behaviors with HTNs, and by linking them together in a composable manner. Should work in this thesis prove successful, it could lead to more realistic modeling of complex military behaviors.

1.4 Focus of Research

Research in this thesis is divided into three main phases. The first phase involves the research of techniques utilized to created automated behavior in simulations and games.

This section investigates the evolution of techniques for creating automated behavior that have been used in the military, in the gaming industry, and in academia. It also highlights notable applications of these techniques which should be of interest to the Department of Defense (DOD).

Additionally, this phase discusses the topic of behavior validation. As M&S techniques have progressed, a heavy emphasis has been placed on formalizing the approaches utilized to certify systems. The validation of automated human behavior is a vital, yet underdeveloped field. This section discusses the history of human behavior representation (HBR) validation, deficiencies of current procedures, and newly proposed methods of validation. If successfully implemented, these techniques promise to revolutionize M&S usage across numerous communities.

The second phase of this research involves the study of military doctrine and tactics. The creation of artificial intelligence (AI) that is able to adequately represent human behavior first requires an understanding of the principles and thought processes upon which actual military behaviors are based. Human behaviors in simulations are not adequately represented when entities are scripted to mimic the actions of a military force. Therefore, this phase discusses several important principles of military operations, and illustrates their use in a variety of scenarios. An additional purpose of this phase is to scope the implementation portion of this thesis. While the concept of a basic infantry attack may initially appear to be simple, this fundamental military action includes numerous sub-tasks and decision points. By identifying and analyzing these complexities, this section provides guidance for the creation of an AI system that can execute a basic infantry attack.

The final phase of this thesis documents efforts to create dynamic tactical behaviors in COMBATXXI. The majority of these efforts involve the use of a specialized type of HTN. While differences exist between this HTN implementation and other more traditional HTNs, both forms retain many of the same basic properties. For the purpose of this thesis, COMBATXXI HTNs will simply be referred to as HTNs.

1.5 Research Questions

The use of HTNs within COMBATXXI is a relatively new concept; many aspects of HTN implementation, including reusability and scalability, have not been formally researched.

Work in this thesis will involve construction of HTNs for infantry behaviors, validation of these HTNs in standard scenarios, and reusability and scalability testing of these HTNs. With respect to the modeled behaviors, this section will seek to answer the following questions:

Representation: Do HTNs increase a simulation's ability to represent combat scenarios?
Efficiency: Does the use of HTNs create efficiencies in scenario construction?
Composability: Can HTNs of tactical behaviors operate in conjunction with other HTNs?
Reusability: Can HTNs be successfully utilized in different scenarios and contexts?
Scalability: Are HTNs able to control the actions of various sized units?
Parameters: What parameters are appropriate for HTNs of military behaviors?
Creation: Can doctrinal military publications serve as useful guides for creating HTNs of tactical behaviors?

1.6 Benefits of Study

Military history is rich with examples of armies that did not adapt to changing circumstances, and were defeated because they attempted to fight conflicts in the same manner as they had fought in previous wars. Differences in environments, in unit organizations, and in weapon systems, can all have a significant effect on how wars are conducted. To remain relevant, simulations that are utilized to prepare military forces must be mindful of these changes. One of the ways to do this is to identify, understand, and accurately represent the fundamental behaviors that are seen on the field of battle.

The products of this thesis will satisfy requests for HTNs of infantry behaviors that can be utilized in COMBATXXI scenarios conducted by the Operations Analysis Division (OAD), Marine Corps Combat Development Command (MCCDC). Because the HTNs will model fundamental infantry behaviors and will be based upon doctrinal military publications, these HTNs will provide additional realism in a wide range of scenarios.

Motivating factors for this thesis also include several long term goals. This thesis is intended to be part of the foundational research for a new method of behavior representation that could revolutionize the manner in which COMBATXXI scenarios are created. Additional benefits of this thesis include the following:

- Examine the feasibility of using dynamic HTN-controlled behaviors to model military operations.
- Add relevant and realistic tactical actions to the growing libraries of HTN-based behaviors.
- Demonstrate HTN reusability and scalability.
- Identify parameters that are needed to model tactical behaviors with HTNs.
- Establish a link between doctrinal military behaviors and COMBATXXI behaviors.

This thesis has direct ties to several science and technology objectives (STOs) described in the "2012 U.S. Marine Corps Science & Technology (S&T) Strategic Plan." The first area in which this thesis would benefit the Marine Corps is in the acquisition of combat systems. Numerous STOs in the maneuver and fires categories describe the need to develop new technologies. These include but are not limited to MVR STO-3: Advanced robotic systems in support of ground maneuver; MVR STO-5: Vehicle and surface craft design for Marine usability, habitability, and survivability; and finally Fires STO-4: Increased capabilities and reduced weight of all ground combat weapons systems. Progress in each of these STOs will require simulations similar to the GCV study previously mentioned in this thesis. The creation of more realistic behaviors in COMBATXXI could benefit such studies by reducing wasteful spending and by helping to ensure the selection of the best combat system.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2: Automated Behavior History and Evolution

Constructing a scenario within simulations such as COMBATXXI requires the scenario developer to specify detailed behaviors for every entity within the simulation. Such simulations present several advantages and disadvantages. The detail-oriented nature of the simulation allows programmers to specify the exact behaviors of their agents. This attribute gives simulations the potential to accurately represent military operations. However, achieving representative behavior is not necessarily an easy task. First, the scenario developer is responsible for constructing every part of a military operation; this information includes, but is not limited to, force structures, equipment, actions, and event coordination. This is a process that not only takes a great deal of time, but also one that requires the programmer to have an intimate knowledge of the military operation to be modeled. If models are produced without an adequate amount of development time, or by scenario designers who lack knowledge of military operations, then the produced scenario is at risk of not being able to represent the operation to a sufficient degree of accuracy.

An important goal for this thesis is to ease the development of simulation scenarios by automating the creation of certain behaviors. Several techniques offer potential solutions to this problem. The following section provides a summary of AI techniques that have been utilized to create tactical behaviors in the military and gaming domains. While the purposes of military simulations and video games are different, the tactical behaviors they contain have many characteristics in common. The development environment within both domains also have similar time constraints; being able to develop good behaviors quickly is an important factor for success.

2.1 Reactive Planning Techniques

Reactive planning methods include finite state machines (FSMs) and behavior trees (BTs). Within these systems, non-player characters (NPCs) follow a "pre-programmed strategy that specifies how the NPC should react in the game depending on the current state/node and conditions that currently hold in the game-world" [9, p. 30]. The relative simplicity of these techniques has resulted in their widespread use in the gaming industry.

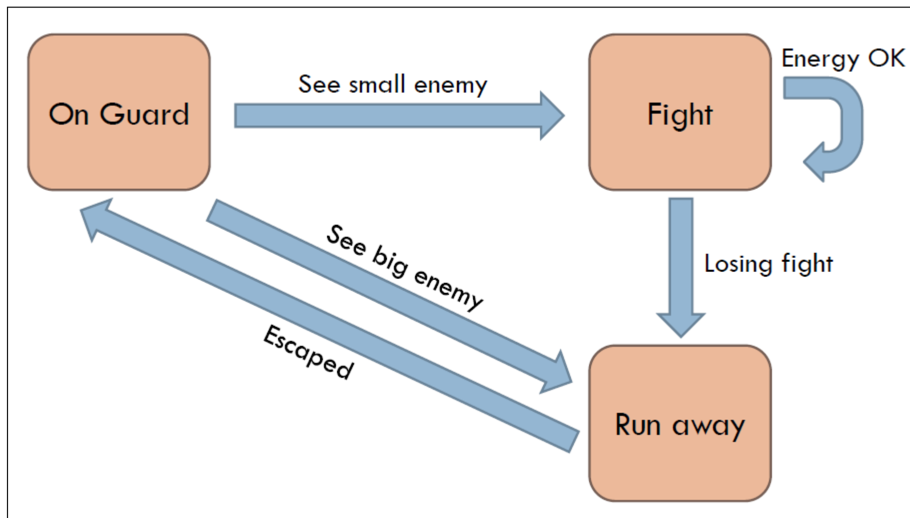


Figure 2.1: Basic Finite State Machine for a Non-Player Character, from [10]

2.1.1 Finite State Machines

One of the traditional ways for encoding agent artificial intelligence is through the use of FSMs. This structure is mainly comprised of a collection of states and a series of transitions between those states. FSMs have been successfully employed in many video games and simulations that called for agents to carry out a simple set of behaviors. Video game programmer Jeff Orkin describes FSMs as structures which tell "AI exactly how to behave in every situation" [11, p. 3].

This attribute presents both advantages and disadvantages. For software developers, FSMs are relatively easy to understand and implement. They can often be implemented by utilizing basic if-then-else statements [10, p. 9]. As the demand for more realistic behaviors increased, the limitations of FSMs became apparent. If used to represent anything but the simplest of behaviors, FSMs become extremely complex. This complexity makes modifying FSMs a tedious task. Figure 2.1 and Figure 2.2 depict how quickly an FSM based system can become unmanageable.

2.1.2 Embedded Behaviors and Affordances

The affordance concept is used to tie physical characteristics of some object or situation to a set of actions made possible by the presence of the object or situation. This idea was

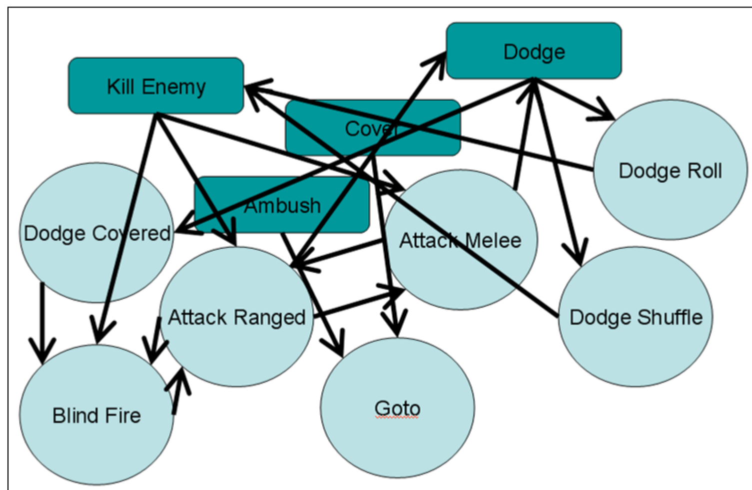


Figure 2.2: Finite State Machine Complexity, from [11]

introduced by James Gibson as part of his work in psychology [12]. Additional information on affordances can be found in the following sources:

- *An Activity Theory Approach to Affordance* by Baerentsen and Trettvik [13]
- *Gibsonian Affordances for Roboticists* by Chemero and Turvey [14]
- *An Affordance-Based Perspective on Human-Machine Interface Design* by Lintern [15]
- *The MACS Project: An Approach to Affordance-Inspired Robot Control* by Rome et al. [16]

The affordance concept has been refined and used in a wide range of disciplines from industrial design to AI. In the gaming industry, this approach has been used to help agents make better context dependent choices. AI programmer Sergio Garces describes how embedding relevant information into terrain is a viable technique for enabling agents to identify context that triggers an appropriate set of possible behaviors:

> Try to get someone to explain to you how to pick a good cover point or to find an advantageous defensive position, and they'll have a hard time, but they'll quickly give several examples when they see a map. It's so effective that frequently the easiest way to get AI to 'understand' a map is to spell everything

> out manually.[3] With proper tools, it shouldn't be a problem for the designers to annotate [their] maps with the information that the AI needs to do its reasoning: cover points, camping spots, possible attack avenues towards a coveted resource, areas that are interesting to defend, etc... Sometimes, instead of placing this information in the map directly, [it can be encoded] in the objects that will go in the map. [17, p. 3]

According to Lars Liden, "waypoints can be exploited to efficiently calculate information about the strategic value of particular world locations" [18, p. 211]. Within this thesis, embedded behaviors will be referred to as affordances. For many years, the gaming realm has utilized this approach to control NPCs through the use of scripts and hints. As stated by Straatman and Beij, "the scripts define how the AI responds to specific situations, whereas the hints indicate where the AI could perform specific actions" [19, p. 2]. Figure 2.3 provides an example of how embedded scripts can be used to trigger specific NPC actions, and Figure 2.4 illustrates how hints embedded into nodes can be used to refine NPC ambush behaviors.

While useful for many applications, affordances run into limitations as NPCs are expected to respond to more dynamic situations. According to Straatman and Beij, "the need to fight anywhere, against threats from any direction cannot be supported using this approach without placing loads of hints. Statically placed hints are less effective when positions of

[3] In this context, the term AI refers to an agent executing a behavior. The academic and gaming realms sometimes use the term in this manner.

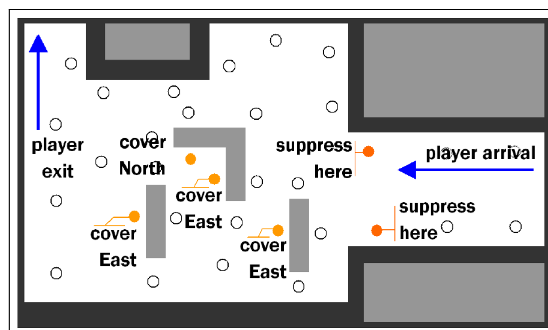


Figure 2.3: Common Embedded Behavior Approach, from [20]

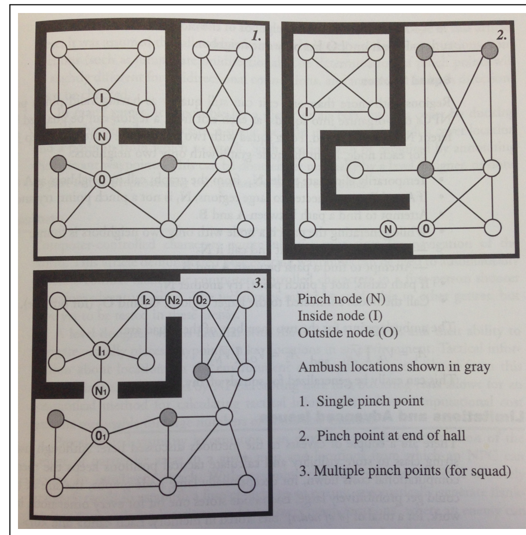


Figure 2.4: Using Embedded Information to Find Ambush Locations, from [18]

the opposing forces are hard to predict, or when the environment is likely to change" [19, p. 2]. Despite such issues, affordances remain relevant tools for behavior control. Embedded hints and scripted tactics can still be useful in "exceptional situations where the default tactics won't work or the default behavior does not provide the intended behavior" [19, p. 2].

It should be noted that the previously described affordances contain relatively low level information. This aspect generally increases the number of affordances required to achieve desired behaviors. An alternate technique would be to program affordances with higher level information that is relevant in a broader context. This approach would mitigate the need to create large numbers of affordances.

2.1.3 Behavior Trees

Behavior trees are a more recently developed technique for producing realistic actions within simulations. If created in coordination with a domain expert, behavior trees have the potential to accurately represent many aspects of the human decision making process. One of the first major games to successfully implement behavior trees was Halo 2 in 2004. Since then, they have been implemented in numerous additional games.

Behavior trees are largely composed of two different task types: leaf tasks and composite tasks. Leaf tasks "check a condition or execute some code" [10, p. 14]. The actions that leaf tasks perform are normally some type of script. Composite tasks are composed of leaf tasks or subordinate composite tasks; their return value depends upon the execution of their child tasks [10, p. 14]. Though rare, it is possible for composite tasks to execute actions of their own [9, p. 10].

Several options exist for creating behavior trees capable of accomplishing interesting behaviors. Figure 2.5 illustrates one such behavior tree which varies the success requirement for composite tasks. Composite tasks denoted by an arrow require all child tasks to be successful; composite tasks denoted by a question mark only require one child task to be successful. As depicted in Figure 2.6, the order that child tasks are attempted is normally from left to right. However the behavior can be made non-deterministic by using a random process to select which child task is executed first. Composite tasks can also be written so that they attempt to perform multiple child tasks at once.

Behavior trees provide the advantage of being simple to understand and to implement. This is part of the reason they are extremely popular within the gaming industry. Professor Stavros Vassos states that behavior trees provide "separation between the work of the programmer and the game designers," and that new methods are constantly being developed to give them more capability [10, p. 29].

Unfortunately, this technique has some inherent disadvantages. First, behavior trees become very large if they represent anything but the simplest human decision making process. Second, the complexity of large behavior trees makes modifying their structure late in the development process a tedious process. These limitations in behavior trees have led some researchers to seek alternate solutions for producing automated behavior.

2.1.4 Emergence of an Alternate Automated Behavior Technique

The 2002 video game No One Lives Forever 2 (NOLF2) was an example of a game in which the designers sought to implement a different method of behavior control. To achieve more realistic behaviors with its agents, NOLF2 would assign goals to characters; the goals were assigned in such a manner as to be applicable to the agent's current situation.

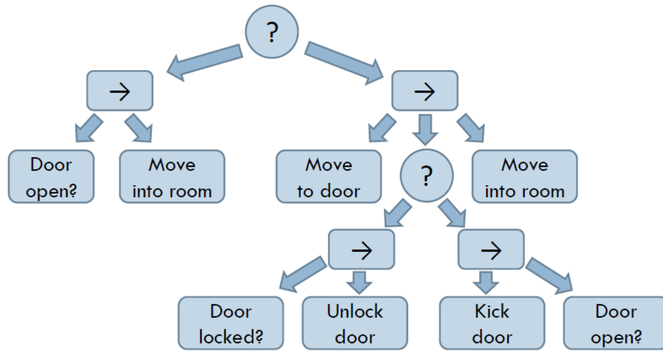


Figure 2.5: BT Composite Tasks, from [10]

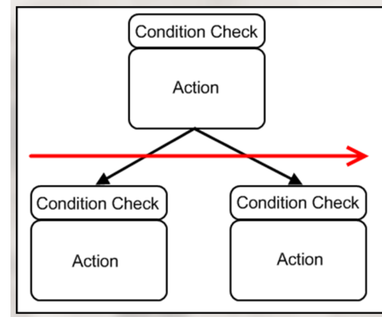


Figure 2.6: BT Execution Order, from [9]

Each goal was tied to an FSM which contained actions that were suitable to various possible states of the agent [21, p. 11]. In an effort to make the agent's behavior as realistic as possible, these plans typically contained conditional branches that were used to select the action most suitable to the situation [22]. Which branch was selected was dependent upon the agent's state and the preconditions for each FSM node. While this approach produced unique behavior properties for NOLF2 agents, limitations were still encountered.

First, "a problem with the hard coded FSM in NOLF2 is that goals cannot communicate with each other" [21, p. 11]. This lack of communication resulted in the appearance of awkward behaviors when agents transitioned from one goal to another. NOLF2 also exhibited many of the previously described problems attributed to FSMs.

When creating variations in agent behaviors, branches had to be added to existing state machines. According to one NOLF2 programmer, after "two years of development, these state machines became overly complex, bloated, unmanageable, and a risk to the stability of the project" [11, p. 7]. This meant that behavior limitations had to be accepted in order to prevent the FSMs from growing too complex. Additionally, it was risky to alter the FSMs late in the development cycle. Despite these issues, NOLF2 represented an important step in a new direction. The work done on this game led AI designers to the realization that separating an agent's goal from the plan that was used to accomplish the goal would offer a revolutionary new way for creating automated behaviors.

2.2 Classical Planning Techniques

As demand for more realistic automated behaviors increased, problems with legacy programming techniques became more apparent. Planning techniques such as the Stanford Research Institute Problem Solver (STRIPS), goal-oriented action planning (GOAP), and HTN planning, offer a different approach to producing automated behaviors. Some of these planning techniques have made a substantial impact in areas such as the robotics industry. However, it is not until recently that they began to play a part in the simulation and gaming realms.

2.2.1 Stanford Research Institute Problem Solver

The STRIPS concept was introduced by Fikes and Nilsson in their 1971 landmark work, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." As described by AI programmer Jeff Orkin, STRIPS "consists of goals and actions, where goals describe some desired state of the world we want to reach, and actions are defined in terms of preconditions and effects" [11, p. 4]. The STRIPS algorithm includes several key concepts. First, the AI decision making takes place in planning space as opposed to state space. This means that a plan is constructed before any actions are taken. This is different from FSMs and behavior trees which make decisions during real time. Second, a STRIPS planning problem consists of three main components: initial state, goal, and available actions [23, p. 18]. These three inputs are used as factors in a search which determines the ideal action for an agent to perform. This differs from behavior trees for which no search is required; "the behavior tree is traversed as a kind of pre-defined program" [10, p. 22].

2.2.2 Goal-Oriented Action Planning Overview

One of the first viable alternatives to FSM and goal tree based techniques was introduced in the 2004 game First Encounter Assault Recon (F.E.A.R.). The success of F.E.A.R. was largely due to its use of a modified form of STRIPS planning known as GOAP. GOAP is a dynamic behavior planning technique that was devised to overcome many of the shortcomings of FSMs and behavior trees. The purpose of GOAP is to create agents which are able to execute actions that are appropriate to their current state. The major difference between GOAP and reactive planning methods is that GOAP goals are not linked to a pre-established plan. According to Jeff Orkin, "a planning system tells the AI what his goals

and actions are, and lets the AI decide how to sequence actions to satisfy goals. FSMs are procedural, while planning is declarative" [11, p. 3]. Through the use of such an automated planner, unique actions are produced each time an agent is assigned a goal.

An unorthodox use of the A-star (A*) search method is the first key to the creation of an automated planner. A* is a technique which is normally associated with determining a navigation path, however, it is actually a general purpose graph traversal algorithm which can be utilized for many purposes. As stated by Jeff Orkin, "in the case of planning, the nodes are states of the world, and we are searching to find a path to the goal state. The edges connecting different states of the world are the actions that lead the state of the world to change from one to another" [11, p. 11]. Typical GOAP-type games will utilize A* for both planning and navigational purposes. Figure 2.7 illustrates the inputs for an A* program when it is used for different purposes. This is a technique that has the potential to produce realistic plans which are uniquely tailored to an agent's specific circumstances.

2.2.3 Case Study: First Encounter Assault Recon

As the first game to implement the GOAP approach, F.E.A.R. represented a revolution in the creation of automated tactical behaviors. In the following quote, programmer Jeff Orkin describes how the use of GOAP provided significant advantages to game developers.

> Late in development of NOLF2, we added the requirement that AI would turn on lights whenever entering a dark room. In our old system, this required us to revisit the state machine inside every goal, and figure out how to insert this behavior. This was both a headache, and a risky thing to do so late in

A*	Navigation	Planning
Nodes:	NavMesh Polys	World States
Edges:	NavMesh Poly Edges	Actions

Figure 2.7: A* Algorithm for Navigation and Planning, from [11]

> development. With the F.E.A.R. planning system, adding this behavior would have been much easier, as we could have just added a TurnOnLights actions with a LightsOn effect, and added a LightsOn precondition to the Goto action. [11, p. 10]

First Encounter Assault Recon Overview

The success of F.E.A.R. was achieved through hard work, and a unique perspective on many traditional gaming techniques. The first significant departure from most gaming systems can be found in the F.E.A.R. FSM which includes only three states: Goto, Animate, and UseSmartObject. GoTo involves an agent moving to a new location, Animate describes an action that an agent takes (such as shooting), and UseSmartObject is a specialized version of Animate [11, p. 2]. All AI actions in F.E.A.R. can be categorized using these three states. For example, "an AI going for cover is just moving to some position, and then playing a duck or lean animation" [11, p. 2].

The difficulty in this system arises when determining when an agent should transition from one state to another. In traditional reactive planning techniques, this transition has either been scripted, or written into the FSM states. In F.E.A.R. this transition logic was implemented into the planning system.

The planning system connects to the FSM by designating a chain of actions to accomplish, and then by activating these actions at appropriate times. Jeff Orkin states that "when we execute our plan, we sequentially activate our actions, which in turn set the current state, and any associated parameters. The code inside the ActivateAction() function sets the AI into some state, and sets some parameters. For example, the Flee action sets the AI into the Goto state, and sets some specific destination to run to." [11, pp. 12-13].

While the F.E.A.R. planning system is based on STRIPS, several key adjustments were necessary in order to ensure the system was suitable for a gaming environment. Major differences included adding a cost per action, eliminating Add and Delete Lists for effects, and adding procedural preconditions and effects [11, p. 2]. These adjustments were made for several reasons.

- Assigning costs to actions enables the A* based planner to search for "the lowest cost

sequence of actions [that] satisfy some goal" [11, p. 11].

- Replacing Add and Delete Lists with a fixed-sized array representing the world state "makes it trivial to find the action that will have an effect that satisfies some goal or precondition" [11, pp. 11-12].
- Procedural Preconditions were added in order to optimize system functionality. Jeff Orkin states that "it would be impractical to always keep track of whether an escape path exists in our world state, because pathfinding is expensive. The Procedural Preconditions function allows us to do checks like this on-demand only when necessary" [11, p. 12].
- Procedural Effects are important for linking changes in the world state to the actions that are carried out. "We don't want to simply directly apply the effects that we've represented as world state variables, because that would indicate that changes are instantaneous. In reality it takes some amount of time to move to a destination, or eliminate a threat" [11, p. 12]. This is important because while an action is being carried out in a dynamic environment, the state of the world can change for reasons having nothing to do with the procedure being executed.

Goal-Oriented Action Planning Benefits

This planning system's first major contribution was the separation of goals and actions. Unlike in previous systems, F.E.A.R. developers could make changes to the system when they were well into the development process. One example of this included adding a new character to the game. The modular nature of the system enabled developers to combine previously created goals and actions, and ultimately create a "unique new enemy type in a minimal amount of time, without imposing any risk on existing characters" [11, p. 8].

According to Jeff Orkin, "decoupling the goals and actions forces them to share information through some external working space. In a decoupled system, all goals and actions have access to information... We can take this knowledge into account when we formulate a plan to satisfy" a specific goal [11, p. 8]. This information enabled the system to devise a more intelligent plan. It also prevented the appearance of awkward behaviors observed in previous systems, when agents transitioned from one goal to another. [11, p. 8]. Figure 2.8 depicts the organization of goals and behaviors in F.E.A.R. This organization is in stark contrast to NOLF2 which linked goals to FSMs.

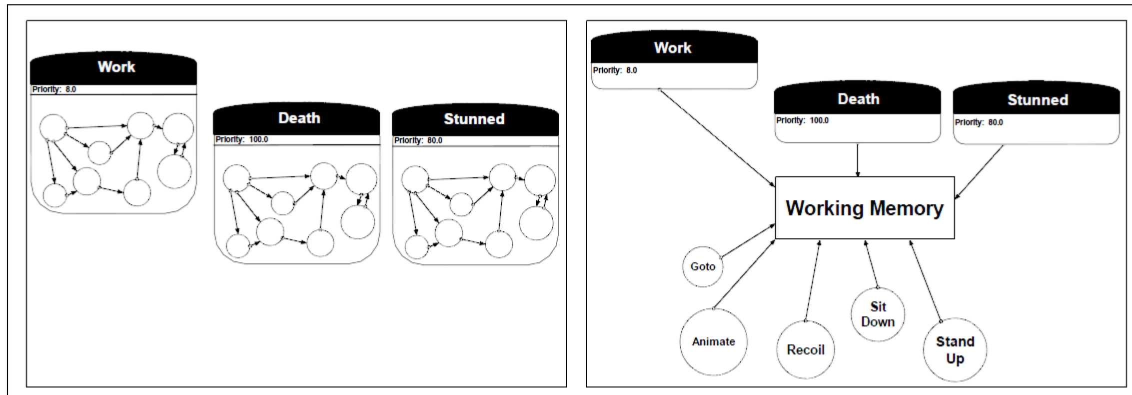


Figure 2.8: Comparison of Behaviors in No One Lives Forever 2 and First Encounter Assault Recon, from [11]

“The second benefit of the planning approach is facilitating the layering of behaviors” [11, p. 8]. This attribute enabled developers to create interesting agent behaviors which ranged from simple movement techniques to more complex tactical procedures. The real benefit of this ability was that actions could be added without adversely affecting other behaviors. The quote at the beginning of this section which discussed adding a "TurnOnLights" action is an example of this. Figure 2.9 lists some of the behaviors assigned to various characters in F.E.A.R. The planning system could choose from these behaviors when attempting to accomplish a specific goal.

The final benefit of planning has to do with the emergence of dynamic behavior. A common axiom among military professionals is that the best laid plans are often nullified as soon as enemy contact is made. This saying is not only applicable to real life, but also to simulations and games. When an agent recognizes that it has failed to execute a task, the agent’s current plan becomes irrelevant, and the agent uses the planning system to determine a new course of action. According to Jeff Orkin, “dynamic behavior arises out of re-planning while taking into account knowledge gained through previous failures... As the AI discovers obstacles that invalidate his plan... he can record this knowledge in working memory, and take it into consideration when re-planning to find alternative solutions to” his goal [11, p. 10].

These attributes were instrumental in allowing individual entities within F.E.A.R. to carry

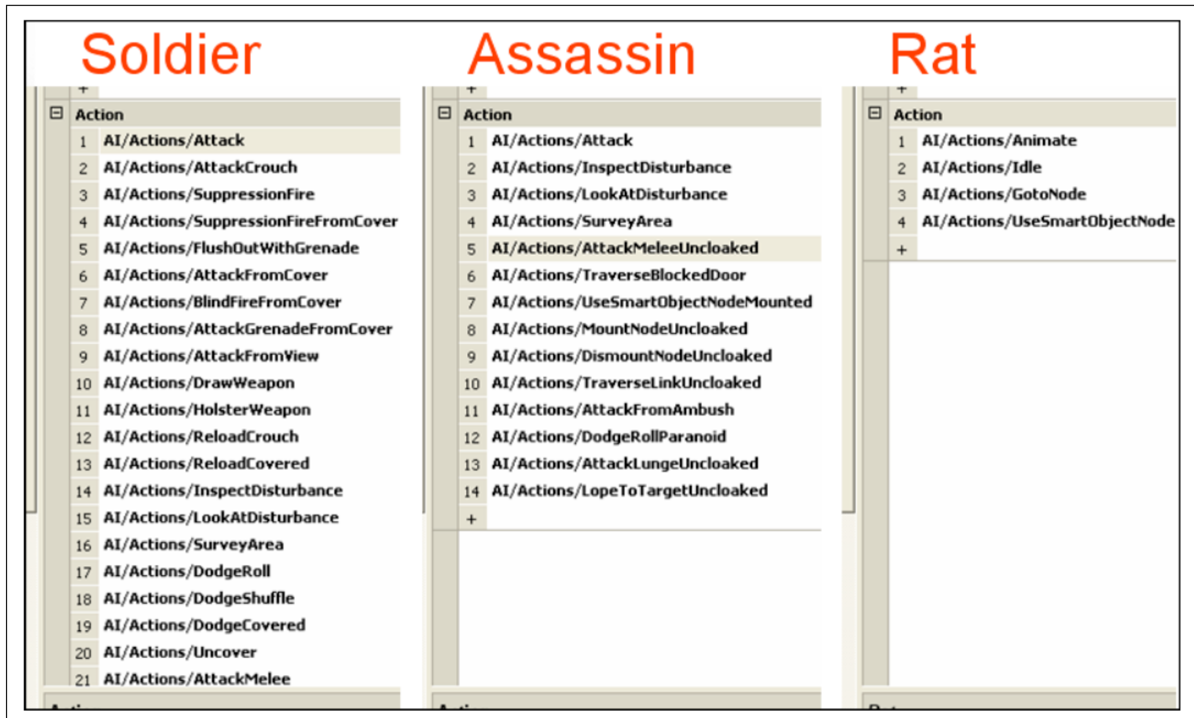


Figure 2.9: Example Actions used in First Encounter Assault Recon, from [11]

out realistic actions which suited their current situation. In the following quote, Jeff Orkin describes how the game structure limited some of the problems associated with reactive planning techniques, and allowed developers to focus their efforts on other areas:

> The design philosophy at Monolith is that the designer's job is to create interesting spaces for combat, packed with opportunities for the AI to exploit. For example, spaces filled with furniture for cover, glass windows to dive through, and multiple entries for flanking... This means that the AI need to autonomously use the environment to satisfy their goals. [11, p. 6]

Squad Behaviors

The last innovation included in F.E.A.R. was an ability for automated agents to coordinate their actions in a realistic fashion. By periodically grouping nearby agents into a squad, the system gained the ability to implement one of four squad behaviors [11, p. 13]:

Get-to-Cover: gets all squad members who are not currently in valid cover into valid cover, while one squad member lays suppression fire

Advance-Cover: moves members of a squad to valid cover closer to the threat, while one squad member lays suppression fire

Orderly-Advance: moves a squad to some position in a single file line, where each AI covers the one in front, and the last AI faces backward to cover from behind

Search: splits the squad into pairs who cover each other as they systematically search rooms in some area

The mechanism for creating the squad behavior was a non-STRIPS system that was separate from the planner used to construct individual behaviors [11, p. 15]. This process was accomplished in the following manner:

> First, the squad behavior tries to find AI that can fill required slots. If it finds participants, the squad behavior activates and sends orders to squad members. AI have goals to respond to orders, and it is up to the AI to prioritize following those orders versus satisfying other goals. For example, fleeing from danger may trump following an order to advance. The behavior then monitors the progress of the AI each clock tick. [11, p. 13]

The combination of the squad and individual behavior control mechanisms produced a unique type of automated tactical behavior in F.E.A.R. One of the key reasons for this is an individual agent's ability to override a squad order that does not make sense due to the agent's current state. Though such a decision may cause the squad behavior to fail, it often resulted in the emergence of more complex tactical behaviors which software developers never programmed into the system. Examples of these behaviors show strong similarities to military tactics such as fix and flank maneuvers, double envelopments, and withdrawals under pressure. According to game designers, the primary reason for this emergent behavior was due to the separation of the individual and squad level planning systems [11, p. 15].

The use of GOAP in F.E.A.R. served to illustrate a revolutionary new way of controlling automated behaviors. Unfortunately, this method appears to be limited in capabilities. The

emergent behaviors described above mostly included very small numbers of automated characters. One of the reasons for this is that for GOAP "to formulate a plan, the planner must be able to represent the state of the world in a compact and concise form" [22]. Through much hard work, the developers of F.E.A.R. were able to accomplish such a task. The use of an FSM with only three states played a vital role in their success. However, large numbers of agents and inherent complexity of simulations limit the ability to expand upon the techniques used in F.E.A.R. This has led to the emergence of a new type of planning system.

2.3 Hierarchical Task Network Planning

The topic of automated planning is an important area of research within the computer science realm. Much work is devoted to producing newer and more optimal planning models. Numerous methods exist, each with their own advantages and disadvantages. One such solution, HTN planning, is of interest to this work because it is a type of automated planning that is still evolving. In the following statement, Jeff Orkin describes how HTN planning could advance the capabilities demonstrated in F.E.A.R.

> A logical next step from what we did for F.E.A.R. would be to apply a formalized planning system to the squad behaviors. A developer interested in planning for squads may want to look at HTN planning, which facilitates planning actions that occur in parallel better than STRIPS planning. Planning for a squad of AI will require planning multiple actions that run in parallel. HTN planning techniques have been successfully applied to the coordination of Unreal Tournament bots. [11, p. 15]

2.3.1 Hierarchical Task Network Overview

First introduced as "procedural nets" in 1975 by Sacerdoti, HTN planning has been applied to many domains because of advantages it possesses in efficiency, scalability, and reusability [24]. According to Imre Balogh, "the idea behind the HTN methodology is to have a way to represent many overlapping plans within a single structure that can be used to generate a specific plan based on the state of the world at the time the plan is generated" [25].

As stated by Dana Nau, “the objective of an HTN planner is to produce a sequence of actions that perform some activity or task [26, p. 380].” Figure 2.10 depicts an example HTN behavior. HTN tasks are generally grouped into the following three categories [25]:

1. Goal Tasks – tasks that, when completed, correspond to the goal of the network/tree
2. Compound Tasks – tasks that need to be decomposed into sets of simpler tasks
3. Primitive Tasks – tasks that require no further decomposition to be performed and can be directly executed

Task decomposition is the primary method through which HTNs function; creating an HTN is generally accomplished through the following decomposition process. The first step is to identify a high level task to be accomplished, and to define a goal state which represents the system state upon completion of the high level task. The second step involves problem decomposition in which one must identify sub-tasks that are necessary to reach the goal state. The final step involves determining whether each sub-task represents a primitive task, or a complex task. This process is repeated until all complex tasks are reduced to primitive tasks.

One of the key advantages of HTNs is that they can contain numerous paths for reaching a single goal state. Path selection is accomplished through two key processes. First, the ordering of the nodes controls the priority given to different paths (if more than one is possible). Second, constraint nodes require preconditions to be met before a path is undertaken; this serves to provide a mechanism through which the context affects the plan. These important HTN attributes enable an agent facing a decision point to “select” the solution that

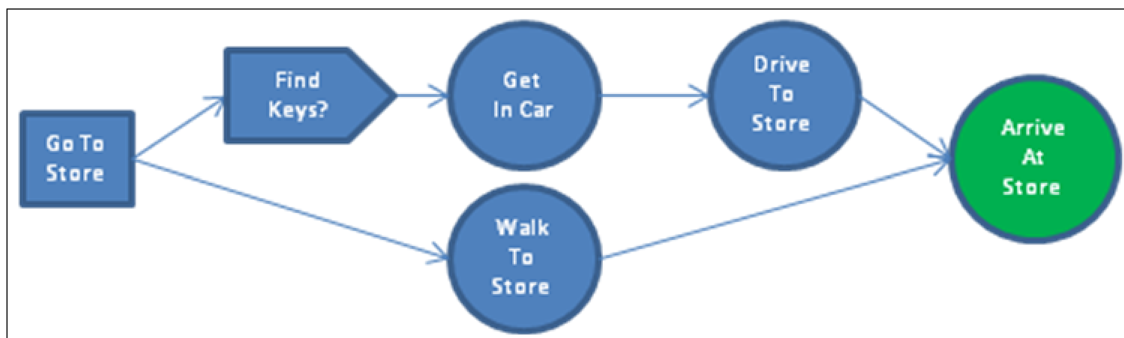


Figure 2.10: Example of Hierarchical Task Network Task Decomposition, from [25]

is most appropriate to its current situation. In conjunction with properties including efficiency, reusability, and scalability, this example of automated planning establishes HTNs as a tool which could greatly improve the effectiveness of many programs, including military simulations.

2.3.2 Simple Hierarchical Ordered Planner

These attributes made HTNs extremely popular for many planning problems, and within domains such as production-line scheduling, crisis management, logistics, equipment configuration, manufacturability analysis, and evacuation planning [26, p. 395]. Many advances in HTN usage have been made by Professor Dana Nau and his colleagues at the University of Maryland. They are the creators of the award winning Simple Hierarchical Ordered Planner (SHOP).

SHOP includes a family of HTN planners which earned acclaim for their ability to generate strategies in the game Bridge. The performance of many planning systems are often negatively impacted by large search spaces. SHOP offers a solution to this problem. For the Bridge problem, the original SHOP HTN planner conducted a search to determine a player's best possible move. However, instead of initially considering every potential move, the HTN first determined a general Bridge strategy to pursue. [27]. This process of designating general strategies, as opposed to designating specific actions, had the effect of creating smaller search spaces than those seen in conventional planners such as STRIPS.

This illustrates a major difference between HTN planning and behavior trees. HTN planners specify the world state that corresponds to a goal, simulate numerous plans for reaching that world state, and then speculate which plan is best. This final plan is executed by the player. This sequence differs from behavior trees which perform actions in real time, and apply changes directly to the world state [27]. HTN planning also possesses many distinctions from other planning techniques. One of the problems with some conventional planners is that when creating plans, they may not link actions together in the correct order [27]. Because subject matter experts construct hierarchical plans which accurately represent actions within their domain, HTN planning can minimize this problem.

The success of SHOP encouraged Professor Nau and his associates to continue their work with HTNs. Simple Hierarchical Ordered Planning 2 (SHOP2) is a domain-independent

program that contains many improvements over the original SHOP, and allows users to utilize HTNs in new ways [26, p. 379]. The first improvement concerns the way that tasks are ordered. The order in which potential tasks are specified is also the order in which the planner will consider each task. While advantageous from the perspective of allowing the programmer to tailor the planner's decision making, not all situations should have their potential tasks considered in the same order. SHOP2 offers "a convenient way for the author of a planning domain to tell SHOP2 which parts of the search space to explore first" [26, p. 380]. The SHOP2 method of "partial ordering" enables the planner to interleave subtasks from different tasks, and ultimately "makes it possible to specify domain knowledge in a more intuitive manner than was possible with SHOP" [26, p. 380]. Finally, SHOP2 incorporates several features such as quantifiers and conditional effects from the Planning Domain Definition Language (PDDL) which was created in an effort to standardize AI techniques.

One additional program of note is Pyhop. Developed by Dr. Nau and his associates, Pyhop is an HTN planner that is written in the Python language. While its planning algorithm bares many similarities to that of SHOP, several differences exist. One such difference is that HTN operators and methods are represented by Python functions [28]. As open-source software that is relatively easy to understand and implement, Pyhop provides an excellent way to learn more about HTN functionality.

2.3.3 Planning in Dynamic Environments

As the capabilities of HTNs expanded, researchers began applying HTNs to new domains. The first person shooter (FPS) game Unreal Tournament showcased one of the first uses of HTNs in the gaming industry. In this example, HTN planning was used to create an overall strategy for AI controlled teams, and to assign specific tasks to individual entities within those teams. By retaining the event-driven nature of individual entities, this technique allowed entities to "react to the [dynamic] events in the environment while contributing to [their team's] grand strategy" [8, p. 1]. While this effort provided numerous advantages over previous AI techniques, the integration of HTNs into video games was not without difficulties.

Despite its success in many domains, HTN pioneer Dana Nau was surprised with the use

of HTN planning systems in the gaming realm [27]. One of the major reasons for this is that HTNs were designed to be run and employed in a static environment. This presents a problem in the gaming and simulation domains where constantly changing environments can invalidate an agent's plan for achieving a stated goal. In the Unreal Tournament example, event handlers were "used to detect relevant events that [might] require interrupting the current action being executed to select a new one. For example, while performing the exploring action, the Bot may interrupt the explore action and start a hunting action if it detects an enemy in the surrounding area" [8, p. 2]. Another way this issue can be dealt with is by repeating the HTN planning process, or replanning, at a specified time interval. This technique has been used in gaming applications where there exist a limited number of entities. However, because the planning process is potentially expensive, this technique will not work for many types of simulations which contain hundreds or even thousands of entities.

One current method for minimizing this problem involves placing a mechanism within the HTN that automatically initiates replanning. This is accomplished in two ways. First, selected tasks within an HTN are designated as replanning nodes. Second, primitive HTN tasks which fail in execution are designed to trigger a replanning event [25, p. 4]. The advantage of this approach is that it places "the burden of deciding when replanning should occur on the person creating the HTN" [25, p. 3].

Another method of increasing the efficiency of HTN planners is through "lazy expansion." This is a technique which halts the execution of an HTN planner before it creates a complete plan capable of achieving some goal state. Elements of the robotics industry accomplish this by employing "rolling horizons" within planning algorithms [27]. Another way to employ lazy expansion of HTNs is through the addition of sub-goals known as interrupt goal nodes. These mechanisms are used to "represent tasks that take some time to complete... [and] to limit how far ahead task related events are scheduled" [25, p. 4]. This technique is uniquely suited to military simulations and games in which conflict between opposing forces will almost always force the modification of battle plans.

Each of these techniques provides software developers valuable tools for controlling the behavior of automated forces. Methods for triggering replanning are likely to increase realism of an entity's behavior. However, unnecessary replanning events can adversely

impact system performance, particularly for those simulations with a large number of automated forces. Similarly, utilizing the lazy expansion concept for HTNs holds the promise of increasing system performance, but could affect a plan's suitability. In situations where it is important to ensure that optimal plans are selected, HTN planners may need to look further into the future than interrupt nodes or rolling horizons allow [27]. Choosing the best technique to employ depends heavily upon the characteristics of the intended simulation or game.

CHAPTER 3:
Dynamic Behavior in Current Applications

Chapter 2 discusses the history of how automated behavior techniques developed. It provides important examples of successful and unsuccessful efforts to create automated human behavior. Understanding these examples can help simulation developers create applications which are able to adequately represent desired behaviors.

At the time of this writing, the following examples represent some of the most advanced examples of automated behavior techniques within the gaming industry. These applications include an FPS game, a military mission planning program, a real time strategy (RTS) game, and a military behavior composition language. While the appearances of these systems are drastically different, they all possess powerful AI capabilities. FPS and RTS systems have been described as working toward the same goal, but from different directions [29, p. 75]. For those persons interested in advancing current combat simulation capabilities, the techniques and capabilities of these systems should be of great interest.

3.1 Killzone 2

Killzone 2 is an FPS application that was created by Guerilla Games and released in 2009. Killzone 2 was highly acclaimed upon its release; one of the reasons for this was the realistic behavior of its automated entities. In most FPS games, automated characters are expected to exhibit good reactive behaviors, but are not usually charged with the accomplishment of long term objectives. As demand for realistic automated behaviors increased, many individuals in the gaming industry found this technique to produce inadequate results. The success of Killzone 2 can be largely attributed to the use of an HTN planner that created realistic battle plans for squads of automated characters.

3.1.1 Hierarchical Task Network Planning

Killzone 2 was one of the first games to use a complete HTN implementation. Utilizing one of the main strengths of HTNs, Killzone 2 employs “a goal-driven approach by separating what to do and how to do it” [29, p. 30]. The game’s HTN planner exhibited the following important characteristics [28, p. 12]:

- Utilized a special-purpose HTN planner for planning at the squad level
- Utilized method and operator syntax similar to SHOP and SHOP2
- Quickly generated linear plans that were appropriate to static environments
- Replanned several times per second to account for dynamic game environment
- Created actions that appeared believable and consistent to human users; did not attempt to create an ideal plan

An important aspect to consider is that of replanning. A plan is designated to have failed if a current task fails. It is desirable to "abort [a] current plan preemptively when a better plan [becomes] available, [or when the] current plan is no longer feasible" [29, p. 15]. The task of knowing in advance when a plan is going to fail is one of the difficulties with HTNs. In Killzone 2, this problem was avoided by replanning five times per second. While this technique worked in Killzone 2, this is not always possible in larger simulations, especially those involving hundreds or thousands of entities. Solving this issue is a major hurdle for those researchers interested in advancing the use of HTNs.

3.1.2 Command Structure

The first item of note was the creation of a hierarchical AI structure. As depicted in Figure 3.1, Killzone 2 categorizes its AI capabilities into Strategy AI, Squad AI, and Individual AI. Within this construct, the Strategy AI gives orders to a squad, and then the squad provides feedback on whether or not the order succeeded. Squads give orders to individuals, and individuals provide combat information as feedback to the squad. One item not depicted in Figure 3.1 is that individuals have the capability to communicate with the Strategy AI [29, p. 9].

This command structure takes advantage of HTN capabilities, one of which is assigning missions to specific units or entities. This is depicted in Figure 3.2. Standardized missions within Killzone 2 included "search and destroy, capture and hold, search and retrieve, assassination, and body count" [29, p. 31]. These missions are of interest because of their reusable and scalable attributes. Within Killzone 2, it was important to be able to assign these missions to units of different composition, and still be able to depend upon the unit to act in an intelligent manner. Another important aspect of the HTN planner was the ability to assign missions to units that were best suited to that specific task.

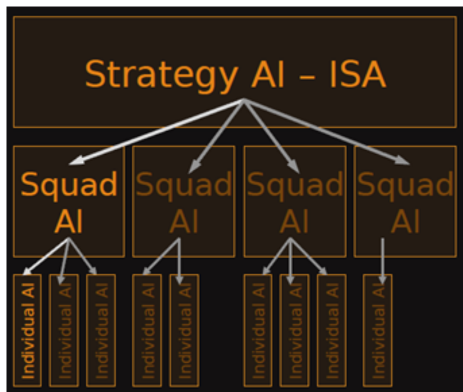


Figure 3.1: AI Architecture, from [29]

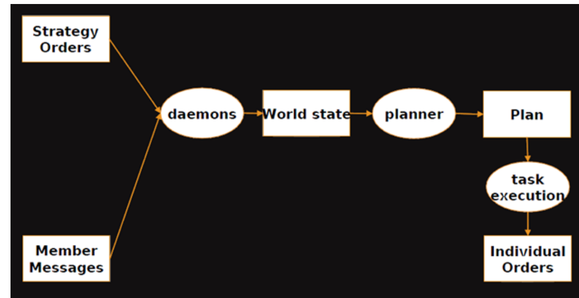


Figure 3.2: Squad AI, from [29]

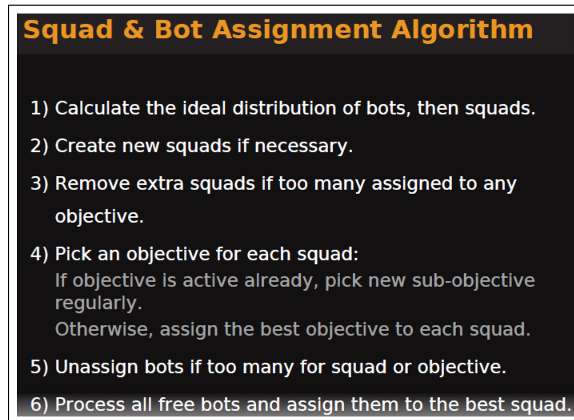


Figure 3.3: Killzone 2 Task Organization Algorithm, from [29]

One prerequisite to this capability was the need to create tactical units that were suitable for fulfilling various objectives and sub-objectives. Heuristics involved in this process included assigning entities to squads based on proximity to objectives, trying to respect the desired squad size parameters, and keeping squads balanced [29, p. 39]. This process was conducted according to the algorithm depicted in Figure 3.3.

3.1.3 Additional Behavior Techniques

Within Killzone 2, several more traditional AI techniques also fulfill important roles. Killzone 2 refers to the affordances described in Chapter 2 as annotations, and uses them to create areas of the map that will influence NPC dynamic tactical reasoning. As previously

mentioned, when advanced behaviors are desired, “AI scripts and their ‘if-then’ constructs are not well suited to interpret and balance the dozens of inputs necessary to tailor a tactical maneuver to the situation and terrain at hand” [19]. Killzone 2 takes a different approach to the use of annotations; some of the considerations made to produce more dynamic behaviors are shown in Figure 3.4. Typical uses for these annotations include helping NPCs decide where to hide, regroup, defend, or stage an attack. Figure 3.5 depicts annotations within a Killzone 2 map. Items of particular interest in this scene include the sniper positions. These annotations were placed to indicate “an approximate area where it would be good for a sniper to operate” [29]. However the annotations do not micromanage the entities. Instead, they give “snipers rough strategic guidance while low level AI determines a sniper’s detailed behaviors” [29].

Pathfinding is one final technique which was instrumental in achieving Killzone 2’s automated behavior. A tool with roots that can be traced back to an A* search, pathfinding was an important method for making “medium-term strategic decisions in space” [29, p. 62]. This was accomplished through the use of both low-level pathfinders and high-level “strategic” pathfinders. Additional advanced capabilites in Killzone 2 enabled important features that would prevent multiple squads from crowding into the same location, and would also choose routes or objectives that were suitable to a squad’s characteristics and capabilities.

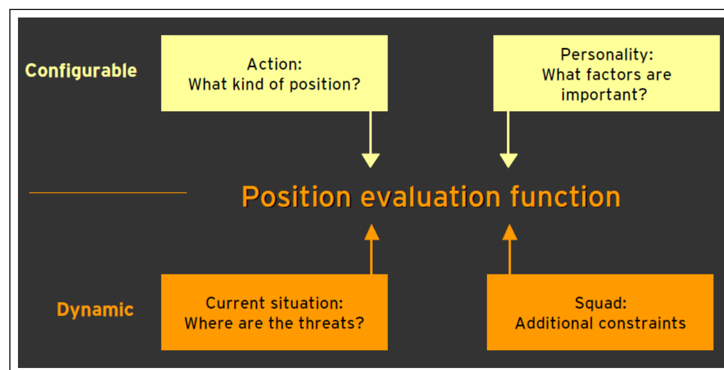


Figure 3.4: Advanced Embedded Behavior Considerations, from [20]

Figure 3.5: Killzone 2 Manual Annotations, from [30]

3.1.4 Killzone 2 Summary

As stated by Killzone 2 programmer Alex Champandard, "strategy is more than the sum of its parts" [29]. More than any other quote, this statement best describes the way that Killzone 2 designers were able to create an extremely high level AI. None of the AI components they utilized were new (including the HTN planner); however, game designers stated that this was the first time such components were combined within a single FPS application. These techniques should be of interest to those who seek to improve upon military simulations.

3.2 PlannedAssault

As video games and military simulations have evolved, each system has developed the potential to display military behavior in a highly realistic fashion. Major achievements have been made in the ability to visualize the detailed actions of a large number of agents; however, the ability for AI systems to devise complex and coordinated maneuvers has gone unrealized. Creating such actions within simulations generally requires a great deal of manual scripting by scenario designers.

Unsatisfied with automated mission planning capabilities in FPS games, a Killzone 2 developer named William van der Sterren sought to remedy this shortfall. His program "PlannedAssault" is a web-based mission planner for Bohemia Interactive's Virtual Battlespace 2 (VBS2) and Armed Assault programs. PlannedAssault creates mission plans that incorporate numerous military tactical considerations including coordinated maneuvers, resource allocation, event triggers, and event synchronization.

3.2.1 Hierarchical Plan-Space Planning

An experienced video game programmer and AI developer, William van der Sterren considered numerous artificial intelligent planning methods for PlannedAssault. Figure 3.6 illustrates the issues that were encountered with planning techniques that were originally considered for use in the PlannedAssault program. Ultimately, Hierarchical Plan-Space Planning offered the best solution for creating an artificially intelligent mission planner.

Within this technique, a plan is represented as a hierarchy of tasks. At the top of the plan are one or more compound tasks needed to achieve a goal. The plan hierarchy is created when methods are utilized to decompose these compound tasks into lower-level compound tasks, and primitive tasks. The tasks themselves describe "activity, ordering, inputs & outputs, and costs" [31]. Many of these tasks bear a strong resemblance to military actions. Examples include tactical tasks such as clear, occupy, and defend, as well as low-level unit and individual tasks such as move, wait, load vehicle, and unload vehicle. Tasks are also classified according to various scope categories including mission, objective, team, tactic,

	Approach, as considered for a multi-unit planning problem						
	"Game Industry" techniques				"Academic" planning techniques		
Issues	(MC) Brute force	GOAP / A*	Behavior Trees	Command Hierarchy	STRIPS	JSHOP	Constraint solving
Incomplete Plan			X				
Plan may contain actions which cannot be explained	X	?					
Picks first/any plan instead of "best"	?			?	X	X	
Synchronizes actions	?		?		X	X	
Cannot explore multiple alternatives			X	X			
Drowns in combinations	X	X			X		?
Cannot bring all facts into planner					X	X	

Figure 3.6: AI Techniques Considered During PlannedAssault Development, from [31]

and unit(s). Figure 3.7 and Figure 3.8 illustrate how these concepts are connected.

These parameters aid in the establishment of a hierarchical plan which is representative of actual military actions. Such a hierarchical structure enables software developers and users to analyze the plan for realism, to identify problems with the program, and to more readily implement solutions. While each of these attributes plays an important role in developing a realistic military plan, the unique features of Hierarchical Plan-Space Planning are found not in the plan's structure, but instead in the method of the plan's creation.

To construct the previously described hierarchy of tasks, PlannedAssault conducts a search in plan-space as opposed to state-space [31]. The significance of this is that the search identifies which higher level tasks are best suited to the situation, prior to determining lower level tactical tasks for individual units. An advantage in efficiency is gained through this technique because "higher level (abstract) choices typically have few options" [31]. This technique shows a strong resemblance to the SHOP Bridge strategy that was described in Chapter 2.

3.2.2 Technical Specifications

The PlannedAssault program provides a unique capability that has not previously been demonstrated in FPS applications. In part, this capability was made possible by the following PlannedAssault technical specifications:

- The main loop consists of an A* search that conducts a best-first analysis of mission plans.
- "Planner methods detail a plan by picking a single abstract task and decomposing it into sub-tasks and/or setting its output values" [31, p. 25].
- Heuristics are used to "estimate task and plan costs... [and advisors] generate, filter, sort, and select the best options for tasks" [31, p. 14].
- Additional factors of interest include task inputs and outputs. These are used "to parameterize tasks, to share results, to act as variables that need to be grounded, and to take in hints" [31, p. 21].
- Typical PlannedAssault input and output types include "unit, objective, area, avenue of approach, landing zone, and path" [31, p. 21].
- PlannedAssault uses plan duration as one of its primary cost indicators.

scope	planner method responsibility	task examples
mission	arrange objectives, allocate units to objectives	mission
objective	define team activities, organize combat and support units in teams	clear, occupy, defend
team	execute tasks as a team, distributing the work according to roles	move, form up, attack, air land, defend, counter-attack, para drop
tactics	synchronize tactical moves between multiple units	formation ground attack, planned fire support, smoke screen
units	arrange co-operation between complementary units	transported move
unit	define end-state	defend sector, guard, close-air-support

Figure 3.7: Planner Methods, after [31]

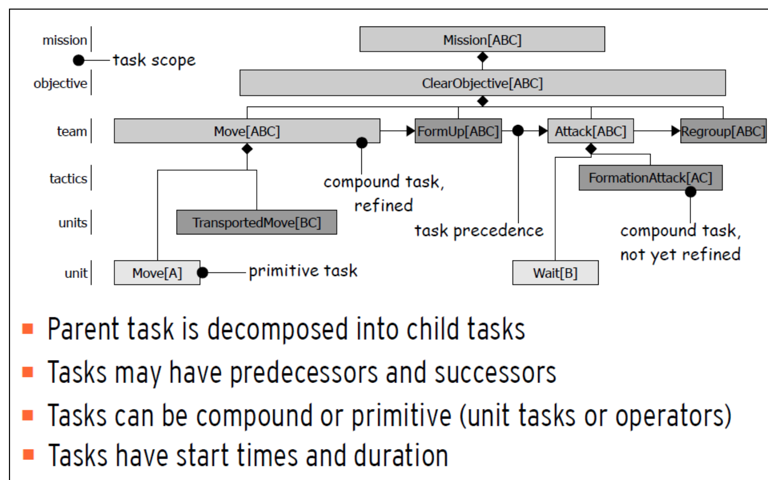


Figure 3.8: Task Types, States and Relations, from [31]

3.2.3 User Interface Description

The AI capabilities of PlannedAssault provide a powerful new way to create complex missions, however that is not its only attribute. PlannedAssault provides an intuitive mission creation interface which could serve as a template for military simulations. The entire program is run through the use of a standard web browser. The PlannedAssault user is presented with several initial choices including terrain and force selection. To ease the development process, the user is provided numerous unit templates which represent military units and their standard equipment. Examples of these unit templates are depicted in Figure 3.9. This process serves as an important step in reducing the amount of time required

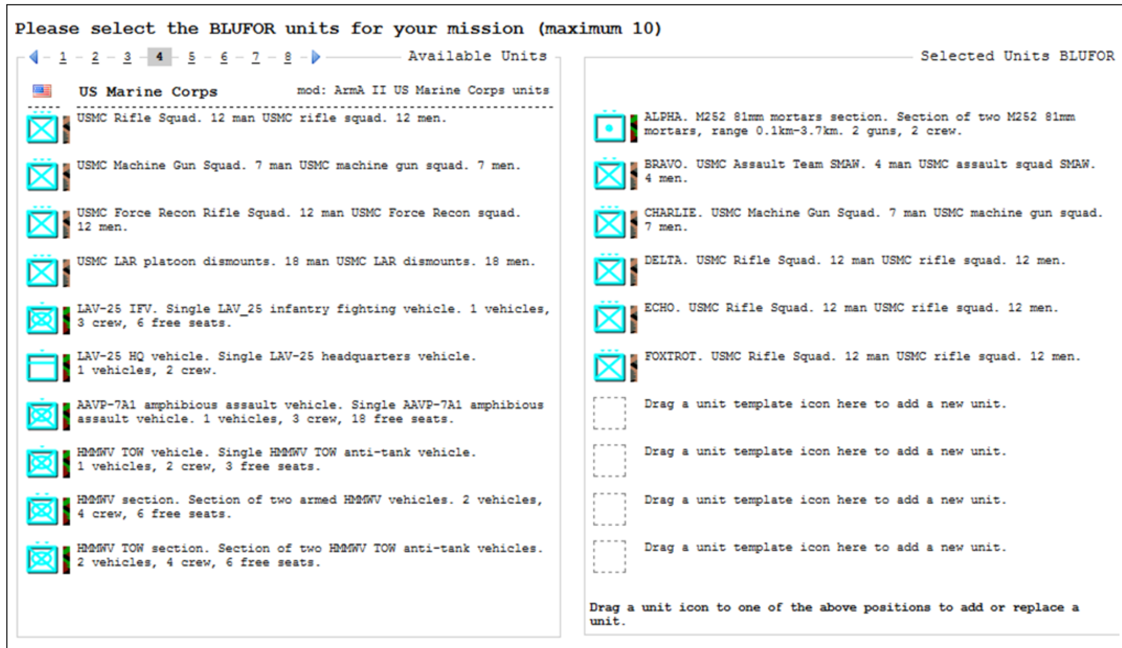


Figure 3.9: PlannedAssault Unit Selection, from [32]

for scenario creation.

Mission assignment is the second important aspect of the PlannedAssault interface. Options to choose from are similar to doctrinal military tasks for both offensive and defensive operations. Figure 3.10 illustrates tasks that were available at the time of this writing. The first type of offensive task describes a standard clearing mission. Options for the selection between a one-, two-, or three-pronged attack enable scenario developers to scale the width of an attacking force's frontage. The second type of mission involves the combination of a frontal and a flanking attack: a fundamental behavior for military units. Program features enable the scenario developer to designate which direction the flanking attack is to come from, or to let the program make that decision.

While the assignment of behaviors to units is of much interest to this work, PlannedAssault also includes several additional features for creating a military plan. Once all forces have been constructed and their missions identified, the scenario designer designates each unit's initial location, as well as several additional inputs relating to time of day, weather conditions, and visibility. Because PlannedAssault is designed to complement both simu-

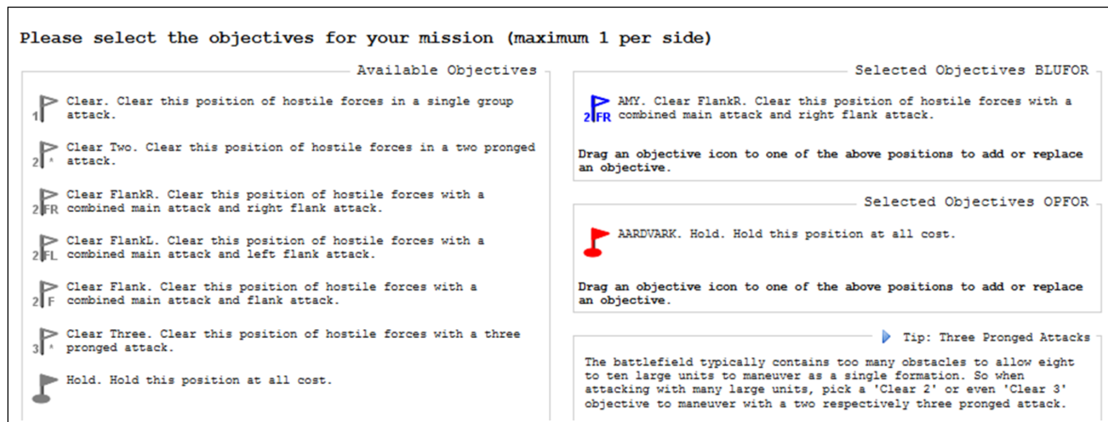


Figure 3.10: PlannedAssault Mission Selection, from [32]

lations and games, the scenario designer also selects to play the role of either a participant or an observer. Once all inputs have been completed, the program begins the process of Hierarchical Plan-Space Planning.

PlannedAssault's finished product is a mission file which the user can download and run in the appropriate version of Bohemia Interactive software. The program is assessed to produce generally realistic schemes of maneuver for both attacking and defending forces. An example of one such scenario is depicted in Figure 3.11 One notable aspect of the program is the utilization of military tactical control measures (TCMs), including attack positions and phase lines. PlannedAssault also attempts to task organize a force in a way that takes advantage of its subordinate units' capabilities. Examples of this include utilizing indirect fire assets to cover the movement of an assault force, and placing a machine gun unit in a position to provide suppressive fires.

3.2.4 PlannedAssault Conclusion

The PlannedAssault program represents a revolutionary step in the process of creating complex military battle plans. Several aspects of the program have the potential to affect the way in which military simulations are created. The first feature of interest is the graphical user interface and the unit templates which allow a user to quickly create a force. The second important aspect of the program is its representation of tactical military tasks. Finally, the hierarchical plan-space planning algorithm is a potentially revolutionary way of

Figure 3.11: PlannedAssault Example Scheme of Maneuver

utilizing artificial intelligence to derive complex plans. Each of these features represents techniques that could prove extremely valuable to military simulations.

3.3 Command Ops

Many of the previously mentioned efforts to refine AI capabilities involve FPS applications. While these games have made important strides in the ability to create dynamic behavior, FPS developers are not the only group interested in AI. RTS games involve a large mixture of automated, semi-automated, and player-controlled agents which interact with each other on battlefields. One of the major differences between the two is that FPS applications utilize entities representing individuals, while many RTS applications utilize entities that represent collections of individuals. This aspect can be observed in the RTS screen shot depicted in Figure 3.12. Unlike their FPS counterparts, the attractiveness of RTS applications is generally not built upon advanced graphics and immersive game play. RTS games depend more heavily upon AI techniques than many other genres. Ensuring that friendly and enemy forces behave in a tactically realistic manner is an important requirement for producing a game that is enjoyable for players and commercially successful.

One of the companies that has made notable strides in this field is Panther Games, creator

Figure 3.12: Command Ops Screenshot, from [33]

of the Command Ops (formerly Airborne Assault) series of RTS games. Formed in 1986, Panther Games is an Australian company that has produced numerous board games and RTS computer games. True to the motto of “When Realism Counts,” Panther Games concentrates its efforts on advancing AI wargame techniques. Focus areas for AI development include the following aspects [34]:

- enable AI to manage subordinates
- enable macro-management
- support a military command structure
- model realistic decision making
- represent orders delay (simulate surprise and the Boyd Cycle)

Panther Games’ success has not only enabled them to create a series of RTS games each more realistic than the last, but has also brought them to the attention of several professional military organizations. In 2010, Panther games was “awarded a contract to conduct a scoping study for the Australian Defence Simulation Office (ADSO) to determine the

defence requirements for a course of action (COA) analyser simulation tool and to plan and cost the development of the Command Ops operational warfare simulation to meet the Defence requirements" [35]. Since then, their success at creating wargame AI has brought them to the attention of many other organizations including the U.S. military.

To understand the reason for this attention, this paper will discuss some of the AI aspects found in two applications from Panther Games' Command Ops series. Conquest of the Aegean (COTA) is an RTS game released in 2006 and set during the World War II (WWII) Greek Campaign. Battles from the Bulge (BFTB) was released in 2010 and "covers the climactic WWII struggle in the Ardennes region of Belgium when Hitler launches his last offensive just before Christmas 1944" [35]. Built upon the COTA foundation, BFTB incorporates several improvements, most notably in the areas of unit tasking and force organization. Command Ops entities generally represent military units that range from platoon to division size, and have been cited for displaying some of the most advanced military AI on the market. Examples of these units and the orders they can be given are shown in Figure 3.13 and Figure 3.14. Highly acclaimed by the gaming industry, COTA was the recipient of numerous awards including the Wargamer's Best Wargame (Gold), CyberStratege's Laurier D'Or (Game of the Year) and War-Historical's Game of the Year (Gold) [35]. Much of this success can be attributed to the creation of entities that are "aware of their situation, able to make decisions, able to act within a command structure (receive and send orders), and able to act independently" [34].

3.3.1 Situational Awareness

One of the first requirements for the creation of smart entities is situational awareness of an entity's surroundings. While there are many things to be aware of, they can generally be categorized as friendly, enemy, or spatial (also referred to as terrain). Knowledge of one's own forces is an important prerequisite for deciding upon a course of action. Some of the friendly force attributes that Command Ops considers include friendly force numbers, capabilities, tasks, and plans. Important considerations can also include force limitations and current location. Once friendly force information has been processed, friendly forces must attempt to collect the same information about their enemies. Friendly forces must prioritize which enemies present the most danger, which enemies create unique challenges, and ultimately which ones must be dealt with first. An additional real life factor implemented

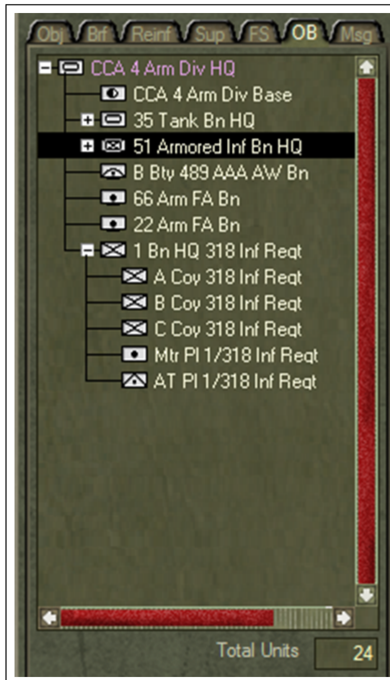


Figure 3.13: Command Ops Unit Hierarchy, from [33]

in Command Ops is intelligence reliability. Battlefield information for both friendly and enemy forces is rarely perfect; the reliability of such information must be considered because it could have a significant impact on operations. Examples of information exchanged between units is shown in Figure 3.15. The final piece of situational awareness concerns spatial or terrain factors. Entities in Command Ops focus on where they are, where they can go, and where they cannot go. These factors play an important role in determining if an entity must react, reassess, or replan [34].

According to Panther Games' President Dave O'Connor,

> The number one thing that you have to get right is route finding. If you can't get good route finding, [then] forget doing AI; it is not your career. It is fundamental. Of all the things you do, it determines how smart [your entities are], if you are developing in any form of a spatial environment. [34]

While being able to develop good routes is extremely important, being able to develop

Figure 3.14: Example Orders and Associated Parameters, from [33]

routes quickly is of equal concern. Because route planning and line of sight (LOS) calculations represented two of the game's most computationally expensive algorithms, Panther Games developers spent a great deal of time optimizing these processes. Combining these terrain related capabilities with information on the dispositions of friendly and enemy forces, is an action that makes situational awareness a fundamental building block for the creation of smart entities.

3.3.2 Decision Making

Decision making is the next aspect of building smart entities. One of the unique decision making aspects of Command Ops is the use of doctrine. While doctrine can be defined in a number of ways, Dave O'Connor describes doctrine as a way of "doing the decision making process that implements a task or a plan" [34]. Command Ops doctrine can be viewed as a set of templates that are consulted during the production of scenario-specific plans and tasks. Because units employ different types of doctrine, the entities within Command Ops can display a wide variety of behaviors that are representative of their real life historical

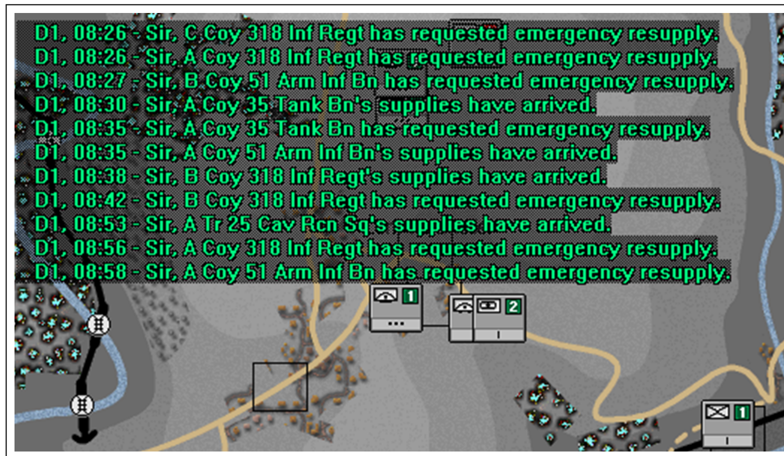


Figure 3.15: Orders Traffic During Gameplay, from [33]

counterparts. Command Ops categorizes its doctrine in two different ways. Plan doctrine covers rules utilized during the formal planning process. Plan doctrine is utilized when a unit creates a new plan or is forced to replan. This differs from tactical doctrine which describes lower level behaviors, and controls how units react to immediate events. Tactical doctrine is assessed continuously throughout the game [33].

The basis for decision making within Command Ops is the Boyd Cycle created by former United States Air Force (USAF) Colonel John Boyd. The Boyd Cycle is embodied by the observe, orient, decide, act (OODA) loop which describes the actions combatants perform while engaged with enemy forces. To account for the interactions of simulated entities, Panther Games made some adjustments to the Boyd Cycle, however the process maintains the spirit of the original OODA loop. Within Command Ops, entities receive (orders), plan, send (orders), act, react, and assess [34]. Receiving is the first step in the process because tasks or guidance received from a higher headquarters have a significant affect on a unit's ultimate actions. When this guidance is combined with a unit's level of situational awareness, that unit is able to create its own plan of action. Each plan contains series of tasks which are the actions that an entity intends to accomplish. When it comes time to act, Command Ops unit actions are not only based on "hard factors" such as weapon performance and unit maneuverability, but also include "soft factors" that embody human attributes such as aggressiveness, decision making ability, and fatigue [34]. These first four

parts of the Command Ops reaction cycle cover the actions taken in the traditional OODA loop, are included in Command Ops' formal planning processes, and depend heavily upon plan doctrine.

It is important to recognize that planning is not finished once a prioritized list of tasks has been produced and is ready for execution. The dynamic nature of combat environments requires units that can react to unexpected circumstances, recognize when their current plan is no longer feasible, and create a new, more suitable plan. The fifth step in the Command Ops decision cycle, react, functions similarly to the immediate action (IA) drills conducted by actual military units. This step makes use of Command Ops' tactical doctrine, is conducted automatically, and is mostly driven by the local context rather than the overall mission. [33]. Another important capability of Command Ops entities is their ability to change or refuse orders. One of the ways this is accomplished is through the reassess step. The dynamic nature of combat operations often serves to invalidate existing plans. When this happens in Command Ops, the reassess step triggers a replan and the reactivation of the formal planning process [33]. Examples of this include a damaged entity's refusal to advance upon a larger force in the face of overwhelming enemy fire. The ability for the Command Ops AI to recognize such dilemmas and take appropriate actions is an impressive accomplishment.

3.3.3 Task Organization

> Our approach was to support a command structure... if you can't do that, then you can't really model military operations. - Dave O'Connor - Panther Games [33]

One aspect of decision making not discussed in the previous section was the act of sending orders. As hierarchical organizations, most military units can be expected to possess a number of subunits. Even in games such as Command Ops where a single entity can be used to represent a large military organization such as a company or battalion, the overall number of subunits in a scenario can grow quickly. Expecting a player to control every entity is an unreasonable task. Command Ops solves this problem by enabling a player to task a single command-level unit, and then use AI to properly task the command-level unit's subordinate units. This is an important capability that requires the player "to trust that an

AI controlled subordinate will do a reasonable job of managing itself and its subordinates" [34]. Accomplishing this required Command Ops designers to develop a series of modular tasks which can be combined to create a battle plan. According to Dave O'Connor, making entities aware that they are part of a command structure was the key to producing scalable and reusable tasks. Assigning tasks to subordinate units is an important capability, however additional capabilities are needed for the creation of smart organizations.

The ability to create and adjust a unit's task organization is the second requirement for creating smart organizations. To best deal with the inherent chaos of combat operations, military units are constantly adjusting their task organization. One example of this technique involves reinforcing a unit prior to an operation. If a unit is tasked with conducting a difficult attack, they will often be reinforced with mortar units that will be used to cover their approach, and assault units which are used to destroy enemy bunkers. Another situation that requires an updated task organization is when a unit is tasked with a mission that requires a number of separate forces which are able to conduct independent or semi-independent operations. If a mission clearly calls for one assault force and two support by fire forces, then a unit must adjust the organization of its subordinates so that forces will be available for each of these roles. This task could involve combining or dividing existing subordinate units.

Much of the capability to accomplish this depends on the three methods through which Command Ops organizes its command structures. The first method, called an organic command structure, describes a unit's normal command structure. The second method is the player command structure. This method is utilized when a Command Ops player places one unit under the command of another organization. An example of this is when a player places an artillery battalion from one division under the command of a different division. The final method is called the current structure; this refers to situations where the Command Ops AI has provided further refinements to a player's command structure [34]. Building upon the previous example, a current command structure could be observed if the Command Ops AI moves the artillery battalion from its player-assigned boss (the division) to one of the division's subordinate brigades. Such actions usually occur during the orders process when a player selects the units that are to receive a specific order. Command Ops' abilities to allocate resources in an intelligent manner and to assign appropriate tasks to

subordinate units are important AI advancements.

3.3.4 Formal Plan Development

Situational awareness, decision making, and smart organizations are important features which Panther Games has spent a great deal of resources developing. Each of these aspects is an important building block for one of the most intriguing aspects of Command Ops, its ability to create realistic military plans. In part, this is made possible because Command Ops designers took an object oriented approach when designing their AI. They anticipated that such an approach would facilitate the creation of a scalable planning system which would function regardless of the entity hierarchy system in use [33].

This was accomplished by embedding decision making AI into every Command Ops entity so that each unit could receive orders, create a plan, and issue orders. Unlike other games which utilize separate strategic AI and tactical AI constructs, the AI in Command Ops is scalable and will function regardless of what military echelon an entity represents [33]. These capabilities are utilized within a process that Panther Games developers refer to as spiral plan development. "Spiral plan development is a recursive methodology which allows [Command Ops entities] to develop a plan within a plan within a plan... It is a way of decomposing complex tasks" [34]. Common orders within Command Ops include the following [33]:

- Move, Probe, Attack
- Defend, Delay, Withdraw
- Fire, Bombard
- Secure/Deny Crossing
- Reorganize
- Rest

To understand this process, one must first understand the hierarchical manner in which Command Ops organizes its plans. Each entity has an overall operational plan which consists of a number of mission plans. Each mission plan is composed of a series of tasks. Major task components include a subject or force group (series of forces arranged in a tree), an action to perform, and an objective. The key part to recognize is that these tasks are scalable, and can be assigned to any size force. This results in a system which dis-

plays many aspects of composable behaviors, a concept that seeks to create "an easy to use, adaptive, and flexible framework for simulating group behaviors" [5].

3.3.5 Command Ops Summary

These aspects are combined with a few other techniques to form the foundations of an extremely powerful AI system. The ultimate result is an RTS game which contains highly refined AI capabilities that have caught the interest of both the U.S. and the Australian militaries. Command Ops AI is also of interest to FPS developers. Though it was developed many years before his own program, PlannedAssault creator William van der Sterren stated that Command Ops continues to embody one of the best known examples of AI planning [36]. The ability to integrate these techniques into modern military simulations would provide numerous advantages in both scenario creation and model realism.

3.4 Tesla Tactics Language and Behavior Validation

The validation of models and behaviors is an important process that affects numerous communities in both the civilian and DOD realms. Many models have mature validation processes and procedures that can be utilized to assure stakeholders of a model's quality. Unfortunately, the same cannot be said for those systems which depend heavily upon human behavior. The validation of automated human behavior is a vital, yet underdeveloped field. Figure 3.16 illustrates the complexities of HBR validation.

As M&S techniques have progressed, a need has arisen to ensure M&S products are suitable for their intended purpose. The suitability of simulations impacts numerous areas including business finances, heavy equipment operator safety, and the combat effectiveness of military forces. Because of the serious nature of these impacts, both businesses and the government have placed an emphasis on formalizing the approaches utilized to certify M&S systems [38]. The process of ensuring simulations are correct and reliable is referred to as verification & validation (V&V) [38]. Traditionally, verification can be thought of as answering if "one built the model right" while validation can be thought of as answering if "one built the right model." Verification determines if a model complies with specifications detailed in a model's requirements. Validation is a process that determines if a model accomplishes its intended use [39].

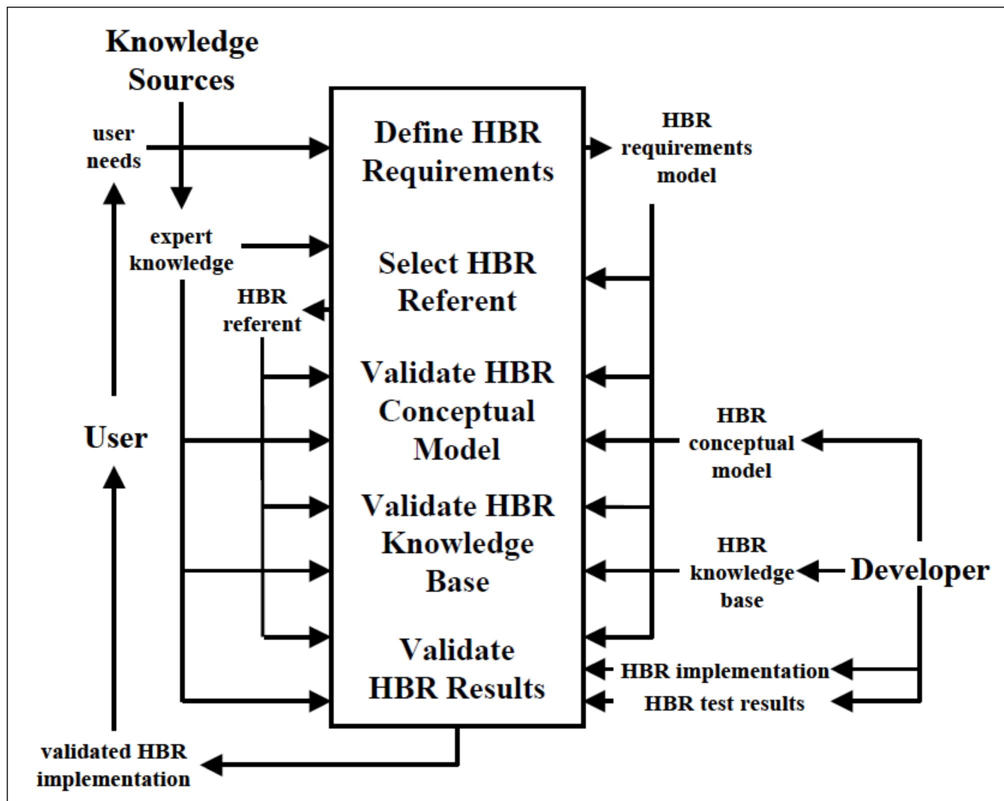


Figure 3.16: Validation Task Information Flow, from [37]

Two recommendations for improving HBR validation involve maintaining close ties with subject matter experts (SMEs) throughout a product's development, and constructing a simulation in a way that enables the developers to understand what is going on inside the model. AI developer Evan Clark describes a method for doing this within DOD simulations. Through the use of a military tactics language known as Tesla, he describes a behavior composition approach to simulations which integrates SMEs and can produce models that are easier to validate. Behavior composition is a process through which complex behaviors are constructed from primitive behaviors, and other complex behaviors. Though the exact definition can vary, primitive behaviors refer to "functionality implemented in source code and packaged up so as to be available to an editor application or scripting engine" [40, p. 12]. Behavior composition systems can generally be classified as goal-based systems in which goals are assigned to individual agents and groups, or knowledge-based systems which utilize some type of FSM. Knowledge based systems are also known as

rule-based systems or embedded expert systems [40].

According to Clark, the Tesla tactical language was created "to bridge the gap between the art and science of war studied by domain SMEs on the one hand, and the logic and algorithms of software applications on the other" [40, p. 10]. Tesla is classified as a knowledge-based system. It is partly graphical and partly textual, and makes heavy use of abstract tactical templates. An example of these tactical templates is depicted in Figure 3.17. One of the strengths of the system is that it can express complicated behaviors more easily than with standard knowledge-based FSM systems. Additionally, because of the language's graphical nature and its strong tie to U.S. military doctrine, Tesla enables SMEs to play a significant role in the software and scenario development. One of the key ways it accomplishes these attributes is through the use of abstract tactical templates. The templates themselves are created using Tesla's graphical interface, and are largely composed of primitive tasks, constraints, and control measures familiar to military personnel. Because their abstract nature prevents references to specific locations, times, or actors, the templates possess scalable and reusable attributes, which can significantly ease scenario development.

While the ability for SMEs to participate in the development process is of great benefit, additional aspects of the Tesla language stand to ease the HBR validation process. The first benefit of Tesla is that it can augment face validation efforts. Clark states that Tesla helps structure validation efforts by shifting the majority of the work to "the creation and approval of test scenarios and later on the approval of template libraries" [40, p. 9]. Additionally, the reusable nature of the templates means that they can be created and validated separately from larger scenarios. One of the problems with face validation is designating testing inputs that adequately cover the required behavior space [40]. Tesla templates ease this problem by aiding developers in partitioning behavior space. Tesla's granular knowledge base is easier to work with than complex knowledge bases full of interrelationships [40]. Finally, an important benefit of the Tesla language is that it facilitates the involvement of SMEs throughout the development process.

Unfortunately, the Tesla system still requires qualitative validation techniques, and is therefore still subject to problems including limited reusability and SME bias. Despite this, it provides advantages that could greatly ease the development and validation of simulations

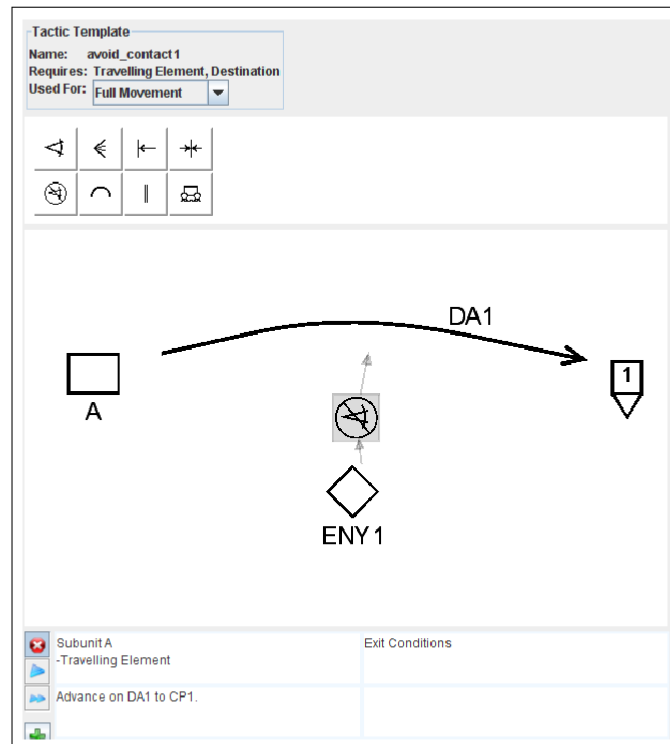


Figure 3.17: Tesla Tactical Template, from [40]

involving automated human behavior. Clark states that "the Tesla system is an approach to tactical behavior composition that gives control over more behavioral elements to the behavior developer than previous behavior composition systems... [and] would augment current face validation efforts by removing impediments identified as significant hurdles in past high profile DOD simulation efforts" [40, p. 11].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Military Doctrine for Simulations

Throughout the course of human history, societies have sought to advance current military tactics in order to give themselves an advantage over their competitors. The USMC provides a unique example of this concept. One of the Marine Corps' unique qualities is that it incorporates, into one organization, many conventional capabilities such as infantry and fixed-wing aviation, which are normally separated by service. For this reason, Marine Corps doctrine is written in a manner so that it is applicable across a wide spectrum of capabilities. A second quality comes from its tie to the United States Navy (USN), a relationship that gives the Marine Corps an advantage in maneuverability at the operational level of war. However one of the drawbacks of staging operations from ships is a limitation on the number of assets a unit can bring to a fight. To succeed in a world where combat assets may be limited, the Marine Corps finds unique ways of employing the people and equipment that a unit has available. The concept of combined arms is one principle that Marines utilize to gain an advantage over their adversaries. MCDP 1-0 describes combined arms in the following way:

> Combined arms presents the enemy not merely with a problem, but with a dilemma–a no-win situation. The commander combines supporting arms, organic fires, and maneuver in such a way that any action the enemy takes to counteract one makes him more vulnerable to another. [41, p. 96]

The harmonious execution of this concept produces an overall unit capability which is greater than the sum of each unit member's capabilities. Military simulations must strive to emulate these effects if they are expected to produce outputs which are representative of actual combat. Unfortunately, this is an extremely difficult task for which there is no simple solution. Software programmers, AI experts, and simulationists have made important strides in creating better representations of human behavior in combat, but there is still much room for improvement.

One way to improve upon combat simulations is through the application of techniques found in military doctrine. Such military documents and manuals offer potential solutions to the problem of replicating human behavior in simulations. This chapter provides a general understanding of important command concepts and doctrinal terminology needed to produce realistic combat simulations. The chapter concludes by providing insights into the mission planning process for a basic ground attack.

4.1 Mission Command and Doctrinal Terminology

Military doctrine represents the collection, organization, and evolution of hundreds of years of military knowledge. The behaviors described are crafted so that they remain relevant in even the most extreme combat situations. Analyzing these techniques is an important step for producing a representative combat simulation. The unique two-way communication techniques and considerations between leaders and soldiers on a battlefield incorporates many aspects that could prove advantageous to AI systems attempting to replicate human behavior. To understand the reason for this, one must first understand certain attributes of the high intensity battlefield for which these military techniques were designed.

- Conventional military forces are organized in a hierarchical manner in order to enable rapid reorganization.
- At all levels, battlefield friction degrades the two-way communication capabilities between units and leaders; this often results in incorrect information.
- Rapidly changing situations often invalidate missions previously assigned to a unit; a mechanism must exist for unit leaders to modify their mission in a way that is beneficial to their higher headquarters.
- Personnel changes due to battlefield casualties can often cause military members to be placed in positions for which they lack experience.

If military operations are to be successful under these harsh conditions, the communications and orders passed between leaders and subordinates must be clear, concise, and flexible. The Marine Corps accomplishes this through the concept of mission command. MCDP 1-0 states that "under mission command, commanders provide subordinates with a mission, their commander's intent, a concept of operations, and resources adequate to accomplish the mission" [41, p. 7-5]. Two elements of mission command are particularly important

to decentralized execution. The concept of operations is an overview of the actions that a unit will perform during a specific mission. According to MCDP 1-0, "commander's intent is the commander's personal expression of the purpose of the operation. It must be clear, concise, and easily understood two levels down. It may also include end state or conditions that, when satisfied, accomplish the purpose" [41, p. 3-15]. Both the concept of operations and commander's intent are powerful methods for conducting decentralized operations in which leaders at all levels can be trusted to exercise judgment and initiative. When battlefield events nullify the utility of a unit's specific mission, the best way for a unit to remain relevant is for its leader to select a new mission that is nested within his commander's concept of operations, or is beneficial to the process of accomplishing the commander's intent.

Within simulations, the dynamic behavior of agents can be improved through mechanisms that represent a concept of operations and a commander's intent. However, this is only part of the solution. Accomplishing these tasks requires standardized terminology that can be used for clear and concise communication. The Marine Corps has devoted much attention to establishing such a lexicon. Clearly defined terminology describes operational aspects that range from the type of operation to be conducted, to the general type of behavior that a unit will execute, to the specific mission that subordinate units are tasked with accomplishing. Understanding this terminology is a vital requirement for accurately representing military behavior in a simulation.

4.1.1 Classification of Marine Corps Operations

Within computer programs, parameters play an important role in the actions exhibited by the program. The terms that the Marine Corps uses to define its actions function in much the same way. At the highest level, a commander must select a major operation for his unit to conduct. The most fundamental of these decisions is represented by a choice between offensive and defensive operations. This is a complicated decision which depends on numerous factors such as the orders a unit has received from its headquarters, a unit's current state, and the unit's perception of its enemy's state. Because a unit's subsequent actions are completely dependent upon this choice, AI systems in simulations must be able to make decisions that are representative of those of military commanders.

It should be also recognized that additional types of major operations exist. Areas for consideration include reconnaissance, security, counterinsurgency, and sustainment operations. Amphibious operations have their own subset of mission types including raids, assaults, withdrawals, demonstrations, and support to other operations [41, pp. 2-23 - 2-24]. The choice among each of these operation types will have a major affect on the unit's eventual actions. Because the choice between offense and defense represents one of the most fundamental decisions that a commander must make, this serves as a logical area to start. The main thrust of this thesis will be to effectively define portions of offensive operations that are applicable to military simulations.

4.2 Offensive Operations Overview

MCDP 1-0 states that "offensive operations seek to gain, maintain, and exploit the initiative, causing the enemy to react" [41, p. 6-4]. As shown in Figure 4.1, offensive operations are generally divided into four major categories: movement to contact, attack, exploitation, and pursuit. Each of these operation types is best suited to different situations. They are of interest to military simulation designers because the type of offensive operation a unit conducts will have a major impact on the type of tactics utilized, the allocation of external resources such as fire support, and the unit's behavior when it makes contact with enemy forces. Joint Publication 1-02 and MCDP 1-0 provide the following descriptions of these operations.

Movement to Contact: "a form of the offense designed to develop the situation and to establish or regain contact" [42, p. 245]. The purpose of this offensive operation is to further develop a commander's knowledge of enemy forces. It is normally conducted at the opening stages of a campaign. It requires formations specialized for scouting,

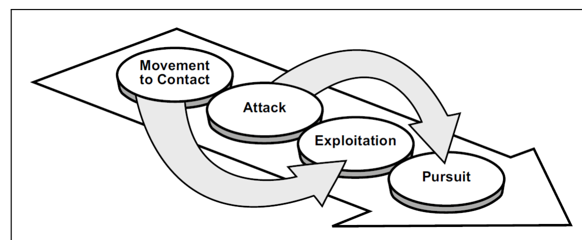


Figure 4.1: Four Types of Offensive Operations, from [41]

speed, and flexibility. Figure 4.2 illustrates how a unit generally task organizes its subunits for movement to contact operations.

Attack: "an offensive operation of coordinated movement and maneuver supported by fire to defeat, destroy, or capture the enemy or seize/secure key terrain" [41, p. 9-5].

Exploitation: an offensive operation that "disorganizes the enemy in depth usually following a successful attack. The exploitation extends the initial success of the attack by preventing the enemy from disengaging, withdrawing, and re-establishing an effective defense" [41, p. 9-7]. Because the reserve force is typically tasked with conducting an exploitation, the unit commander must make the decision to execute this offensive operation by carefully weighing risks and rewards.

Pursuit: "an offensive operation designed to catch or cut off a hostile force attempting to escape" [41, p. 9-8]. This type of attack requires mobile forces that can quickly attack an enemy which is also moving.

These operations are defined so that they occur in a logical flow. However, it should be noted that they will not always be performed sequentially. "For example, a movement to contact may be so successful that it immediately leads to exploitation or an attack may lead directly to pursuit" [41, p. 9-4]. Additionally, it should be recognized that within the bounds of a single operation, the same unit will rarely perform all four types of offensive actions.

4.2.1 Types of Attacks

The attack offensive operation is subdivided into several categories. These include hasty, deliberate, spoiling, counterattack, feint, demonstration, reconnaissance in force, and raid.

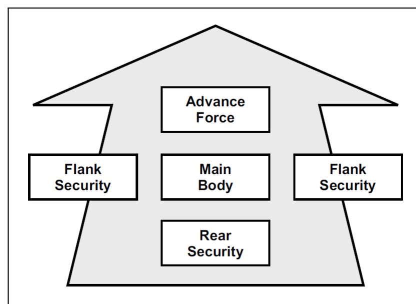


Figure 4.2: Movement to Contact Task Organization, from [41]

They are significant to simulations because the type of attack selected plays a major role in determining a unit's behavior. The choice between hasty and deliberate attacks affects the amount of time that a unit spends preparing for an attack. A spoiling attack is designed to ruin an enemy's attempts to form for its own attack. Feints, demonstrations, and reconnaissance in force operations affect the criteria in which a unit will break contact when it is engaged with an enemy force. A raid signals that a unit must conduct a planned withdrawal upon completion of its primary mission. Finally, a counterattack is conducted by a unit that is normally participating in a larger defensive operation [41, pp. 9-5 - 9-6].

4.2.2 Forms of Maneuver

Forms of maneuver illustrate the manner in which an offensive force seeks to interact with its enemy. If a unit is given the mission of attacking an enemy, that unit's form of maneuver plays a major role in the offensive unit's path selection. It also affects the portion of the enemy force that the offensive force seeks to attack. The six forms of maneuver are frontal attack, flanking attack, envelopment (single or double), turning movement, infiltration, and penetration. The frontal attack, flanking attack, and penetration are of interest to this work and are depicted in Figure 4.3.

A frontal attack seeks to attack the main enemy force, and occurs from the direction that an enemy is facing. It is utilized when an offensive force possesses overwhelming combat power, or when a unit's direction of attack is fixed. An example of this occurs when offensive operations call for several attacking units to be placed next to each other. To minimize the possibility of fratricide, the direction of each attacking unit must correspond to its higher headquarters' direction of attack.

A flanking attack seeks to attack a vulnerable portion of the main enemy force. The primary example of this involves finding a direction of attack that avoids the area that is covered by the majority of enemy defenders. Accomplishing this requires pathfinding capabilities, and knowledge of an enemy's orientation.

A penetration disrupts an enemy defensive position by concentrating an attacking force's combat units, and attacking through a narrow portion of the enemy's position. This operation is undertaken to disrupt an enemy defensive position or to access a specific capability in the enemy's rear area. Three general stages of penetration include rupturing the enemy

position, widening the gap so that follow-on forces can move through it, and seizing an objective in the enemy's rear area [41, p. 9-16].

The three additional forms of maneuver - envelopment, turning movement, and infiltration - play important roles in military operations, but are beyond the scope of this work. Envelopment involves maneuvering around or over an enemy's primary defense in order to secure an objective in the enemy's rear area. A turning movement has a goal similar to an envelopment, however the distance between the enemy's defensive front and the offensive force's objective is much greater. An infiltration involves breaking a unit down into small elements, having them move through gaps in an enemy defense, and then reassembling in order to accomplish a separate mission.

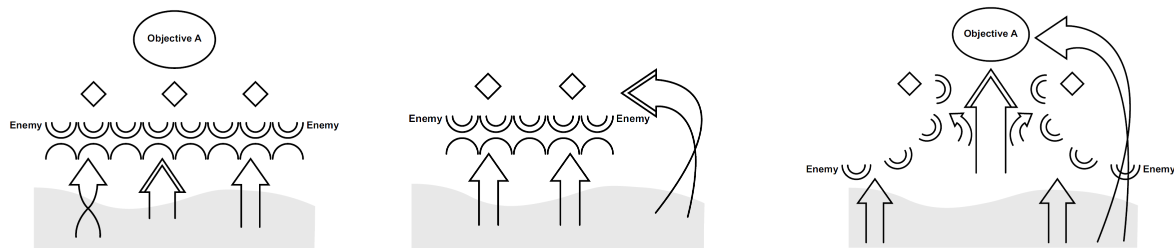


Figure 4.3: Frontal Attack, Flanking Attack, and Penetration Examples, from [41]

4.2.3 Tactical Tasks

The final high level consideration for planning a military operation involves the assignment of specific tasks to each unit. As stated in MCDP 1-0, "tactical tasks have precise definitions that describe what is to be accomplished" [41, p. 3-26]. Understanding the definitions of each task is important for military members, because it allows missions to be communicated more easily.

Tactical tasks can be classified as specified, implied, or essential. A specified tactical task is one which a superior gives to a subordinate. An implied tactical task is one which a subordinate has not been told to do, however he knows it must be done in order to accomplish his primary task. An essential task is one which a commander identifies as being vital for the overall mission's accomplishment.

It is important to recognize that tactical tasks can have different areas of focus. They can be enemy-oriented, friendly force-oriented, terrain-oriented, or environmentally-oriented [41,

pp. C-1 - C-6]. An enemy-oriented task means that that the location of the tactical action is dependent upon the location and disposition of an identified enemy force. Examples of this include block, destroy, and support by fire. Similarly, a friendly force-oriented task is dependent upon the location and disposition of an identified friendly unit. Examples include follow and protect. Terrain-oriented tasks are the final type of task applicable to this thesis. In this situation, a unique terrain feature is the driving force for a unit's mission. These are potentially the easiest tasks to implement in simulations because the location of the task's goal is generally fixed. Example tasks include clear, retain, and seize. Reconnoiter can be a terrain-oriented or an enemy-oriented task.

The correct implementation of tactical tasks within military simulations would have many positive effects, including making a simulation easier to construct and easing communication difficulties between scenario developers and military officers. For a complete understanding of USMC tactical tasks, simulation developers should reference MCDP 1-0 "Operations." A basic understanding of all of these concepts is vital to making simulations more representative of actual military operations.

4.3 Attack Overview

Planning for military operations is a complicated task comprised of many steps. One generalization of the military planning process divides it into two main parts: high-level conceptual planning and low-level detailed planning. Both steps are required to produce an effective plan. Important decisions made during high-level planning include classifying the type of operation to be conducted, deciding upon a general form of maneuver, and developing a mission statement which makes use of one or more tactical tasks. When these steps have been accomplished, a plan can be produced that details exactly how a unit intends to accomplish its mission.

Detailed operational planning requires numerous decisions that involve assigning specific tasks to units, prioritizing the operation's various efforts, and establishing movement control measures. This section discusses several of the key decision points that a commander must make when planning the actions of a military unit. The first concept discussed involves identifying the most decisive part of an operation.

4.3.1 Main and Supporting Efforts

A key lesson taught to Marine Lieutenants learning the art of defensive tactics is that if they attempt to defend everywhere, then they will effectively defend nowhere. This learning point is instrumental in teaching the concept of a main effort. When planning an operation, Marine commanders identify the most critical or decisive aspect of the operation. The subordinate unit charged with achieving this task is designated as the unit's main effort. According to MCDP 1, "the main effort receives priority for support of any kind. It becomes clear to all other units in the command that they must support that unit in the accomplishment of its mission" [43, pp. 90-91]. Other subordinate units are designated as supporting effort 1, supporting effort 2, and so forth. The amount of external support that a subordinate unit receives is highly dependent upon its priority.

Within military operations, this is a powerful tool that increases unity of effort between subordinate units, and focuses combat power on the part of the operation that the commander views as his bid for success. This concept also increases a commander's ability to adjust the actions of subordinate units in the middle of combat operations. Should a unit encounter an unexpected situation or an unanticipated opportunity, a commander can quickly refocus his subordinate units by designated a new main effort. Employing this concept within simulations has the potential to greatly increase the an agent's ability to represent realistic military behaviors.

4.3.2 Supporting Relationship Example

While the concept of the main effort is important to military operations, it should be recognized that this does not prevent other subordinate units from receiving support. The military accomplishes this by establishing several command and support relationships which are defined in MCDP 1-0. For the purpose of the following example, an infantry company will serve as the supported unit. Enabling units will include a mortar section, a machine gun squad, and an engineer team.

general support (GS): "That support which is given to the supported force as a whole and not to any particular subdivision thereof" [41, Glossary-15]. If the mortar section is placed in GS of the infantry company, then each of its infantry platoons can expect to receive fire support. When determining exactly how fire support will be apportioned,

a commander will normally publish a "priority of fires" list which determines what unit receives support if multiple fires requests are received at the same time.

direct support (DS): "A mission requiring a force to support another specific force and authorizing it to answer directly to the supported force's request for assistance" [41, Glossary-13]. A machine gun squad in DS of a platoon gives all its support to the platoon. The machine gun squad is tasked by that unit's platoon commander. The platoon commander does not have the ability to reorganize the machine gun squad's hierarchy. This relationship is utilized when a commander wants to reinforce a unit's capabilities for a specific mission or for a temporary amount of time. If required, DS units can be re-tasked in the midst of combat operations.

attached: This relationship is utilized when an enabling force is placed in support of a specific unit for an extended amount of time. An engineer team that is attached to an infantry platoon essentially becomes part of that infantry platoon. Unlike the previous DS example with the machine gun squad, the platoon commander in this situation has the ability to change the task organization of the engineer team.

4.3.3 Tactical Control Measures

One of the final detailed steps of planning involves the establishment of TCMs. TCMs are tools that a commander utilizes to coordinate the behavior of his subordinate units. The definitions for the meaning of each TCM is standardized, as is a graphic that represents the TCM on a tactical map. A complete listing of Marine Corps and Army TCMs can be found in Marine Corps Reference Publication (MCRP) 5-12A, "Operational Terms and Graphics."[4] TCMs are defined for all types of operations in which the Marine Corps participates; many of these have complex meanings that are beyond the scope of this work. The following case study of a basic ground attack includes additional examples of TCMs that should be implemented in a simulation's mission planner.

4.4 Case Study: Conduct of an Attack

As an organization whose primary purpose is to be on the offense, the conduct of a basic infantry attack is one of the first operations that Marines are taught to conduct. Successfully accomplishing an attack requires the coordinated execution of numerous basic tasks.

[4]For U.S. Army forces, this publication is titled FM 1-02 "Operational Terms and Graphics."

Marine Corps doctrine eases this process of coordination by standardizing the actions that a unit takes when conducting an attack. Figure 4.4 illustrates TCMs that depict the stages of a standard attack. The intent of this section is to illustrate the military techniques and concepts utilized within an example attack. The plan will be developed in accordance with the following high-level attributes:

Operation Classification: Offense
Offensive Operation Type: Attack
Attack Type: Deliberate
Form of Maneuver: Frontal
Mission: Seize

4.4.1 Preparation Phase

The first step in conducting an attack involves the preparation phase. MCWP 3-11.2 states that this phase is "usually accomplished in three steps: movement to the assembly area, final preparations in the assembly area, and movement to the line of departure" [44, p. 4-2]. The first step requires most units to converge on a designated assembly area. Because the assembly area is generally located in a secure portion of the battlefield, movement is often conducted in an administrative manner in order to maximize speed. Likely formations include a route column or a tactical column. If the assembly area is not located in a secure portion of the battlespace, subordinate units should move to the assembly area in an appropriate tactical formation.[5] The appropriate security posture is dependent upon the situation. The chance of making contact with an enemy force is generally classified as contact remote, contact possible, contact probable, or contact imminent. Units typically adopt a tactical formation and a security posture that is appropriate to their perceived chance of making contact with an enemy force.

The assembly area is generally where units conduct their final planning and preparations prior to conducting an attack. Some actions in the assembly area are not applicable to simulations, however other aspects should be considered for inclusion in an automated mission planner. First, there will be some type of security in an assembly area. If security

[5] It should be noted that in this context, the term formation refers to the physical disposition of the members of a unit both when the unit is moving, and when it is not moving. This differs from the way that the term formation is used in COMBATXXI where it is usually only applicable to moving units.

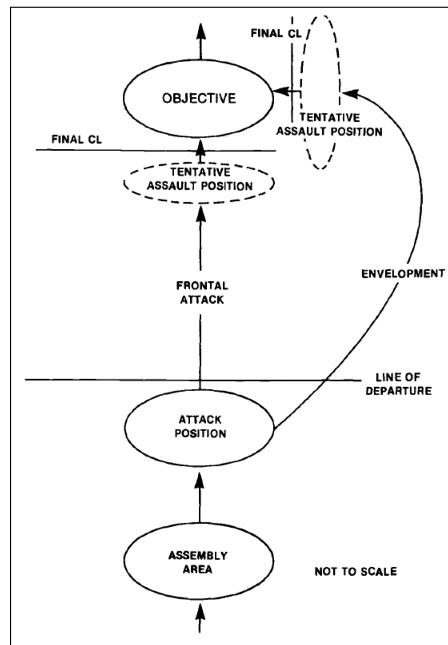


Figure 4.4: Tactical Control Measures for a Basic Attack, from [44]

is not being provided by an external unit, then the attacking unit itself will form a 360 degree perimeter. Adjustments to subordinate unit task organization will be made, and units will begin getting into the proper order of movement dictated in the attack order.

The next step in an attack is to begin movement from the assembly area to the line of departure. This step has several key aspects that should be incorporated into simulations. The first item of interest is the trigger that causes a unit to begin its movement. This trigger can be an order from higher headquarters, a signal of some type, a specified time, or a condition that describes the state of the environment.

When a unit begins movement, they must be in the proper order of movement. As before, tactical formations will still depend on the current enemy threat, however the formation is likely to employ at least some additional security mechanisms. This is typically the first step where a unit employs the concept of a base unit. This is an important mechanism for controlling the movement of large numbers of individuals, and is employed in all types of military operations.

Base Unit

According to MCWP 3-11.2, the base unit is designated by the commander to assist in maintaining direction, position, and rate of march [44, p. 4-16]. Other units use the base unit as a guide to adjust their own movements. It is important to note that the base unit is not necessarily the main effort. The base unit can also change throughout the course of an operation. The unit selected for this task is often the subordinate unit that the commander feels can best navigate the force to its destination. In this case, the unit's destination is an attack position.

Attack Position

According to MCWP 3-11.2, "the attack position is the last concealed and covered position occupied by assault echelons before crossing the line of departure. It is the location where final coordination, last-minute preparations, and, if not already accomplished, deployment into initial attack formations are effected" [44, p. 4-20].

4.4.2 Movement From the Line of Departure to the Assault Position

The next portion of an attack is the conduct phase. By doctrine, it begins when the first troops cross a phase line (PL) known as the line of departure (LD).

Phase Line

Joint Publication 1-02 defines a PL as "a line utilized for control and coordination of military operations, usually an easily identified feature in the operational area." [42, p. 282]. PLs can be used to coordinate actions such as maneuver or fires. If a PL is used to coordinate maneuver, then units are generally not allowed to cross it until an order is received or certain conditions are met. If a PL is used to coordinate fires, then fires agencies will be instructed to commence, shift, or cease fires when a unit crosses the PL. Operation specific PLs can be created in addition to the ones discussed in this case study. PLs are usually named after colors.

Line of Departure

MCWP 3-11.2 states that "the line of departure is a line established to coordinate attacking units when beginning the attack" [44, p. 4-34]. This is a PL that is given its own name because it is incorporated into most military operations. Units are not allowed to cross the line of departure until certain conditions are met, or triggers are received.

When a unit crosses the line of departure, they generally adopt a higher security posture. The most common threat state is contact probable, however certain situations could be classified as contact possible or contact imminent. The base unit concept continues to serve as an important technique for coordinating the movements of individual combatants. Depending on the type of formation adopted, it is possible for a commander to re-designate his base unit.

At this point, a unit is maneuvering to its assault position. While this simple attack example incorporates only basic techniques and TCMs, MCWP 3-11.2 describes several additional concepts that are worth noting.

Release Point: clearly defined points where units are released to the control of their respective leaders [44, p. 4-36]. If an attack plan calls for it, this is typically the area where maneuver elements break away from base of fire elements.

Base of Fire Element: "the base of fire covers the maneuver element's advance toward the enemy position by engaging all known or suspected targets. Upon opening fire, the base of fire seeks to gain fire superiority over the enemy" [44, p. 4-22].

Maneuver Element: "the mission of the maneuver element is to close with and destroy or capture the enemy. It advances and assaults under covering fire of the base of fire element. The maneuver element uses available cover and concealment to the maximum" [44, p. 4-22].

Fire and Maneuver: "the process whereby elements of a unit establish a base of fire to engage the enemy, while another element maneuvers to an advantageous position from which to close with and destroy or capture the enemy. Supporting fires from weapons not organic to the unit may be provided" [44, p. 4-21].

Assault Position

The focus of this stage is on maneuvering the entire force to the assault position. MCWP 3-11.2 states that "the assault position is located as close as the assaulting element can move by fire and maneuver without sustaining casualties or masking covering direct or indirect fires. The assault position should be easily recognizable on the ground and ideally should offer concealment and cover to the attacking force" [44, p. 4-25].

Direction of Attack

The location of an assault position depends heavily on whether or not a unit has been granted the ability to choose its direction of attack. If a unit is given the ability to choose their own direction of attack, then they can locate their assault position in the place that most supports their scheme of maneuver. This situation generally occurs when the attacking unit is the only force attacking an enemy objective, and when it does not have to worry about its fires affecting adjacent friendly forces. However, if a unit's direction of attack has been specified, then its options are more limited. This situation occurs when multiple units are attacking the same objective, or when a maneuvering element seeks to close on an enemy position, but not get in the way of suppression provided by the base of fire element.

4.4.3 Movement From the Assault Position Through the Objective

The next stage involves movement from the assault position to the enemy objective. Because assault positions have a high likelihood of being covered by enemy indirect fire weapons, a unit will stay in the assault position only as long as is necessary to make final preparations. For a simulation, this will typically mean moving a unit into its final assault formation, and updating the environment state to contact imminent. Though there is not an official "line of departure-type" PL between the assault position and the objective, units will often coordinate their actions with something known as an inter-visibility (IV) line. This is typically a terrain feature that provides the last amount of concealment between an attacking force and an enemy defense. The attacking force will attempt to simultaneously cross the IV line with as many combatants as possible in order to have the maximum number of weapons systems available for firing on enemy positions. Because of the hectic nature of this stage, success is heavily dependent on the execution of specific battle drills and leadership techniques which are described in MCWP 3-11.2.

Fire and Movement: "Once the maneuver element meets enemy opposition and can no longer advance under the cover of the base of fire, it employs fire and movement to continue its forward movement to a position from which it can assault the enemy position. In a maneuvering squad, fire and movement consists of individuals or fire teams providing covering fire while other individuals or fire teams advance toward the enemy or assault the enemy position" [44, p. 4-21].

Fighter-Leader Concept: "In the attack, fire team leaders act as fighter-leaders, control-

ling their fire teams primarily by example. Fire team members base their actions on the actions of their fire team leader. Throughout the attack, fire team leaders exercise such positive control as is necessary to ensure that their fire teams function as directed" [44, pp. 4-22 - 4-23].

Squad Leader Actions in the Assault: "The squad leader locates himself where he can best control and influence the action... To maintain control of the squad under heavy enemy fire, the squad leader positions himself near the fire team leader of the designated base fire team. By regulating the actions of the base fire team leader, the squad leader retains control of the squad. The base fire team leader controls the actions of his fire team; the other fire team leaders base their actions on those of the base fire team" [44, p. 4-23]. Alternate locations for a squad leader include positions that offer better abilities to observe the fight, to communicate instructions, or to control key weapon systems.

Final Coordination Line & Limit of Advance

As the assaulting force makes its final advance toward the enemy position, it relies upon the suppressive effects of friendly fires to prevent enemy forces from returning effective fire. If external agencies are being used to provide suppressive fires, the assault unit will pass through PLs that trigger the supporting agencies to shift or cease fire. The PL at which all external fires cease is called the final coordination line (FCL). At this point, the assault force depends on its own fires to suppress enemy positions. When the assault force closes the distance to the enemy position, it makes use of grenades, suppressive fire, and close combat to eliminate enemy forces. The assault ends when the unit reaches the limit of advance (LOA). This is a final PL that is established beyond the objective. At this point, the unit begins the consolidation and reorganization phase.

4.4.4 Consolidation and Reorganization

Due to casualties, low ammunition, and general disorganization, an assaulting force is typically at its most vulnerable when it reaches the LOA. For this reason, this is the time period during which it must be most concerned about enemy counterattack. MCWP 3-11.2 states that "consolidation is the rapid organization of a hasty defense in order to permit the attacking unit to hold the objective just seized in the event of an enemy counterattack" [44, p. 4-30]. Goals during this step include redistributing ammunition and positioning forces so

that all sectors of fire are covered. The direction that a unit covers is generally the direction in which it was attacking, however this could change depending upon unique enemy situations or terrain features. If a unit includes members with automatic weapons, those individuals are usually positioned to cover the enemy's most likely avenues of approach. During consolidation, the emphasis is on quickly establishing a defensive position and not reforming units. Units which have become fragmented during the assault will not reform until the order to reorganize has been given.

Reorganization is a process which refines many of the hasty defensive measures taken during consolidation. This process involves tying together the flanks of multiple defensive positions, replacing leadership and automatic riflemen who have become casualties, and reforming units which were separated during the assault. The positioning of special weapons systems such as machine guns, rocket launchers, and mortars may also be adjusted during this final step.

4.5 Additional Considerations for Implementing Military Doctrine in Simulations

This chapter discussed high level issues that are typically considered when a commander develops an offensive operation, and presented a case study of a basic ground attack. The following section details additional considerations of interest to military planners and simulation developers. These techniques are conducted by units at all echelons of command.

4.5.1 Relative Combat Power Analysis

According to MCWP 5-1, "relative combat power analysis (RCPA) provides planners with an understanding of friendly and adversary strengths and weakness relative to each other. While force ratios may be important, the numerical comparison of personnel and major end items is just one factor to balance with other factors, such as leadership, morale, equipment maintenance, training levels, weather, demographics, and cultural environment" [45, p. 2-5]. While commanders planning operations utilize this tool to help decide upon courses of action, it can also be used in the midst of operations to determine appropriate tactical behaviors.

4.5.2 Suppress, Assess, Move, Kill

The suppress, assess, move, kill (SAM-K) principle is used by combat forces to enable movement in the face of enemy opposition. This is a technique which should be of interest to military simulations. The first step in this process is to assign subordinate forces the mission of suppressing perceived enemy positions. When they begin this action, the unit leader will assess whether or not the enemy is adequately suppressed. If not, the leader will modify his orders concerning the suppression of enemy positions. If the unit leader believes the enemy is suppressed, then he will authorize another portion of his unit to maneuver to its next position. This process is repeated until some or all of the unit has reached a position that allows the destruction of the enemy force, or the accomplishment of the unit's assigned mission [46, p. 7-56].

Defining effective suppression is a difficult task that must be accomplished in order for the SAM-K principle to work. The following list describes the techniques that Marine instructors use within training scenarios to define effects of enemy suppression on a moving unit [47, p. 3].

No Fire: unit can move freely
Sporadic Fire: unit is receiving pop-shots from enemy forces, but can continue to move
Effective Fire: unit can continue as long as they perform good fire and movement
Heavy Fire: moving unit is pinned down and can not move without assistance

4.5.3 Battlespace

A commander will often divide his battlespace into different categories. Typical classifications include an area of operations, an area of influence, and an area of interest. Classifications are assigned based on a military commander's ability to affect an area. The doctrinal definition for these terms is found in **JP!** (**JP!**) 1-02 "DOD Dictionary of Military and Associated Terms." A major purpose of these classifications is to shape a unit's intelligence collection efforts. Within simulations, a battlespace management tool would be useful for scoping the amount of information that is used to build a unit's situational awareness.

4.5.4 Areas for Future Consideration

The attack case study in this chapter presents a synopsis of a basic attack as one is described in MCWP 3-11.2 Marine Rifle Squad. An attack is one of the fundamental behaviors upon which many other actions are built. If a simulation is able to correctly replicate the actions taken during an attack, then other behaviors should be considered.

Defensive Operations: Offensive and defensive operations are closely linked. Successful units must be able to transition seamlessly between the two.

Complex Offensive Operations: In normal operations, units rarely conduct attacks by themselves. Marines seek to utilize the combined arms concept; this technique calls for the integration of a variety of units and weapons systems.

Specialized Actions: Units and subordinate units will often be required to perform specialized actions in response to specific situations. Such tactical operations include passage of lines, linkups, reliefs in place, obstacle crossings, and breakouts from encirclement [41, p. 6-4].

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Challenges Inherent to Scripted Scenarios

One of the difficult areas of human behavior representation concerns choosing a test scenario that is capable of demonstrating, and ultimately validating, the simulated human behavior. Just because a simulation is capable of creating reasonable human behavior in one situation does not mean it will be capable of doing so under different conditions. A potential solution to this problem can be found by studying and simulating pre-existing military training scenarios.

The Marine Corps Air Ground Combat Center (MCAGCC) in 29 Palms, California, is the Marine Corps' premier location for conducting live fire combined arms training. Within the MCAGCC, Range 400 (R400) and Range 401 (R401) are the primary venues for training Marines to conduct a company (reinforced) live fire attack. R400 has existed for many years, while R401 was created during the wars in Iraq and Afghanistan in order to allow more forces to conduct training. Both of these ranges achieve similar training objectives in similar terrain. R400 has been selected for this study because of the existence of additional literature concerning range operations and more mature performance evaluation standards. Figure 5.1 and Figure 5.2 depict the notional enemy defensive position that Marines are assigned to attack at R400. It should be noted that light armored vehicle (LAV) platoons are also able to conduct offensive actions on this range.

5.1 Range 400 Overview

The execution of R400 at MCAGCC is a capstone event for USMC infantry units. The first reason for this is that R400 involves the use of every type of weapon system that is organic to a Marine infantry battalion. Each R400 run involves an infantry company which has been reinforced by battalion 81mm mortars, heavy machine guns, and scout snipers. Additionally, Combat Engineer units are usually placed in support of infantry companies. The second reason for this range's importance concerns its associated training goals. These goals include small unit tactical actions such as grenade employment, trench clearing, fire and movement, and fire and maneuver. Higher level command and control, and fire support coordination skills are also practiced at R400. Training goals for all of the company's

Figure 5.1: Notional Enemy Forces on R400, from [47]

specialized units and battalion reinforcements are also integrated into the range. The final reason for the range's importance concerns evaluation procedures.

R400 training goals are directly tied to performance standards described in the USMC T&R manual. Evaluations are conducted by Marines from the Tactical Training Exercise Control Group (TTECG). These Marines accompany each unit during its R400 run, and fulfill several important actions. First, they provide an assessment on how well units execute appropriate battle drills, how efficiently they communicate, and how the units' leaders make decisions. Next, the evaluators have the ability to adjust the type of simulated enemy actions that the training unit faces during its range run. These simulated enemy actions known as "paints" typically involve describing the type and volume of enemy fire that a friendly unit is receiving. The paints have an effect on the available actions a unit can take. They force unit commanders to make good decisions and execute tactical actions with

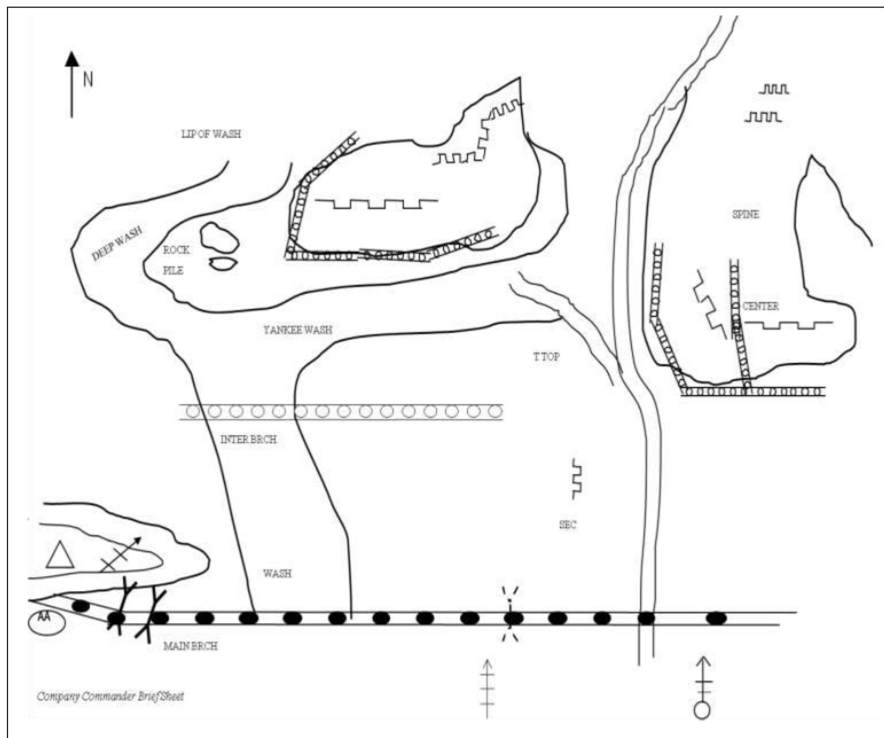


Figure 5.2: Range 400 Company Commander Brief Sheet, from [47]

precision. Finally, the evaluators document each unit's R400 performance so that higher level commanders can assess the proficiency of one of the most fundamental infantry units, the reinforced infantry company. Because of the range's comprehensive training goals and because of the serious manner in which Marine Corps leadership views the execution of this range, a simulated version of R400 is the perfect venue for assessing the effectiveness of a military simulation's ability to execute automated tactical behaviors.

An example R400 attack is depicted in Appendix B. This example attack involves a reinforced infantry company conducting an attack against the range's standard array of enemy forces. While this organization is typical for actual Marine units conducting exercises at R400, this setup is not required for simulated versions of R400. Within COMBATXXI, adjusting the size and composition of friendly and enemy forces allows simulation designers to test many aspects of automated tactical behavior capabilities.

5.2 Training Objectives and Performance Evaluation

Because the normal size of unit conducting training at R400 is an infantry company reinforced by battalion assets, numerous tactical actions are available for evaluation. Figure 5.3 and Figure 5.4 depict some of the Training and Readiness Manual Tasks that a unit is expected to demonstrate on R400. The majority of these R400 T&R tasks are at the 6000 level (company) or 5000 level (platoon or section). These are generally higher level tasks that are applicable to larger units. The performance standards listed within each T&R task reflect as such.

While these T&R tasks provide standards for the performance of larger military units, the first step for simulation developers should be to validate the actions of individuals and small units. Validating the behavior of automated forces is a difficult task. Fortunately, two methods are available to aid in this task. The first way to evaluate the performance of certain actions is with the T&R manual. After a high level task to be evaluated has been selected, one locates the task within the T&R manual and identifies this task's "chained events." Chained events are tasks which are related to the high level task, but are often performed by lower echelon military units. This process can be repeated until the evaluator locates a task which is to be performed by small units or individuals. These lower level tasks often provide detailed performance instructions or standards. When combined with the conceptual information provided by higher level T&R tasks, this technique can be a valuable tool for the validation of automated human behavior.

Infantry	
INF-MAN-6001	Conduct a Ground Attack
INF-MAN-6208	Conduct obstacle breaching
INF-MAN-6217	Employ Scout Snipers
INF-ASLT-5001	Provide Direct Fires
INF-ASLT-5002	Occupy Firing Positions
INF-MGUN-5001	Provide Offensive Fires
INF-MGUN-5002	Occupy Firing Positions
INF-MORT-5001	Provide Indirect Fires
INF-MORT-5002	Occupy a Mortar Position
Assault	
INF-ASLT-5001	Provide Direct Fires
INF-ASLT-5002	Occupy Firing Positions

Figure 5.3: Range 400 Training & Readiness Tasks 1, from [47]

Command and Control	
INF-C2-6007	Execute Command and Control of an Operation
Combat Service Support	
INF-CSS-6002	Process Casualties
Fire Support	
INF-FSPT-6006	Conduct Fire Support Team (FiST) Operations
Intelligence	
INF-INT-6002	Conduct Intelligence Collections
Engineer	
1371-MOBL-2025	Perform manual breaching
1371-MOBL-2016	Conduct obstacle breaching operations
1371-MOBL-1003	Breach explosive obstacles/hazards
1371-MOBL-2015	Employ the APOBS
1371-DEMO-2004	Compute the Net Explosive Weight (NEW)
1371-DEMO-2005	Take appropriate protective measures

Figure 5.4: Range 400 Training & Readiness Tasks 2, from [47]

The second method for evaluating the performance of units is with assessment documents such as the TTECG-produced *R400 Handbook*. Because of the frequency with which units conduct training at R400 and the serious nature of the evaluation, specialized assessment sheets have been produced for the TTECG exercise controllers. These assessment sheets organize the numerous aspects of R400 into a sensible manner and highlight the key concepts that a unit must display in order to be successful. TTECG describes these documents as "an extension of the T&R standard[s]" and while they are not official military doctrine, they provide valuable guidance for conducting a ground attack that encompasses "current and relevant techniques, procedures and best practices" [47, p. 19]. With this in mind, it is important to note that these assessment sheets should not only be used in the validation of automated tactical behavior, but should also be considered by simulation developers when they are developing tactical behaviors.

5.3 Description of a Typical Range 400 Exercise Force

The standard force to execute an attack on R400 is a reinforced rifle company. This organization consists of a full infantry rifle company, as well as attachments from different organizations

A standard rifle company is divided into several elements. The main subunits include three rifle platoons and one weapons platoon. Each rifle platoon is led by a platoon commander and a platoon sergeant; the rifle platoons are themselves subdivided into three rifle squads.

Rifle squads are led by a squad leader and are subdivided into three fire teams of four Marines each. The members of each fire team include a team leader, automatic rifleman, assistant automatic rifleman, and a rifleman. The automatic rifleman is an important member of the team because of the firepower he provides. If the automatic rifleman were to become a casualty, another team member would assume use of the automatic rifleman's primary weapon system at the earliest possible opportunity.

An infantry company's weapons platoon consists of three sections capable of employing 60mm mortars, rockets, and medium machine guns. The 60mm mortar section is responsible for the care and operation of three mortar systems. The section is led by a section leader and is comprised of three mortar squads, each of which contains three Marines. The mortar section usually operates as an independent unit in support of the company, however individual squads can be assigned to infantry platoons in order to conduct close range fire missions. The assault section is responsible for employing the shoulder-launched multipurpose assault weapon (SMAW), and conducting breaching operations through the use of various demolition charges. The section includes a total of 13 Marines and six SMAWs. They are led by a section leader and divided into three assault squads consisting of four Marines each. During combat operations, these units are usually attached to rifle platoons which have need of an assault squad's specialized capabilities. The medium machine gun section consists of 22 Marines and six M240B medium machine guns. They are led by a section leader and divided into three machine gun squads. Each squad consists of seven Marines, is led by a squad leader, and is subdivided into two machine gun teams. Each machine gun team includes a team leader, a gunner, and an ammunition man. Though these units can operate in a variety of ways, machine guns squads will rarely be split into separate teams.

One of the main goals of R400 is to test a company's ability to integrate its organic capabilities with numerous reinforcements. These non-organic forces typically include an engineer squad, a heavy machine gun section, a 81mm mortar section, a scout sniper squad, and several forward observers. The engineer squad typically consists of approximately seven Marines who specialize in counter-mobility and mobility (breaching) techniques. The heavy machine gun section consists of approximately seven Marines and employs either two M2 heavy machine guns, two Mk-19 automatic grenade launchers, or a combina-

tion of the two. Additionally, these systems are usually employed from the turret of a high mobility multipurpose wheeled vehicle (HMMWV). The 81mm mortar section consists of approximately 15 Marines and is responsible for employing four 81mm mortar systems. The scout sniper squad includes four individuals who are responsible for providing forward observation and precision fire capabilities to the infantry company.

The integration of all of these assets is the responsibility of the company commander, his headquarters staff, and his fire support team (FiST). The FiST is a special organization responsible for coordinating the employment of most of the specialized weapons units, and for deconflicting their fires from the maneuvering rifle platoons. The team usually consists of an artillery forward observer (FO), an 81mm mortar FO, a forward air controller (FAC), and several radio operators (ROs). The members of the team organic to the infantry company are the FiST Leader and his RO.

5.4 Building a Scripted Scenario of Range 400

With the simulation scenario selected, the next step is to build a COMBATXXI scenario that is representative of an actual live fire exercise. For scenarios involving human behavior, it is recommended to "create the information by hand first, then automate it later if necessary" [29, p. 43]. The reason for this is that creating a scripted scenario helps to identify key behavior aspects that actual military forces display on R400. These important behaviors will play a role in the eventual creation of automated behavior mechanisms. Also, by building the target scenario using the standard scripted approach, insights can be gained into the limitations of this approach. The following discussion is based on the actual implementation of the scenario using the scripting methodology.

For demonstration purposes, the size of both friendly and enemy forces will be scaled down. This step allows simulation developers to focus their attention on selected tactical actions. These actions will be recreated in COMBATXXI using standard development techniques. Difficulties in the scenario creation process and challenges to scenario reusability will be identified and addressed.

Marine forces within this scenario will consist of two infantry rifle squads and one infantry machine gun squad. The rifle squads will be referred to as first squad and second squad. The machine gun squad will be referred to as such. These forces will be attacking an enemy

fireteam that is defending the western trenches of R400. First the machine gun squad will move to a support by fire position on Machine Gun Hill, and then will begin suppressing enemy positions. The rifle squads will utilize this suppression to maneuver on the enemy positions. Order of movement will be second squad, followed by first squad. It should be noted that because of the elevation of Machine Gun Hill, infantry forces are permitted to cross underneath the machine gun fire. When second squad reaches the squad release point, they will establish a support by fire position from which they will suppress enemy positions. During this time period, the responsibility of suppressing enemy positions will pass from the machine gun squad to second squad. Once second squad has established an effective base of fire, first squad will continue maneuvering to its assault position which is just to the west of the enemy. Under the cover of second squad's support by fire position, first squad will assault through the enemy position, and continue to their limit of advance. When the limit of advance has been reached, first squad will consolidate in a defensive position to the north of the western trenches. Second squad and the machine gun squad will consolidate in a defensive position to the north of the enemy command post.

Chapter 4 discusses important principles which are utilized in the conduct of military operations. Unfortunately, the correct implementation of the simple combat operation described in the attack case study encompasses many complex tactical actions that are difficult to achieve through automated behavior algorithms. To scope the efforts of this thesis, the decision has been made to concentrate on replicating the security formations that military units use during the course of tactical operations.

5.5 Scenario Security Formations

Before commencement of an attack, military units conduct several standardized actions. One of the first considerations is security. If a unit is staging for an attack from inside a friendly base, then that base can be assumed to provide security. However, if the unit is staging outside of friendly lines, then a portion of the unit must be devoted to providing security. A 360 degree security perimeter is a standard formation used to accomplish this requirement.

While this may appear to be a relatively simple action, establishing 360 degree security requires actual military units to consider numerous factors regarding friendly capabilities,

enemy threats, and terrain characteristics. Accomplishing these tasks within a simulation is an even more complicated process. Within COMBATXXI, there is no standard command or template to use when creating a security formation. To create a security formation, the scenario developer must do so using COMBATXXI primitive orders, compound orders, and control measures. It is important to recognize that unit security is an ongoing task that must be accomplished before, during, and after tactical operations. Figure 5.5 and Figure 5.6 illustrate T&R manual standards for creating security formations. The following security formations were utilized in the R400 scenario:

Formation 1: Prior to the attack. 360 degree security formation comprised of the entire unit. Machine gun squad is integrated into the formation and they are oriented toward the enemy and along a high speed avenue of approach (a road).

Formation 2: Prior to the attack. 360 degree security formation in which most of the unit has been pulled into the center in order to conduct pre-combat checks and inspections. Designated members of the unit occupy guardian angel positions in order to provide 360 degree security for the unit.

Formation 3: During the attack. 360 degree security formation comprised solely of the machine gun squad. After the machine gun squad finishes their support by fire mission, they set up a small security formation at the base of Machine Gun Hill, and then prepare to relocate to their final consolidation position.

Formation 4: After the attack. 180 degree security formation consisting of an infantry squad. After assaulting through their objective and reaching their limit of advance, the first squad sets a 180 degree security perimeter that is oriented in the direction of a likely enemy counterattack.

Formation 5: After the attack. 180 degree security formation consisting of an infantry squad and the machine gun squad. After second squad has finished its support by fire mission, it moves to a location just north of the enemy command post. Once there, it sets up a 180 degree security perimeter that is oriented in the direction of a likely enemy counterattack. The machine gun squad has been designated to join second squad in this formation, however their arrival is delayed. In order to account for this, second squad positions its forces to cover the entire sector. Once the machine gun squad arrives at the position, they assume defensive positions that suit the capability of their primary weapon systems (firing along the long access of a high speed avenue

INF-MAN-4201: Conduct assembly area actions (D)

SUPPORTED MET(S): 2

EVALUATION-CODED: NO **SUSTAINMENT INTERVAL**: 6 months

CONDITION: Given a unit, an order, remote likelihood of enemy contact, and in preparation for follow on operations.

STANDARD: To accomplish the mission, meet the commander's intent, and prepare for follow on operations.

EVENT COMPONENTS:
1. Establish and improve all around security. (S)
2. Position automatic weapons on most likely avenues of enemy approach. (A)
3. Improve fields of fire, obstacles, fire support plan. (F)
4. Establish and improve positions/entrenchment. (E)
5. Conduct planning.
6. Conduct resupply.
7. Prepare for combat operations.

REFERENCES:
1. MCWP 3-11.1 Marine Rifle Company/ Platoon
2. MCWP 3-11.2 w ch1 Marine Rifle Squad

Figure 5.5: Pre-Operation Security T&R Task, from [46]

INF-MAN-4205: Consolidate (D)

SUPPORTED MET(S): 2

EVALUATION-CODED: NO **SUSTAINMENT INTERVAL**: 6 months

CONDITION: Given a unit, an order, and in preparation for follow on operations.

STANDARD: To enable preparation for combat while maintaining security, reorganizing the unit, and improving the current position.

EVENT COMPONENTS:
1. Establish and improve all around security. (S)
2. Execute command and control.
3. Displace or reposition elements as required.
4. Position automatic weapons on most likely avenues of enemy approach. (A)
5. Process ammunition, casualty, and equipment (ACE) reports.
6. Redistribute ammunition, personnel, supplies, and equipment.
7. Conduct CASEVAC/MEDEVAC as required.
8. Detain/process detainees as required.
9. Conduct information collections.
10. Improve fields of fire/sectors of fire, obstacles, etc. (F)
11. Establish/improve positions/entrenchment. (E)
12. Prepare for follow-on operations.

Figure 5.6: Post-Operation Security T&R Task, from [46]

of approach). Additional members of second squad assume an alternate position (flank security).

While it appears to be a relatively simple process, the creation of a security formation encompasses numerous factors that do not easily translate from an actual Marine in the field, to an entity in a simulation. These factors include analyzing the terrain in which a unit is situated, the unit's mission and capabilities, and the perceived enemy threat. The following sections describe the steps required to create a security formation within COMBATXXI.

5.6 Security Formation Creation Process

When creating a security formation for COMBATXXI entities, a circle formation (or a formation that is close to a circle) that provides 360 degree coverage is a logical place to start. Assigning sectors of responsibility is one of the first issues that the scenario developer encounters. To accomplish this task, the scenario developer must consider the overall size of the unit, the number of subunits, and the number of personnel within each subunit. If all of the subunits have the same number of personnel and contain the same capabilities, the area of responsibility for each subunit can be found by dividing 360 by the number of subunits. This is a simplifying assumption that can be made initially, however the scenario developer must remember that this will not always be the case.

The next step is determining the overall size of the security formation. This step is dependent upon the number of personnel participating in the security formation and the dispersion between each entity. A standard dispersion amount can be found in military publications, however the scenario developer must consider special circumstances that could cause a unit to alter its dispersion. Heavily forested terrain that limits visibility can cause a unit to lower it dispersion, as can missions requiring a unit to avoid detection by enemy forces. Alternatively, open terrain such as a desert, or the existence of a heavy enemy indirect fire threat could cause a unit to use an increased amount of dispersion.

Once subunit area of responsibility and overall formation size has been determined, the scenario developer can create waypoints which represent each subunit's location. To get each subunit into its proper position, a compound order must be created for each of these waypoints. Compound orders are comprised of one or more primitive orders. Commonly used primitive orders include the following:

- MOVE_TO_POINT
- CHANGE_FORMATION
- CHANGE_POSTURE
- CHANGE_ORIENTATION
- STOP_MOVING.

It is the scenario developer's responsibility to create a sequence of primitive orders which will ultimately place a subunit in the correct position. One of the first challenges is getting

a unit to assume a proper formation. The CHANGE_FORMATION primitive order allows the scenario developer to choose from numerous standardized formations which can result in a reasonable overall security formation. However, if the scenario developer wants each subunit to maintain the appropriate amount of dispersion, and for each subunit's flanks to be linked to the adjacent subunit's flanks, the scenario developer will likely have to create a new formation. This is accomplished by using the Scenario Integration Tool Suite (SITS) formation editor. This tool allows the scenario developer to specify the number, location, and orientation of entity positions within a formation.

Once a proper formation has been designated, the next task involves moving the unit to its intended location, getting the formation oriented properly, and getting individual entities oriented in the proper direction. This is accomplished by adding more primitive orders to the subunit's compound order. The simplest way to accomplish this (after using the CHANGE_FORMATION command) is to move the subunit to their location using MOVE_TO_POINT, order the unit to halt with STOP_MOVING, and then getting the unit to look in the proper direction by using CHANGE_ORIENTATION. This is a simple set of commands which can be easily created, however several problems exist. First, even though the individual members of the subunit will be facing in the proper direction, the overall orientation of the subunit's formation is likely incorrect. This means that instead of having a line of entities which stretches from east to west, the simulation could easily create a column of entities which stretch from north to south. This mistake could seriously impact the subunit's mission effectiveness by limiting the number of entities able to engage enemy forces (especially if the subunit is positioned on rolling terrain) and by making the subunit more vulnerable to enemy fire (machine gunners prefer to engage targets by firing along the long axis of a column). One method devised to avoid this problem involves the creation of an intermediate waypoint that the subunit must pass through prior to arriving at its final position. This intermediate waypoint should be positioned so that the direction from this point to the final position corresponds to the desired orientation of the subunit's formation. An example of this technique for a semi-circular formation is illustrated in Figure 5.7.

Though this technique complicates many of the compound orders that must be created, it also offers simple solutions to two additional problems. The first problem is that when units arrive at their final destination, they are often out of formation. As discussed earlier, forma-

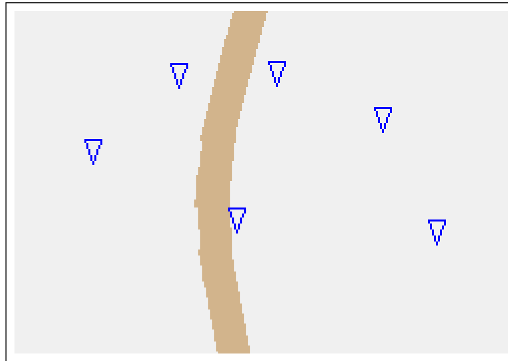


Figure 5.7: Waypoints Needed for a Scripted Security Formation

tion problems can limit a unit's effectiveness or make them more vulnerable to enemy fire. This problem can be avoided by adding using the CHANGE_FORMATION command after the subunit moves to the intermediate waypoint. This is useful even if the subunit is already in the desired formation because it forces the subunit to correct formation deficiencies one last time before they move to their final position. The next problem that the intermediate waypoint technique helps with is the orientation of the subunit entities. This step can be accomplished by using the CHANGE_ORIENTATION command. However this technique forces the scenario designer to calculate and manually input the appropriate orientations for every subunit within a formation. This can be a time intensive process if a unit contains many subunits. Additionally, the work must be repeated for each subunit if the overall unit's orientation or composition changes. By using the intermediate waypoint technique, the scenario developer can avoid using the CHANGE_ORIENTATION command as long as the direction from the intermediate waypoint to the final position corresponds to the direction that the entities should be observing. This saves time by simplifying the compound orders and enables the scenario developer to quickly adjust the intended orientation of the forces by moving the waypoint locations within SITS.

Once the compound orders for each security position have been created, the actions they describe are enacted through coordinated maneuver orders and behaviors. This provides a relatively straightforward approach for creating a basic security formation. If a unit is composed of subunits, this may require the creation of numerous compound orders and behaviors. The COMBATXXI Maneuver Tool can help manage the complexity involved

in invoking these orders, however they still must be individually produced by scenario developers. If a scenario developer desires to ensure the suitability of such formations in more complicated situations, additional factors need to be considered.

5.7 Advanced Considerations for Security Formations

Creating security formations within COMBATXXI that are representative of actual military security formations requires the scenario developer to consider several additional factors, and then modify the basic security formation accordingly. For short security halts, subunits will quickly be assigned sectors of responsibility that are appropriate to the subunit's size. If a unit intends to occupy a location for a longer period of time, security positions will be tailored to match the combat capabilities of each subunit. One way this is accomplished is by adjusting sectors of responsibility to match each subunit's capabilities. Within standard security formations, standard riflemen can be expected to spread out and cover the sector of fire that has been assigned to their fire team. This is not the case for special units such as medium machine gun teams. The primary mission of the machine gun team's members is to ensure the effective employment of their machine gun. This involves a different formation than that practiced by standard riflemen. For a medium machine gun team, this typically means that the gunner and the team leader will be co-located at the machine gun. The third person of the team, the ammunition man, will cover an alternate sector of fire in order to protect the gunner and the team leader. As long as there are riflemen nearby to support the machine gun team, the ammunition man will not be the only person assigned to cover this sector of fire. This is important because during an engagement, the ammunition man must be free to move in order to prepare ammunition boxes, or to assume an alternate role if his team suffers casualties. Finally, machine gun teams will typically operate in pairs, and will be called a machine gun squad. This task organization increases the chances that at least one team will be able to provide suppressive fires in support of an assigned mission. This technique was utilized in several of the scenario's security formations.

The previous paragraph describes how different types of subunits fit together in a security formation. The next major requirement involves deciding how to position subunits within a formation, and in what concentrations. The unit mission, enemy threat, and terrain characteristics must be considered in order to accomplish this task. A unit's mission usually includes an assigned sector of responsibility. This sector could range in size from a very

narrow sector assigned to an individual entity (such as in an urban environment), to a 360 degree perimeter that an entire unit must cover. Within such a sector, subunits will often position themselves in locations that maximize their primary weapon system's effectiveness, and provide the most resistance to enemy threats. Examples of this include placing assault units with rocket launchers near roads that enemy vehicles may utilize, and placing machine gunners in locations that allow them to engage targets at extended ranges or to employ grazing fire techniques (machine gun fire along flat terrain that is no more than one meter above ground). This technique was accomplished in Formation 5 by placing the machine gun squad along the long axis of a road that could serve as a high speed enemy avenue of approach. The desire to properly position more powerful weapons systems is also applicable to the automatic rifleman within a standard infantry fire team. By doctrine, all members of a fire team are supposed to be assigned a single sector of fire (note this is not always possible due to personnel shortages). Within a fire team's assigned area, the automatic rifleman must be positioned so that he can cover the team's entire sector of fire. This is an important step within military operations, however this action was not programmed within the scripted scenario.

The unit's mission and the terrain in which they are situated are major factors in how entities are positioned. One complaint of AI controlled entities is that they will position themselves next to a wall or terrain feature that severely limits their observation capabilities. This is often cited as a lack of realism in games and simulations. To overcome this deficiency, the scenario developer must consider the amount of observation that a unit requires to execute its mission, and then locate suitable positions within available terrain. Some reconnaissance or forward observer units will want to be able to see as far as possible. Regular combat units usually desire to see as far as their weapons systems can effectively engage enemy forces. At other times, units will want to ensure the enemy does not observe their forces and will therefore remain behind a terrain feature, even if it limits their ability to employ their weapons systems. To accomplish any of these goals, the scenario developer must place a waypoint at a location on the map, and then use the SITS LOS tool to ensure the location provides appropriate observation. This step was accomplished for every unit location in the scenario.

One final consideration concerns the actual number of entities that are actively providing

security, and the state of these members. It is important to recognize that not every member in a unit will hold a position on the outer edge of the security formation. This is especially true for unit leaders and mortarmen. In most circumstances, these entity types will establish their own positions inside the formation that allow them to perform their leadership responsibilities or allow them to effectively employ their primary weapon systems. This was illustrated in Formation 2 when a portion of the unit was brought inside the security formation so that they could conduct their attack preparations, while another portion of the unit continued to maintain 360 degree security. The state of the entities on the formation must also be considered. When Marines occupy security positions, they generally assume the prone position in order to limit their exposure to enemy fire. Within COMBATXXI, this can be accomplished by using the CHANGE_POSTURE primitive order within every subunit's compound order for movement. For additional realism, one should recognize that when a unit is set in a fighting position for an extended amount of time, not every member will be actively observing. General readiness levels for units occupying defensive positions for an extended amount of time are 50% during the day and 25% at night. This aspect could be represented in COMBATXXI by turning off the sensors of specified entities.

5.8 Challenges to Overcome

The previous instructions and guidance enable a scenario developer to create security formations which are fairly representative of what takes place in actual combat. Potential critiques of the example scenario include the large number of compound orders and waypoints which must be created and positioned in order to achieve this behavior. One method that attempts to avoid this problem is changing the fidelity of the subunits. Within this scenario, subunits were created to represent each fire team; the number of compound orders and waypoints could be reduced by designating the squad as the lowest subunit in the simulation. Unfortunately this approach has two disadvantages. First, the scenario developer must now create additional squad formations that will place entities into their proper security formation positions. Depending on the scenario, this may or may not reduce the overall scenario designer workload. A second problem is that this technique does not scale well. Within scenarios with large number of units, scenario designers using squads as the lowest subunit will face the same level of complexity as within the example scenario which used fire teams as the lowest subunit.

The final and potentially largest problem concerns how a unit deals with casualties. If a security formation is created in the middle of a scenario, the unit must be able to account for casualties taken in previous engagements, and adjust the security formation so that no gaps exist. Limited functionality exists to deal with this problem. Within the example scenario, if a fire team is destroyed, subsequent unit security formations will contain gaps. As mentioned before, one potential solution calls for making the smallest subunit to be a squad instead of a fire team. By prioritizing the positions within a formation, the scenario developer can help prevent major gaps from appearing in a unit's overall formation. Unfortunately this does not account for the changing priorities that a unit will experience as its numbers and capabilities change. While feasible for the scenario developer to create multiple formation that would enable effective operations for units of different sizes, this has the potential to create an extremely large amount of work. Furthermore, this solution does not scale to large levels; all of these efforts would have to be repeated if the lowest subunit were changed to a platoon instead of a squad.

These problems lead to one of the fundamental issues that must be addressed: the ability to organize subunits so that their compositions are appropriate to the situation, and the ability to assign each subunit a task that is appropriate to its capabilities. Examples of programs displaying this functionality were illustrated in Chapter 3. These examples will be used as inspiration during the development of dynamic behavior capabilities for use in COMBATXXI.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 6: Dynamic Behavior Development and Demonstrations

COMBATXXI is a robust simulation capable of providing insights into military operations. However, scenario designers face several challenges in the areas of scenario creation and reusability. Chapter 5 described the behavior definition process for a standard COMBATXXI scenario. For this scenario, a large amount of work was required to create relatively basic infantry actions. The purpose of this chapter is to document efforts to ease the COMBATXXI behavior definition process by leveraging the new dynamic behavior methodology developed at the Naval Postgraduate School (NPS).

6.1 Scoping a Behavior to Develop

The first problem concerns the large amount of effort required to create a sequence of coordinated tactical events which are representative of actions conducted in actual combat. The initial goal of this thesis was to produce a scalable and reusable HTN that was capable of coordinating a simple infantry attack similar to the one described in Chapter 4. However, a thorough task decomposition of the proposed attack behavior reveals that even relatively simple actions are almost always composed of numerous pieces. The most basic deliberate attack can be divided into the following phases, each of which involves several sub-behaviors.

1. Movement to Assembly Area
2. Movement from Assembly Area to Attack Position
3. Movement from Attack Position To Assault Position
4. Actions on the Objective
5. Consolidation

Creating a representative attack behavior requires its associated sub-behaviors to be executed at the correct time, at the correct location, and in the correct manner. This is a task that requires a planning process capable of executing appropriate decision making based on a unit's situational awareness and its task organization. Inspiration for how such a planning system could be developed is described in the Chapter 3 case studies.

An important prerequisite for a planning system is to correctly model the sub-behaviors required in the execution of that action. To be useful by an automated planner, these sub-behaviors themselves must be scalable and reusable. One of the impediments to the rapid creation of scenarios within COMBATXXI is that many of the sub-behaviors do not exist. This requires that for each scenario, developers must create each sub-behavior from existing primitive behaviors. Because such actions are taken with specific scenarios in mind, this results in the production of scripted behaviors that are rarely scalable or reusable. This thesis begins the process of correcting this deficiency by constructing scalable and reusable sub-behaviors necessary for the conduct of a basic attack.

6.2 Security Formation Behavior

Establishing a static security formation was the first behavior chosen for development. This behavior was selected because it is one that is frequently used by ground combat forces. Major places to employ security formations are at the beginning and end of an operation. In practice, security formations will also be employed periodically during other phases of an attack. Examples of this include having a subunit hold security while the remainder of the force crosses a danger area, conducting a hasty security halt so that subunits can make final preparations, and finally by tasking a subunit to hold flank security so that enemy forces do not interfere with the assault element.

The establishment of a scalable and reusable behavior which is capable of creating a security formation serves two purposes. First, this behavior would play an important role for AI planning systems that have been tasked with creating an attack behavior. Second, this behavior could be utilized to ease scenario creation within scripted scenarios such as that described in Chapter 5.

Devising a scenario-specific security formation for a simulated military unit that is facing a known threat and is surrounded by a set type of terrain is often a straightforward task. Creating a generalized behavior capable of devising security formations that cover a wide range of friendly, enemy, and terrain situations is a much more difficult task. Initial efforts to create such a behavior involved the use of four core HTNs. Figure 6.1 illustrates the relationship between these HTNs.

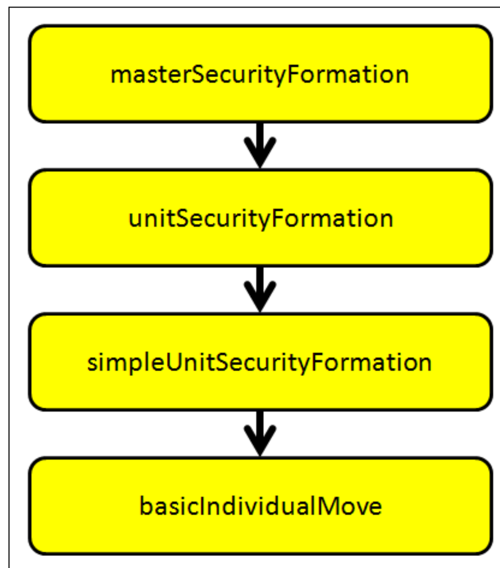


Figure 6.1: Current Hierarchy of Security Formation HTNs

6.2.1 masterSecurityFormation

The first and arguably most important HTN utilized is *masterSecurityFormation*. This HTN is responsible for receiving security formation parameters and then calculating the formation's approximate location, geometry, and orientation. While this process itself can be considered a simple geometry problem, the way that *masterSecurityFormation* approaches this problem is unique and therefore should receive special consideration. First, the HTN parameters include a list of units that are to take part in the security formation. This approach enables multiple units to cooperate with each other and ultimately create a single tactically sound security formation.

The second item of note concerns the fact that this HTN calculates preliminary locations for individual entities. This step is necessary in order to determine the overall formation geometry, and to ensure that the formation accomplishes the mission described within its assigned parameters. However, it is important to recognize that *masterSecurityFormation* never assigns individual entities to occupy specific locations. This task is accomplished by other HTNs. While this approach may seem artificial, it should be recognized that this process has parallels to actual military techniques. During reconnaissance operations, military leaders often select preliminary individual positions so that when the rest of the

unit arrives, defensive positions can be occupied in a timely manner.

masterSecurityFormation finishes by activating *unitSecurityFormation*. It does this by assigning *unitSecurityFormation* as a goal for an individual entity in the unit responsible for creating the security formation. The primary parameter for this HTN consists of a list of individual security positions that the unit has been tasked to fill. If multiple units have been tasked with creating one security formation, there is a requirement to assign separate lists to each unit. However, because this step is identical to one conducted within *unitSecurityFormation*, it is discussed in the following subsection.

6.2.2 unitSecurityFormation

Once the formation's general parameters have been defined, *unitSecurityFormation* is assigned to a single unit. It fulfills two important steps. First, it identifies the main unit's existing subunits (if any exist) and then analyzes their basic properties. Based upon this analysis, *unitSecurityFormation* then determines an appropriate number of individual security positions for each subunit to occupy. As with the previously described HTN, *unitSecurityFormation* does not give orders to individual entities. Instead, the HTN gives generalized orders to members of each subunit, and then "trusts" each of those individuals to create subunit-specific detailed plans which contribute to the accomplishment of the overall mission. This is a technique which is not only consistent with military procedures, but is also a key aspect to HTN reusability and scalability.

The next important part of this HTN concerns how tasks are delivered to subunits. This process is carried out in two ways. Which method is utilized depends upon whether or not a subunit can be broken down into smaller subunits. Complex subunits are those units which themselves consist of multiple subunits. Complex subunits receive their tasks through *unitSecurityFormation*. If a subunit does not contain further subunits, it is considered homogeneous or "simple." Simple subunits are assigned their tasks through *simpleUnitSecurityFormation*. As with *unitSecurityFormation*, the main parameter for *simpleUnitSecurityFormation* consists of a list of individual security positions that the simple unit has been tasked to occupy.

6.2.3 simpleUnitSecurityFormation and basicIndividualMove

The final process of completing a security formation is accomplished through a combination of two HTNs. The first of these HTNs is *simpleUnitSecurityFormation*. Parameters for this HTN include a list of security positions for which the unit is responsible. Unlike the previously described HTNs, *simpleUnitSecurityFormation* makes the significant step of assigning individual entities to occupy specific security positions. Once these assignments are created, *simpleUnitSecurityFormation* assigns the *basicIndividualMove* HTN to each entity in the unit that is to take part in the security formation. *basicIndividualMove* is assigned to individual entities and its parameters encompass instructions on how to occupy a specific security position. *basicIndividualMove* is significant because unlike its predecessors, this HTN causes simulation entities to take action. It is also important to recognize that *basicIndividualMove* was designed to aid this study's demonstrations, and that it does not incorporate many of the actions a real combatant would perform when moving about a battlefield. The modular nature of the security formation structure allows *basicIndividualMove* to be easily replaced by a more representative movement control mechanism. This topic is discussed further in Chapter 7.

6.2.4 Security Formation Examples

Working together, the security formation HTNs provide scenario developers an intuitive and flexible tool for accomplishing a tactical task that is fundamental to military operations. Figure 6.2 illustrates some of the security formation geometries that can be created using the exact same set of unmodified HTNs with different sets of controlling parameters. Several additional capabilities should also be noted:

- Dispersion between entities can be adjusted to suit the tactical situation.
- Multiple units can work together to create one cohesive security formation.
- Vehicles and entities with unique capabilities can be integrated into security formations.
- Unlike the security formations described in Chapter 5, each formation only requires a single waypoint to mark its location.
- Waypoints can designate either the center of the formation (center and right examples in Figure 6.2) or the forward line of troops (FLOT) (left example in Figure 6.2).

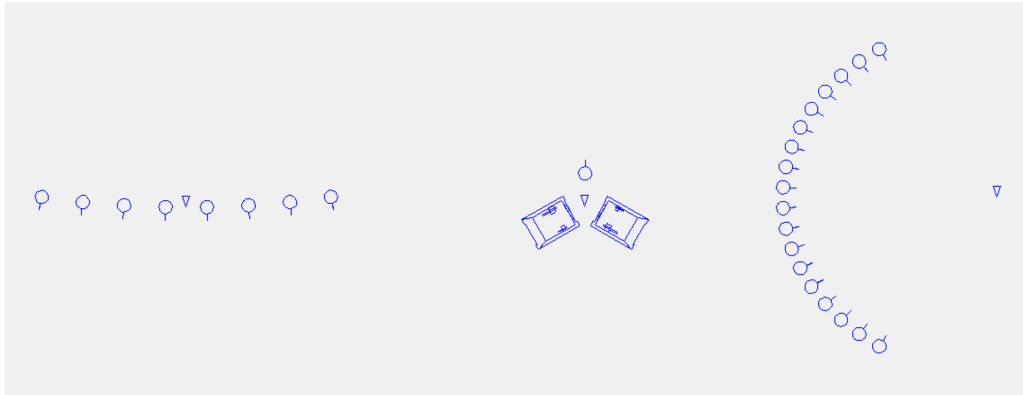


Figure 6.2: Three Examples of Security Formations

Each of the core HTNs fulfills an important role in accomplishing the security formation. The question could be raised as to why some of these functions are not combined into a smaller number of HTNs. Keeping these functions separate makes the behaviors modular. This plays an important role in the behavior's reusability and scalability. More importantly, it eases the process of adding additional capabilities to the security formation behavior.

6.3 Affordances

Once the basic security formation behavior had been created, the next step involved demonstrating its use in tactical scenarios. As described in Chapter 3, embedding information into terrain through the use of affordances is a behavior control technique that is widely used in the gaming industry. This technique holds many advantages when it comes to producing basic tactical behaviors; therefore, it was decided to develop such a capability within COMBATXXI.

Affordances, as implemented for this thesis, are markers embedded into the terrain that contain information relevant to that specific location, that entities can use. Since the information may be specific to a side or constrained in some other way, an access mechanism controls which entities may access the information or even be aware of the existence of the affordance. These constraints are dynamic and can change over the course of the simulation run. Details of affordance functionality can be found in Appendix A. The following test scenarios were produced to demonstrate the capabilities of the security formation behavior and the affordance mechanism.

6.4 Urban Patrol Demonstration

The first test of this thesis' products offered a chance to demonstrate both security formation and affordance capabilities. Specifically, this example highlights the diverse ways that procedural affordances can be utilized within a scenario. The selected scenario involved an infantry Platoon Commander and two infantry squads maneuvering through an urban environment. The patrol route is depicted in Figure 6.3. This scenario also made use of an existing HTN called *SquadMove*.

6.4.1 Demonstration Overview

The scenario begins when the Platoon Commander conducts an unaccompanied patrol through the urban environment. This is accomplished through the application of three *SquadMove* HTNs that are instantiated in the scenario's *jump_start* file. It should be noted that these movements are the only scenario behaviors activated through the normal application of HTNs. All remaining scenario behaviors are triggered through the use of procedural affordances. This approach is not necessarily the preferred way of creating a COMBATXXI scenario, however it was constructed in this manner so as to demonstrate affordance capabilities.

As the Platoon Commander moves along the route, he passes within range of 16 procedural affordances. Until he reaches the final checkpoint, the Platoon Commander is denied access to all of these affordances due to the access constraints placed on them. When the Platoon Commander reaches the final checkpoint, he activates a series of procedural affordances that cause the two infantry squads to take action. First, the squads form a single security formation around the initial waypoint. After holding this formation for a designated amount of time, one of the squads assembles in the middle of the formation and makes preparations to begin a patrol. When this happens, the other squad readjusts its members so that 360 degree security is maintained. This process is depicted in Figure 6.4.

Next, the squads begin to bound through the patrol route. When the lead squad reaches a new waypoint, it establishes one of many security formation types available with *masterSecurityFormation*. Once this position is set, the rear squad breaks down its security formation, bounds through the lead squad's position, and establishes a new security formation at the next waypoint. Figure 6.5 illustrates an example of this behavior.

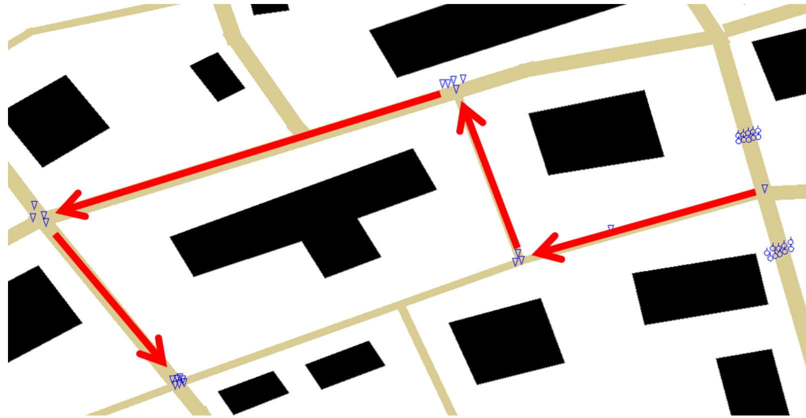


Figure 6.3: Patrol Route and Affordances

This process is continued until one of the squads establishes a security formation at the final checkpoint. At this time, the rear squad also moves to the final checkpoint where it integrates into the existing security formation. This behavior is shown in Figure 6.6.

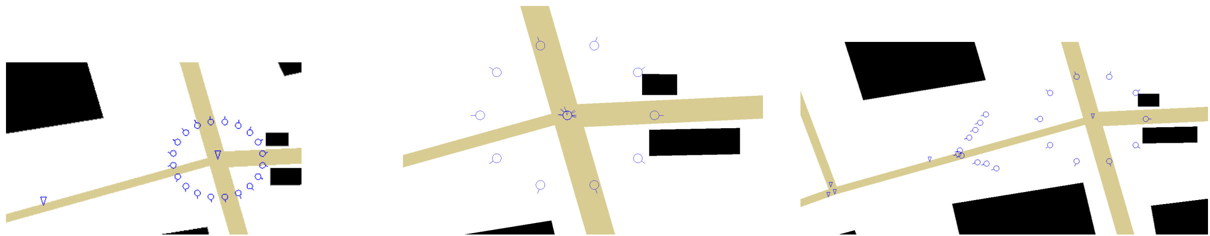


Figure 6.4: Removing a Unit from a Security Formation (From Left to Right)

6.4.2 Demonstration Features to Note

Several behaviors in this demonstration deserve special attention. Many of these actions are possible because of higher level HTNs designed to modify the behavior of the primary HTNs.

- Some of the affordances trigger HTNs that are executed by the unit activating the affordance; other affordance HTNs are executed by a unit that is not the activating unit. These actions were accomplished through the use of higher level HTNs that were designed to work with the existing *SquadMove* and *masterSecurityFormation* HTNs. Examples of these "adapter" HTNs include *securityFormation_byProcessingUnit*,

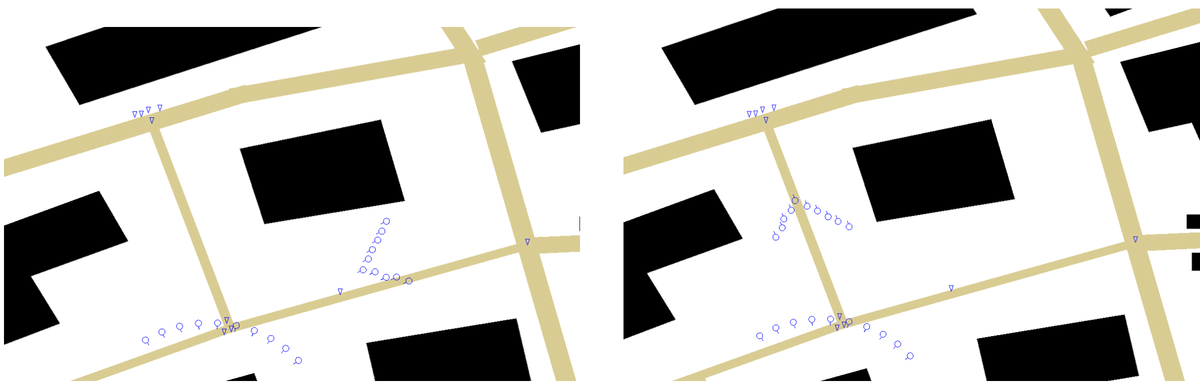


Figure 6.5: Bounding Past a Stationary Squad

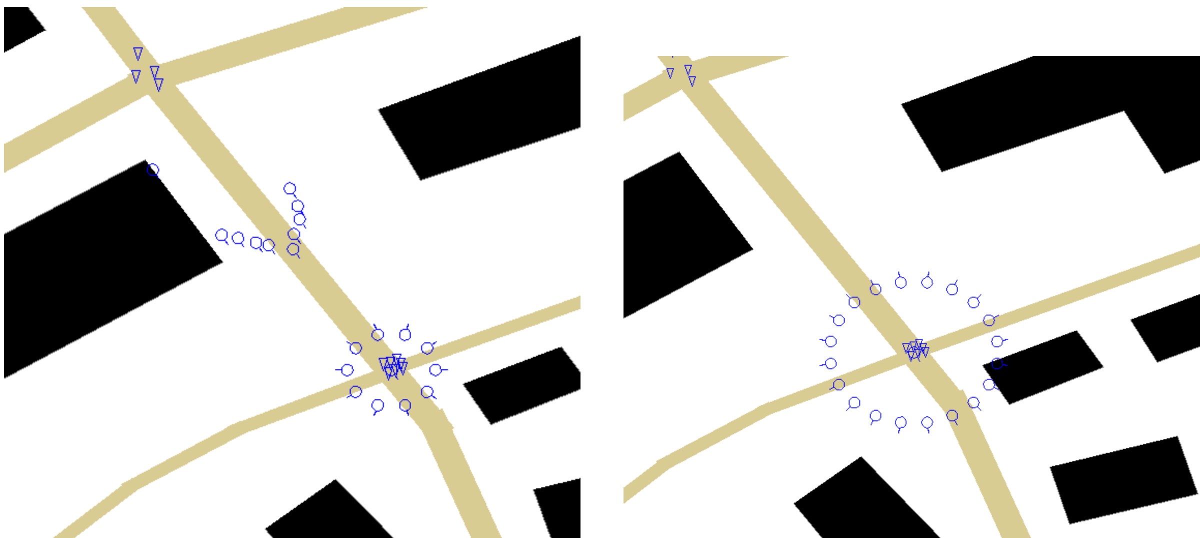


Figure 6.6: Reinforcing an Existing Security Formation

SquadMove_MoveDesignatedUnits, and *SquadMove_OtherUnitsInArrayList*. Figure 6.7 depicts the use of this technique.

- Removing a unit from an existing security formation is accomplished through *removeUnitFromAffSecForm* which works in conjunction with a procedural affordance that utilizes *masterSecurityFormation*.
- Adding a unit to an existing security formation bears many similarities to the process of removing a unit. The HTN *securityFormation_byPhaseAffordanceMod* accomplishes this for arriving units by updating HTN parameters stored in *affordanceUtilities*.

- The same scenario executes regardless of which squad is ordered to begin the patrol. This capability is made possible because both squads can access the maneuver related affordances.
- The act of bounding one squad through the other requires two affordances in the same location. The lead squad must access the security formation affordance while the following squad must access a *SquadMove* affordance. It is important that individual squads are not able to access both HTNs. This is accomplished by selecting appropriate valid times for TEMPORARY affordances or by utilizing ONESHOT affordances. This can also be accomplished by manipulating affordanceValidBoolean values through the use of an additional *changeAffordanceValidBoolean* affordance.
- Upon completing his patrol, the Platoon Commander entity accesses an affordance that brings his *processAffordances* HTN to a goal state, and stops him from attempting to process affordances. In this example, it was important that this process not take place until several other affordances were executed. This was accomplished by placing a large time delay on *stopProcessAffordancesHTN*'s execution.

6.4.3 Demonstration Findings

Creating this initial demonstration was an important process that drove further refinement of the existing security formation and affordance structures. The ability to access and update stored HTN parameters and the flexibility provided by the "adapter" HTNs were both unexpected benefits made possible through the use of procedural affordances. However, with these additional capabilities, also come several areas for caution.

The first area concerns behavior complexity. Layering multiple behaviors or HTNs on top of one another is a technique that can increase reusability. However, because HTNs are slower than standard scripted behaviors, simulation developers must ensure that each additional level of complexity serves a purpose. In situations like this when multiple HTNs are employed to achieve a single behavior, utilizing a clear naming convention is an important consideration.

The next area for caution concerns how procedural affordances are employed. Scenario developers must be aware that PERMANENT affordances can cause problems if accessed by too many entities, and that ONESHOT affordances can create issues if accessed by an

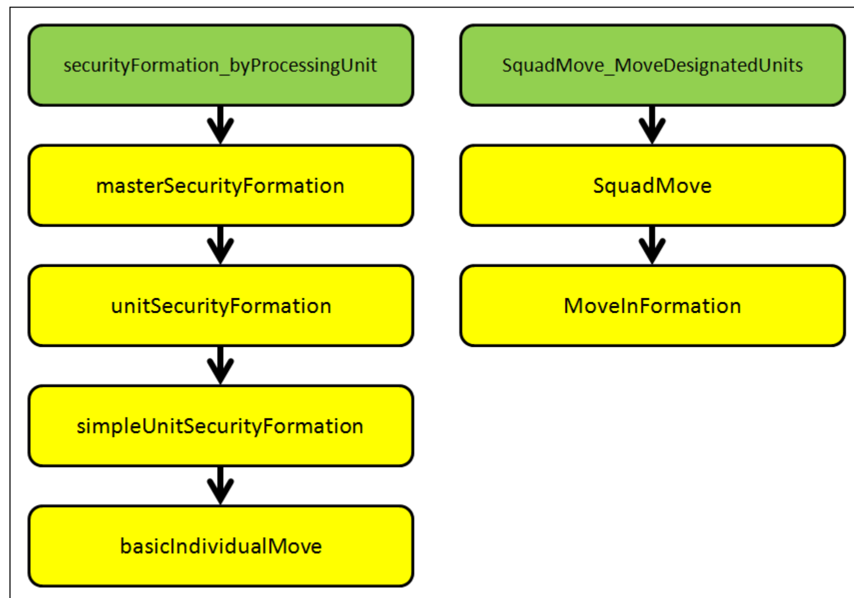


Figure 6.7: Using Adapter HTNs to Modify the Functionality of Core HTNs

unintended entity. All classifications of procedural affordances can display problems if they attempt to execute their associated HTN before a previous behavior has been completed. While these issues are potential pitfalls to the efficient use of affordances and HTNs, the affordance parameters were designed to let scenario developers tailor their behaviors to be successful in any situation. Understanding both the capabilities and limitations in this demonstration promises to greatly ease future COMBATXXI scenario development.

6.5 Dynamic Range 400 Demonstration

The next demonstration of the security formation HTN and affordance capabilities is provided through a modification of the R400 scenario described in Chapter 5. This scenario demonstrates how dynamic behaviors lower scenario creation time and enable entity behaviors which are more representative of actual military tactics. Specifically, the ability of the security formation HTN to handle casualties is highlighted. Finally, the scenario provides additional insights into ways that these tools can be improved.

6.5.1 Reduction of COMBATXXI Behavior Mechanisms

The process of building the updated R400 scenario primarily involved removing scripted aspects of the original R400 scenario, and replacing them with HTNs and procedural affordances. A total of two HTNs and six procedural affordances were utilized to create the behaviors in the dynamic scenario. The application of these tools immediately resulted in a reduction of waypoints, compound orders, and behaviors. The reduction of these mechanisms is shown in Figure 6.8. This benefit is primarily due to the fact that HTNs and affordances can be used to control the actions of multiple units. This aspect of HTNs and affordances could significantly lower the development time required for larger COMBATXXI scenarios. Using these tools to control the actions of multiple units has additional benefits. When scenarios must be modified, this design approach reduces the number of scenario mechanisms which must be changed. This advantage directly affects a scenario's reusability.

6.5.2 Efficient Behavior Development and Increased Reusability

The next advantage of using dynamic behaviors is illustrated through their ability to create tactically sound security formations. Figure 6.9, Figure 6.10, and Figure 6.11 compare security formations conducted in both versions of the R400 scenario. The security formations from the dynamic R400 scenario contain several advantages over their scripted counterparts. These advantages include properly oriented entities, appropriate spacing between entities, and fewer COMBATXXI waypoints.

It is important to differentiate between behaviors that are technically feasible through the use of traditional scripted methods, and behaviors that are not feasible. In regards to the former type of behavior, it is possible to create scripted security formations that look as

COMBATXXI Tools	Scripted Scenario	Dynamic Scenario
Waypoints	72	58
Compound Orders	32	12
Behaviors	37	15
HTNs	0	2
Procedural Affordances	0	6

Figure 6.8: Comparison of Required Behavior Mechanisms for Scripted and Dynamic Scenarios

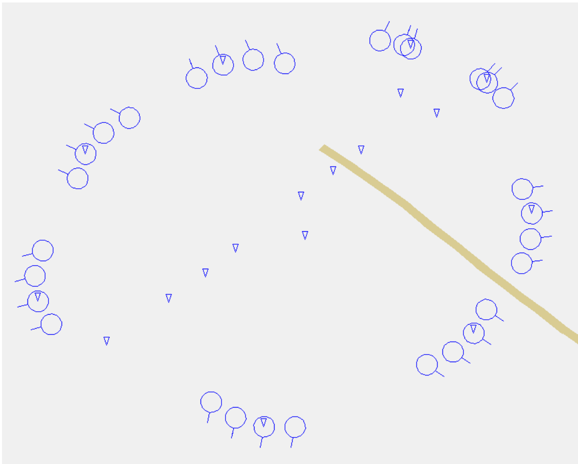
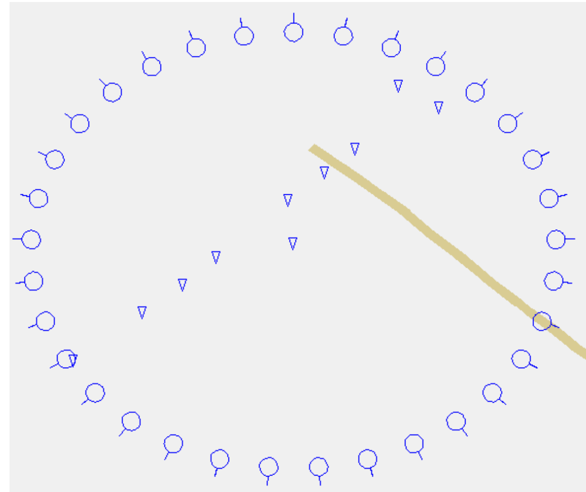



Figure 6.9: Unit Security Formation - Scripted (Left) and Dynamic (Right)

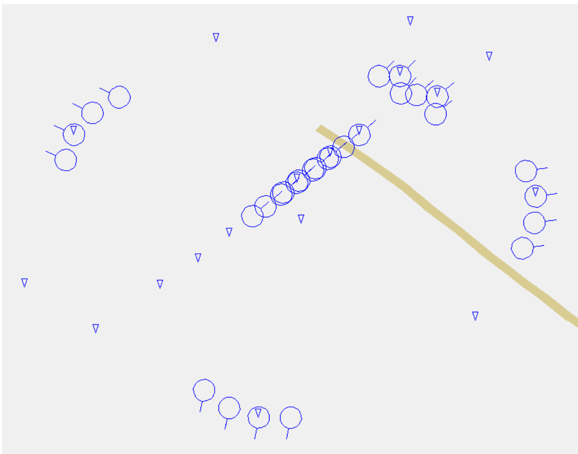
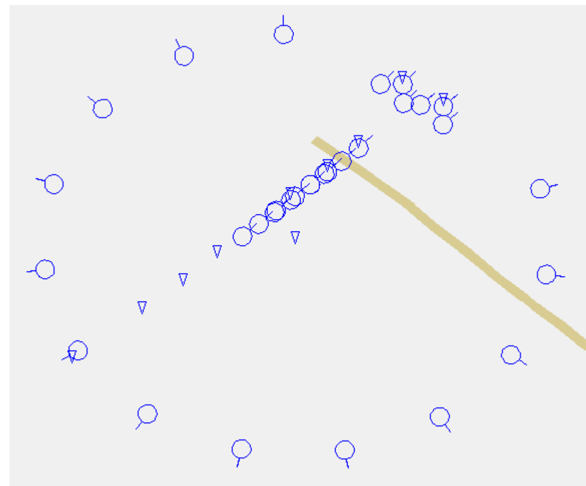



Figure 6.10: Guardian Angel Security Formation - Scripted (Left) and Dynamic (Right)

good as their dynamic counterparts. However, this would require a large increase in the number of waypoints, compound orders, behaviors, and scenario specific formations. Furthermore, the work that was invested in such a process would not be reusable; it would have to be repeated if the requirements for the security formation changed.

This complicated process of creating security formations through scripted techniques is why this fundamental military action is left out of many military simulations. The dynamically created security formations took significantly less effort to create than their scripted

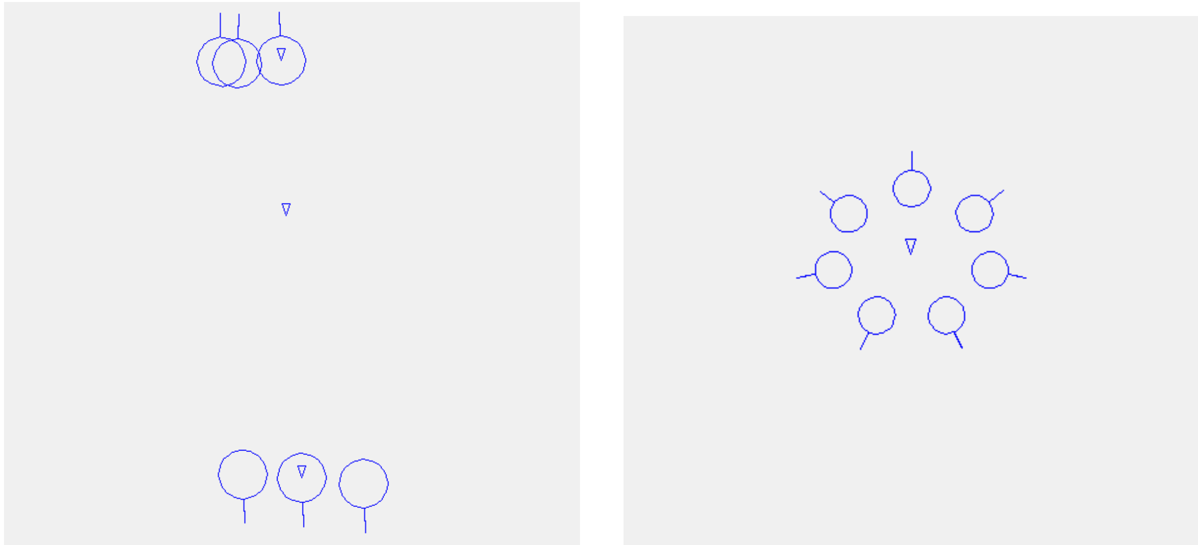


Figure 6.11: Small Unit 360 Security Formation - Scripted (Left) and Dynamic (Right)

counterparts. Additionally, these dynamic security formations can have their geometries quickly adjusted, can change to accommodate units with different numbers of entities, and can be reused in different situations.

6.5.3 Additional Capabilities

Saving time and resources are important benefits of utilizing dynamic behaviors, however, it is even more important to recognize the things that dynamic behaviors can accomplish but their scripted equivalents cannot. Figure 6.12 displays a standard military security formation that a military unit is likely to conduct after seizing an objective. The dynamic security formation behavior does a better job of properly spacing and orienting each entity. As previously noted, the scripted example could reproduce these results if enough time were invested. What the scripted formation cannot deal with are the effects of battlefield casualties. Figure 6.13 illustrates what the formations from Figure 6.12 look like when a unit incurred casualties during an operation.

The scripted security formation in Figure 6.13 contains large gaps that were not present in the example with no casualties. Alternatively, the dynamic security formation in Figure 6.13 simply looks smaller than its no casualty counterpart. The reason for this discrepancy is that in the scripted example, each entity's exact position has been predetermined.

When units are attrited during the course of an operation, the scripted security formation behavior has no mechanisms for identifying the entities that became casualties, and readjusting the security formation. Gaps ultimately appear in the scripted security formation because the scripted behavior functions as if the unit incurred no casualties.

Dynamic security formations created with *masterSecurityFormation* do not suffer from these problems because entity positions are not predetermined. Currently, the only predetermined information includes parameters such as formation location, orientation, angle to cover, and spacing between entities. When the security formation behavior is activated during the scenario, it analyzes the current personnel strength of the unit or units tasked with creating security formation. Only after this step does the behavior assign locations to individual entities. This dynamic behavior approach is what enables *masterSecurityFormation* to create security formations which accomplish the mission, regardless of whether or not a unit has incurred casualties.

6.5.4 Areas to Utilize Caution

One important purpose of dynamic behaviors it to decrease scenario creation time through the use of reusable and scalable behaviors. The dynamic R400 scenario provided an excellent example of how composable behaviors can significantly reduce scenario development time, and can make entity behaviors more representative of security formations conducted by Marines on R400. This was possible because considerable effort was devoted to the creation of composable behaviors. However, the creation of this scenario also brought to light several ways that the affordance and HTN processes can be improved. As new dynamic

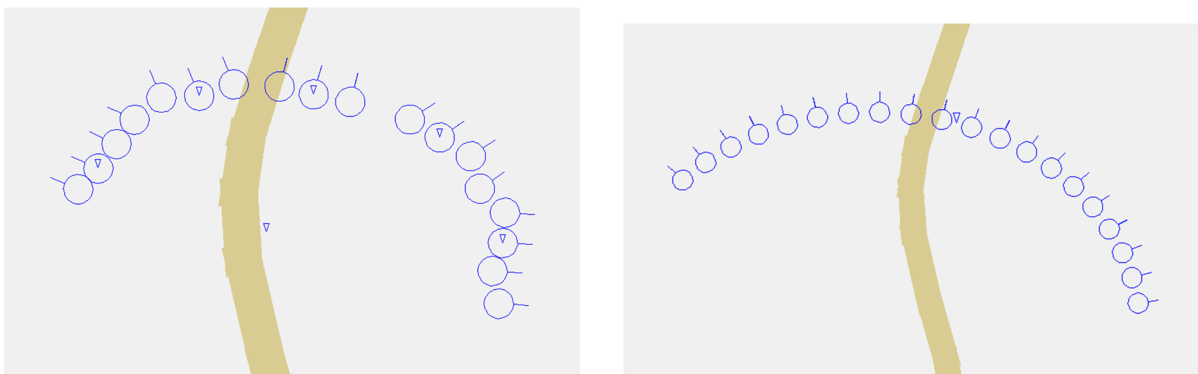


Figure 6.12: 180 Security Formation (No Casualties) - Scripted (Left) and Dynamic (Right)

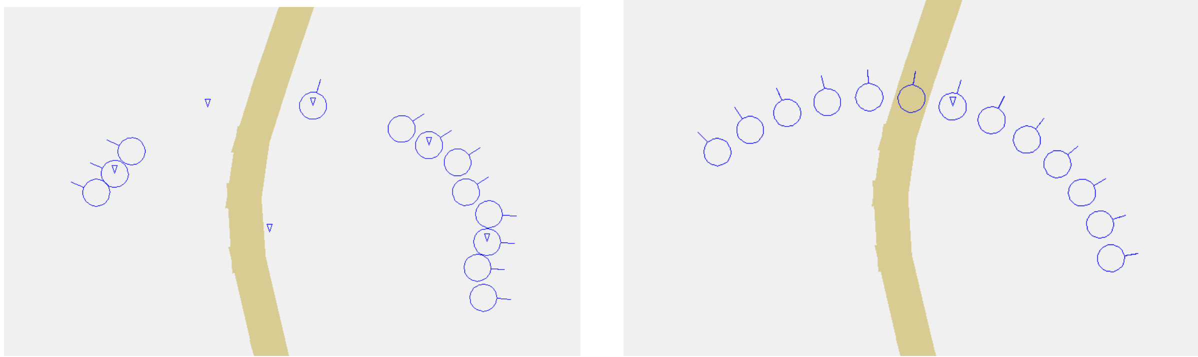


Figure 6.13: 180 Security Formation (With Casualties) - Scripted (Left) and Dynamic (Right)

behaviors are created, simulation developers should be aware of several lessons learned during the creation of the dynamic R400 scenario.

One of the keys to HTN reusability is to ensure that HTN functionality is not dependent upon specially created scripted behaviors. An example of such a problem was found when the *SquadMove* HTN was utilized in the dynamic R400 scenario. This HTN initially failed because it was dependent upon a non-standard COMBATXXI unit formation. The problem was corrected by importing the formation into the R400 scenario; however, this is an issue that should be avoided when creating composable behaviors. It is recommended that such behaviors utilize only standard COMBATXXI capabilities, or that unique HTN behavior is encapsulated within the HTN itself.

Creating sub-behaviors which are modular in nature is an important step toward creating composable behaviors. HTNs are extremely powerful tools, however a single large HTN cannot be expected to achieve representative behavior in all conceivable situations. A better approach to achieving reusability is to create a behavior that is constructed of numerous smaller HTNs. If these smaller HTNs are created in a modular fashion, then scenario developers can easily replace or modify them so that the overall behavior remains representative of actual military operations. An example of this action approach can be seen through the *basicIndividualMove* HTN that is a part of the security formation behavior. This behavior moves simulation entities in a very simple manner that is not representative of many military situations. Ground forces conducting a stealthy patrol will move differently than forces conducting an assault, or forces moving through a minefield. To account for this

aspect, *basicIndividualMove* was created separately from the rest of the security behavior. Should a scenario developer desire additional realism in entity movement, this HTN can be replaced with one or more HTNs that achieve the desired goals.

One of the major benefits of the security formation behavior is its ability to function with units that have incurred casualties. Because future HTNs must be able to do the same thing, it is important to understand some of the challenges that were overcome to achieve this capability. When the security formation behavior was first tested in a scenario where casualties occurred, it failed in cases where the casualties included unit leaders. The reason for this was that COMBATXXI often did not designate a new unit leader, and that many of the security behavior HTNs would only function if they were assigned to a unit leader. This is the first situation in which simulation developers must exercise caution. The next situation in which the security formation behavior initially failed was when a command level unit was completely destroyed. The security formation behavior was designed to replicate the orders process of actual military units in which a command unit receives orders from its higher headquarters, processes those orders, and then passes modified orders down to its subordinate forces. In real life, if that command unit were destroyed, then subordinate unit members reconstitute the command unit. This action does not happen automatically in COMBATXXI, and is the reason why the security formation behavior initially failed. When devising future HTNs, simulation developers must create behaviors that are robust enough to handle such limitations.

6.5.5 Range 400 Conclusion

The previously mentioned issues are ones that developers should be aware of when constructing new scenarios. Truly composable behaviors are rarely easy to develop; however, the advantages they provide are tremendous. Within the dynamic R400 scenario, a greater amount of time was required to properly remove older scripted behaviors, than was required to apply the newer dynamic behaviors. Should this scenario continue to be developed, the reusable and scalable aspects of these behaviors will drastically ease the creation process of future R400 iterations, and add additional realism to the tactical actions being conducted.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 7:
Recommendations and Conclusions

Dr. Andrew Ilachinksi's book *Artificial War* details his work at the Center for Naval Analysis (CNA) when MCCDC Commanding General, Lieutenant General Van Riper opened the "Offices of New Sciences to delve into the possibility of employing complexity theory in support of the command's mission" [1, p. viii]. This work produced two simulations, Irreducible Semi-Autonomous Adaptive Combat (ISAAC) and Enhanced ISAAC Neural Simulation Tool (EINSTein), which ushered in a new era of simulation technology. For the first time, simulations offered the potential for analysts to move beyond the typical Cold War era question of, "which computer-generated army won," and focus on the more insightful question of, "why did one of the computer-generated armies win?"

According to Ilachinski, models usually fell into one of two categories, those "highly realistic models that [provided] little insight into basic processes... [and those] ultra-minimalist models that [stripped] away all but the simplest dynamical variables and [left] out the most interesting real behavior" [1, p. 14]. The work accomplished by Ilachinski produced simulations that found a balance between these two extremes. One of the reasons ISAAC and EINSTein represented revolutionary systems was that they provided the opportunity to model detailed system behaviors while retaining the ability to analyze overall system processes.

Within the United States Army and Marine Corps, the COMBATXXI simulation has the potential to become the present day embodiment of the ideas produced by Ilachinski. It is a system which models the most detailed elements of combat, including weapons system performance, environmental effects, and human behavior. However, the dynamic behavior potential of this system has only seen minimal use. One aspect of military operations that is missing from most combat simulations, including COMBATXXI, is an effort to model dynamic tactical behaviors.

In relation to many simulations, COMBATXXI provides a detailed representation of weapons, unit organizations, and basic tactics. Unfortunately, many realistic human be-

haviors and military tactics, techniques, and procedures (TTPs) must either be scripted by scenario designers, or modeled implicitly. In older Cold War era simulations, this methodology provided suitable results to the questions being asked. As technology has advanced and simulations have gained the ability to represent more detailed actions, a need emerged to correctly represent the behaviors of smaller units. An additional factor which impacts these considerations is the changing role of the individual in modern warfare. Due to the evolution of technology and tactics, individual soldiers and Marines currently have a greater impact on warfare than at any previous time. If a high-resolution simulation such as COMBATXXI is expected to provide valuable insights into modern warfare, each of the above factors must be considered. These issues guided the research conducted in support of this thesis.

7.1 Recommended Improvements to Thesis Products

The security formation behavior and the affordance mechanism produced in support of this thesis aided the COMBATXXI scenario construction process in the following ways:

Representation & Efficiency: enabled scenario entities to display increasingly realistic tactical actions, and lowered the time required for scenario developers to create these tactical actions

Composability: demonstrated how several modular HTNs could work together to produce a single doctrinally correct tactical behavior

Reusability & Scalability: created a composable behavior with high template based reuse properities that could function well in different scenarios or with different sized units

Parameters: identified parameters necessary for the effective utilization of procedural affordances and dynamic security formation behaviors

Creation: identified future military behaviors to model dynamically; identified doctrinal military sources that should be used to guide the development of these behaviors

The products produced within this thesis were intended to ease the introduction of dynamic behaviors into COMBATXXI.[6] If this goal is to become a reality, the following improvements to the thesis products are recommended.

[6] Additional information concerning thesis products can be obtained from the NPS Modeling, Virtual Environments, and Simulation (MOVES) Institute at the following address: *simulation@nps.edu*.

7.1.1 Security Formation Areas for Improvement

The security formation behavior offers scenario developers many capabilities, however, it needs further improvement in order to address some of the tactical deficiencies cited in Chapter 5. Figure 7.1 illustrates additional capabilities that would increase the security formation behavior's utility. The first major area for improvement involves placing entities in portions of the security formation that best suit their capabilities. In real life this process begins when military commanders analyze subordinate unit capabilities, enemy threats, and terrain characteristics.

Within COMBATXXI, analyzing subordinate unit capabilities should be conducted in conjunction with *masterSecurityFormation* and *unitSecurityFormation*. These HTNs were designed to be integrated with future AI mechanisms capable of analyzing a subordinate unit's capabilities and assigning suitable security formation positions to that subordinate unit. Figure 7.2 provides an example of what a subordinate unit analysis could look like for a Marine Infantry Squad (13 Marines) which has been reinforced by an Assault Team (2 Marines). If such a capability were combined with terrain and enemy threat information, algorithms within *unitSecurityFormation* could be used to place the Assault Team in the security positions most conducive to the employment of the team's anti-armor weapon system. Additional methods of employment for this capability include placing the most effective combat units in more dangerous locations, and placing headquarters units in positions where they can accomplish their command and control responsibilities. It is important to recognize that in these examples, the forces being analyzed and adjusted are units which are composed of multiple entities. Eventually, the positions of individual entities must also be determined. Such a task can be accomplished in conjunction with *simpleUnitSecurityFormation* and should use an algorithm designed to assess individual entity capabilities.

The second major area for improvement involves the refinement of individual security positions. Currently, *masterSecurityFormation* is the HTN responsible for identifying the location of each individual security position. This real life equivalent of this action occurs when a unit leader conducts a reconnaissance of a battle position that his unit will occupy in the near future. During this reconnaissance, the leader designates approximate locations of fighting positions. In real life, initial fighting position locations will often be adjusted in order to maximize a unit's capabilities. Within the current security formation behavior, the

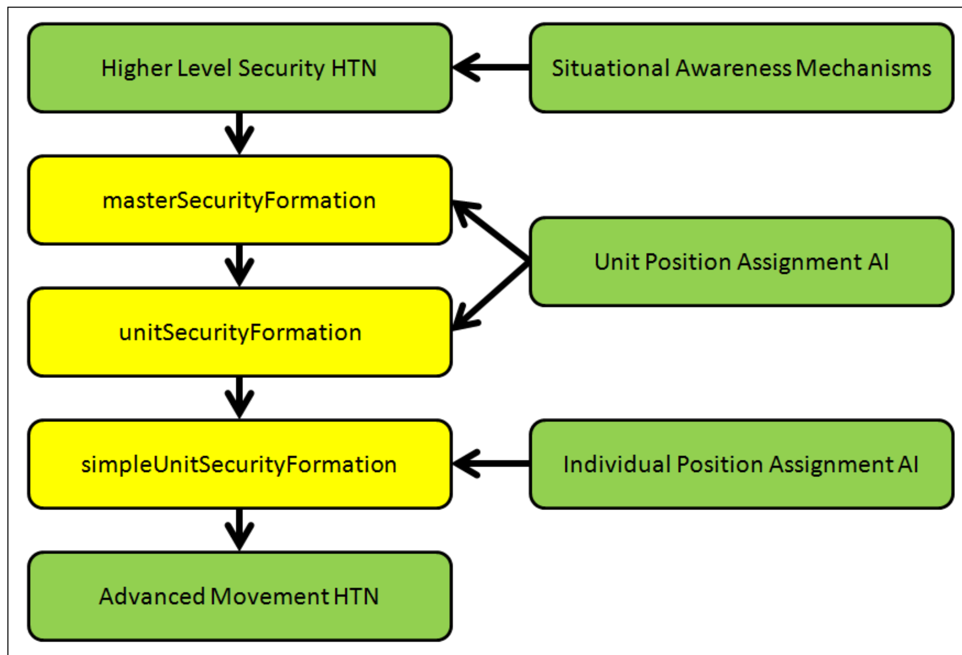


Figure 7.1: Proposed Hierarchy of Security Formation Mechanisms

initial security formation locations are exactly where entities will eventually be located. To improve upon this, functionality must be developed to allow the modification of security positions. Such algorithms would help avoid problems in which entities have their field of view blocked, or situations in which entities fail to take advantage of nearby cover and concealment.

All of these recommendations involve making the action within the security formation behavior more representative of what happens in real life. While accomplishing these recommendations would increase the behavior's utility, the behavior would still be heavily dependent upon formation parameters created by the scenario developer. Examples of these parameters include formation orientation, dispersion among entities, and angle of responsibility. The way to advance the security formation behavior to the next level is to create a higher-level HTN that is able to develop these parameters itself. This HTN would have important ramifications if it was capable of direct reuse, and scenario developers could trust the behavior to create reasonably appropriate security formations. This step would enable an encapsulated security formation behavior to be utilized within other higher level HTNs.

```
infantrySquadAssaultTeam
totalPersonnelInUnit = 15
unitDescriptionOneLevelDown:
[1, 'SIMPLE_UNIT', ['INFANTRY_USMC', 1, 'effectivenessPlaceholder', 'HEADQUARTERS']]
[4, 'SIMPLE_UNIT', ['INFANTRY_USMC', 4, 'effectivenessPlaceholder', 'MAIN_BODY']]
[4, 'SIMPLE_UNIT', ['INFANTRY_USMC', 4, 'effectivenessPlaceholder', 'MAIN_BODY']]
[4, 'SIMPLE_UNIT', ['INFANTRY_USMC', 4, 'effectivenessPlaceholder', 'MAIN_BODY']]
[2, 'SIMPLE_UNIT', ['ANTIARMOR_MEDIUM_USMC', 2, 'effectivenessPlaceholder', 'MAIN_BODY']]
```

Figure 7.2: Analysis of Subunit Capabilities

Such a system has the potential to significantly decrease scenario developer workload. However, producing a behavior that can accomplish these tasks presents many challenges. Friendly unit capabilities, terrain characteristics, and enemy threat are all key pieces of information that a commander considers when deciding how to array his subordinate forces. Each of these areas must be further developed if the behavior is to achieve its full potential. First, the framework for analyzing friendly unit capabilities discussed in this work should be expanded. Second, the COMBATXXI Pathfinder program should be researched to see if it can provide the necessary terrain analysis capabilities. Finally, functionality must be created to assess enemy threats. Unfortunately this is a more complicated topic since enemy threat information can come from many sources, and may or may not be accurate. One way of simplifying this problem is by using informational affordances to convey enemy threat information.

7.1.2 Affordance Areas For Improvement

The current affordance structure provides scenario developers a great deal of capabilities in regard to procedural affordances. However, there exist some areas in which scenario developers must exercise caution, and several ways the affordance functionality should be improved.

- Adding an HTN to an entity's goal stack does not guarantee whether or not an entity will execute a specific action. Execution is ultimately determined by code within the HTN itself. This aspect means that scenario developers must take care to ensure that the wrong entity does not gain access to ONESHOT affordances.
- When entities that are not unit leaders move as part of a formation, they lose their ability to sense affordances. This is because of the way that COMBATXXI executes

formation moves.

- Numerous tools are available for ensuring that the right entities gain access to affordances. These tools could be supplemented by the ability to change an affordance's access roster in the middle of a scenario.
- The ability for entities to create affordances within a scenario could be a powerful tool for producing dynamic behaviors. This improvement would face many of the same challenges involved with changing an affordance's access roster in the middle of a scenario.
- While procedural affordances can be used in a number of ways, limitations will eventually be reached in their utility. The ultimate goal should be the development of informational affordances similar to the Killzone 2 "annotations" that were described in Chapter 3.

7.2 Expanding HTN Usage in COMBATXXI

The products of this thesis shed light on some of the capabilities that dynamic behaviors offer military simulations. Further improvements to these tools can benefit COMBATXXI scenarios. Additional scenario construction tools would also prove beneficial. One of the problems with utilizing Python scripts and HTNs is that they are prone to syntax errors. Improving the graphical user interface (GUI) by adding error checking capabilities would greatly benefit scenario designers. However, the best way to solidify HTNs usage within COMBATXXI is to use these existing tools as inspiration for the creation of additional dynamic behavior mechanisms.

7.2.1 Development of Additional Behaviors

This thesis began with the intent of creating an HTN that was capable of conducting a basic ground attack. Unfortunately, creating a dynamic attack behavior was a more complicated process than originally anticipated. Because of this, the decision was made to focus on producing a composable version of a subordinate behavior that could later be integrated into an attack behavior. The security formation behavior was chosen for this purpose. It accomplished the thesis purpose, which was to demonstrate that doctrinal tactical behaviors can be modeled using HTNs, that HTNs can provide additional realism within the COMBATXXI scenarios, and that HTNs can create reusable and scalable behaviors which

are useful in a variety of situations.

Future work should focus on creating additional behaviors that a military unit exhibits when attacking a defensive enemy position. In general, these actions include those which begin at the unit's assembly area, continue through actions on the objective, and end upon consolidation when the unit has finished preparations for potential enemy counterattacks. Examples of such tactical behaviors include the following:

1. Movement to Contact
2. Support by Fire
3. Fire and Maneuver
4. Fire and Movement
5. Consolidate

Producing dynamic behaviors of these actions would play an important role in establishing a realistic and composable attack behavior. This is a fundamental military action which must be modeled if HTN usage is to be expanded. The creation of an attack behavior would provide many insights into modeling automated military tactics and could ultimately result in the establishment of a behavior library that would revolutionize the manner in which COMBATXXI scenarios are created.

7.2.2 Continued Development of a Dynamic Range 400 Scenario

As additional dynamic behaviors are produced, the R400 scenario should be utilized to demonstrate behavior capabilities. The actual R400 is designed to provide training for all units within USMC infantry battalions. This fact gives a simulated version of R400 an opportunity to showcase a wide range of tactical behaviors. Because a mature R400 evaluation system already exists, this scenario also offers the opportunity to run new behaviors through a structured V&V process. R400 training standards are described in detail by the TTECG produced *R400 Handbook* [47]. Should the R400 scenario continue to be utilized, scenario developers should reference the example scheme of maneuver depicted in Appendix B.

An additional benefit of continuing to use the R400 scenario stems from the fact that this is one of the Marine Corps' premier live fire training facilities. This range is well known to in-

fantry Marines; creating automated behaviors capable of executing tactics representative of those seen during actual R400 runs would garner much attention. Creating higher level behaviors capable of representing the decision making process of unit leaders on R400 would be an even greater accomplishment, and could ultimately take COMBATXXI capabilities to a new level.

7.3 Steps Toward a Mission Planning Capability

From the first days of military boot camps, recruits and officer candidates are taught to perform basic actions as a tactical unit. Over time, military members learn TTPs that dictate what general actions to perform in almost any situation. In order to ensure that the proper actions can be performed in the chaos of combat, these skills are practiced repeatedly until they become second nature. Conducted concurrently with this training are a variety of decision making exercises in which military leaders analyze their current situation and decide upon a course of action for their unit. The unit leader accomplishes this decision making process by selecting a known technique for use, by deciding to modify a previously learned technique, or by devising a completely new course of action. This training exists because the proper execution of tactical behaviors increases a unit's ability to effectively engage enemy forces.

Creating a simulation capable of executing a similar decision making process would represent a revolutionary level of technology. The current limitations of AI make this a long term goal. Making strides toward this goal within the COMBATXXI simulation will require many advances. Capabilities such as the Pathfinder program need to be integrated with dynamic behavior mechanisms, while aspects such as the unit command structure must be updated to better reflect actual military organizations. These are just some of the steps required for the achievement of a mission planning capability; however, the substantial impact such a system could have on the military planning process would make this effort worthwhile. The dynamic behavior techniques established within this thesis are a step in that direction.

APPENDIX A:
Technical Discussion of Affordances

The gaming industry has successfully utilized the affordance concept as a mechanism to control the behavior of NPCs. This appendix describes how an affordance capability was developed for COMBATXXI. From 2007 to 2008, basic affordance components and an initial implementation were developed by Dr. Ron Noel at TRADOC Analysis Center - White Sands Missile Range (TRAC-WSMR) [48]. In this work he demonstrated the feasibility of using affordances in combat models - specifically COMBATXXI. The affordance concept received additional attention during one of Dr. Imre Balogh's classes. The implementation presented in this thesis builds on the concepts initially developed by Dr. Noel, and refined by members of Dr. Balogh's class.

A.1 Functionality

The COMBATXXI affordance system functions in a manner similar to the simple embedded behavior technique described in Chapter 2. At present time, COMBATXXI affordances are similar to those embedded behaviors that provide scripts for an entity to execute. However in this system, HTNs are utilized instead of scripts. Within the scope of this thesis, this technique is referred to as a "procedural affordance." Chapter 2 also describes a technique in which behavioral "hints" can be embedded into terrain; the game Killzone 2 uses a related technique in conjunction with its AI to produce advanced dynamic behaviors. Though not currently possible in COMBATXXI, the ability to develop such capabilities was part of the affordance structure's design. Within the scope of this thesis, this technique is referred to as an "informational affordance." The following files were constructed to give COMBATXXI affordances their current level of functionality:

- affordanceUtilities (Java Library)
- affordanceInitialization (Python Script)
- processAffordances (HTN)

The first file, *affordanceUtilities*, is a library that contains the bulk of the methods related to affordance functionality. Important features include methods for establishing affordance

waypoints, manipulating affordance parameters, and determining if an entity can access an affordance. Because this work focused on procedural affordances, several methods exist to expand HTN functionality. The final purpose of this file is to store affordance related information. When the library is first used, it instantiates several maps that organize information such as affordance parameters, HTN parameters, and affordance-entity access rosters. As of the date of this thesis, this file resides in the COMBATXXI Monterey Extensions package.

The second file, *affordanceInitialization*, is a script that instantiates the affordances for a scenario. The *affordanceInitialization* file should be imported into a scenario's *jump_start* script. This script works closely with the other affordance files, and with SITS. The first step to instantiate an affordance involves using SITS to create waypoints to designate the locations of affordances. Next, the scenario designer will input a list of affordance parameters using the *affordanceUtilities.createNewAffordance()* method. This will result in the creation of a COMBATXXI affordance waypoint. If the affordances are procedural in nature, this is also the time to store HTN parameters. The final function of *affordanceInitialization* is to add the *processAffordances* HTN to the goal stacks of entities that need to access affordances.

The third file, *processAffordances* is an HTN that enables COMBATXXI entities to access and utilize affordances. Figure A.1 depicts the structure of this HTN. So as not to interfere with other HTNs that entities execute, *processAffordances* is not added to the entity's primary goal stack. This HTN has two main parts. The first part is an initialization process that runs once for each entity utilizing the HTN. By accessing information from the affordance-entity access roster in *affordanceUtilities*, this section of code informs entities what affordances they should be aware of as they navigate a COMBATXXI scenario. Once the initialization steps have been accomplished, the HTN "idles" until the entity comes within range of an affordance of interest. When this happens, a replan triggers the second part of the HTN. The purpose of this action is to verify that the entity has access to the affordance. Once access has been granted, *processAffordances* retrieves the HTN that is linked to the affordance, and adds it to the entity's primary goal stack. This last step is simplified because all affordances are currently procedural in nature. As the COMBATXXI affordance capability is refined, additional actions will become possible. For example,

functionality has been built into the process that will allow entities to utilize informational affordances integral to more advanced dynamic behaviors.

A.2 Features

COMBATXXI affordances are most comparable to the gaming industry's legacy technique of defining NPC actions through the use of embedded scripts. However, the current affordance methodology contains features not available in most video games. These features can be generally categorized as mechanisms for allowing an entity to access affordances, and as capabilities created by the combination of affordances and HTNs.

An entity's ability to access affordances is designated within the *affordanceInitialization.createNewAffordance()* method. At this point, access can be authorized for individual entities, all entities within a designated unit, entities of a specified SideType, or all entities within the scenario. The second feature allows scenario developers to designate the range at which an affordance can be detected. This enables entities to access co-located affordances in a desired order. Next, the ability to classify affordances as PERMANENT, TEMPORARY, or ONESHOT further refines the ability to designate when affordances can or cannot be accessed. A PERMANENT affordance can be accessed repeatedly by any number of entities. A TEMPORARY affordance can only be accessed within a specified time window. A ONESHOT affordance can only be accessed a single time. For TEMPORARY and ONESHOT affordances that have exceeded their life expectancy, access is denied to future entities by the automatic adjustment of the affordance's "affordanceValidBoolean." This automatic adjustment can be deliberately executed through the use of the *affordanceUtilities* method that changes affordanceValidBoolean values. This capability enables scenario designers to activate, deactivate, or reactivate affordances at any point of their choosing.

The next set of features was derived from the unanticipated benefit of combining affordances and HTNs. Many of these capabilities were made possible because of the versatile nature of HTNs. The first benefit involves allowing scenario developers to set a time delay until a procedural affordance's HTN is executed. This is accomplished in the *affordanceUtilities.createNewAffordance()* method and allows co-located procedural affordances to be executed in sequence. The second feature of the affordance-HTN combination enables

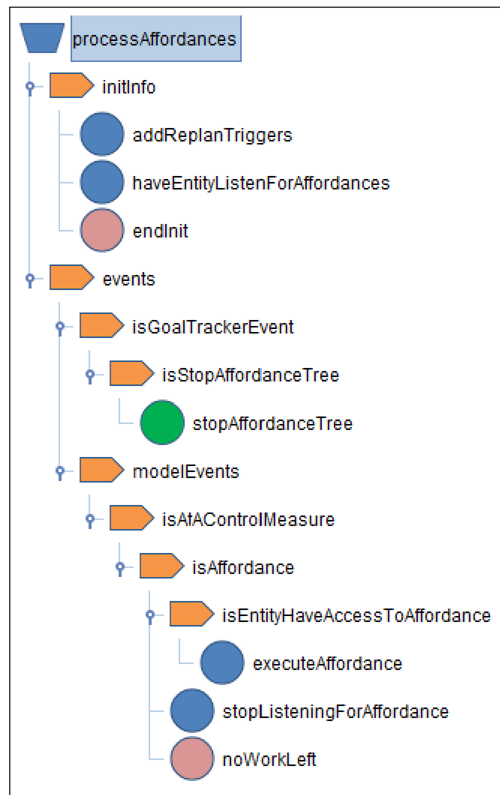


Figure A.1: Depiction of the processAffordances HTN Structure

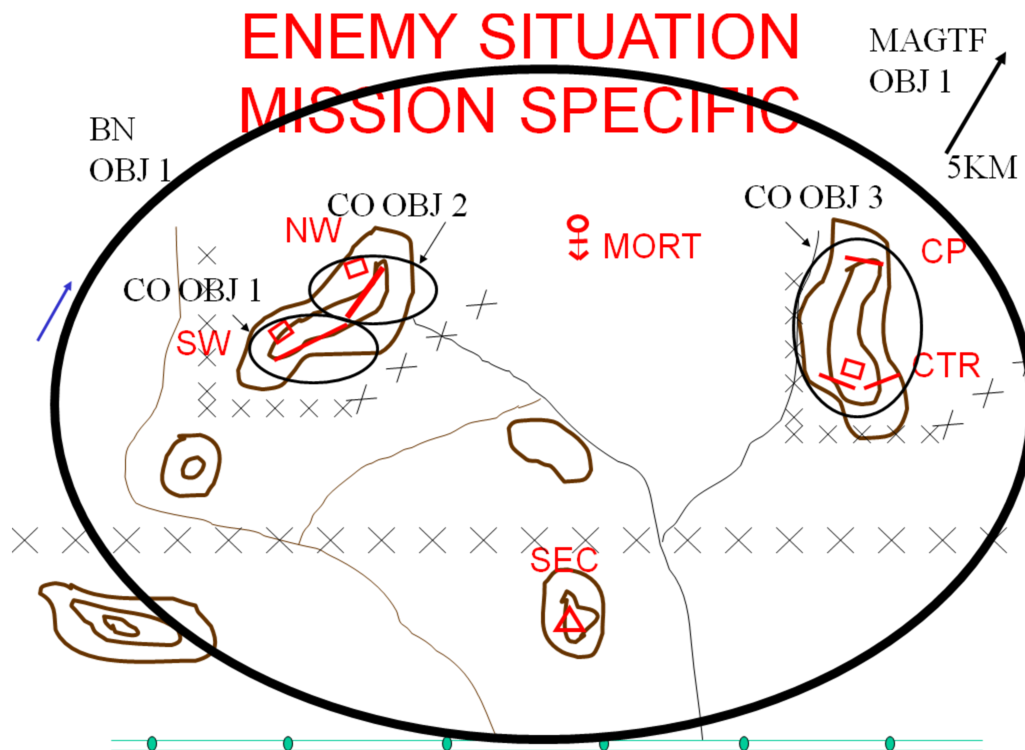
affordances to trigger actions at other places in the scenario. An example of this is the *masterSecurityFormation* HTN described in Chapter 6, in which the HTNs parameters include the unit or units to execute the action. This technique is applicable to military operations because of their event-driven nature. This means that specific movements or actions of one unit can be used to initiate actions of other units. Creating such a behavioral relationship could be facilitated through the use of affordances and HTNs similar in nature to *masterSecurityFormation*. However, care must be used when employing this mechanism to ensure that communication modeling is represented in accordance with the modeling assumptions. The third benefit of procedural affordances concerns the HTN parameters. When procedural affordances are created, HTN parameters are stored within an *affordanceUtilities* HashMap called "htnParameterContainer." This important step enables HTN parameters to be accessed and updated in the middle of a scenario. Uses for this feature with regard to *masterSecurityFormation* include updating the participating units or the geometry of

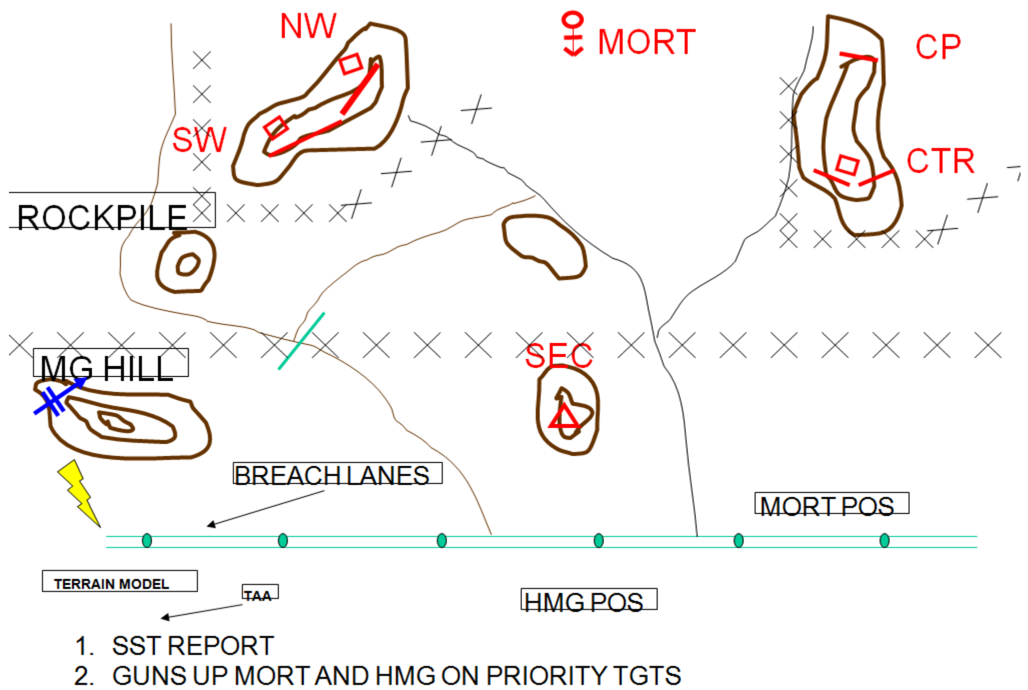
the security formation. The final feature of using the *processAffordances* HTN to process affordances is that this feature can be turned off. In a large scenario, having numerous entities attempting to process affordances can become a computationally expensive process. In order to prevent unnecessary calculations, an entity's *processAffordances* HTN can be brought to a goal state when a scenario developer determines that there is no additional need for that entity to access affordances.

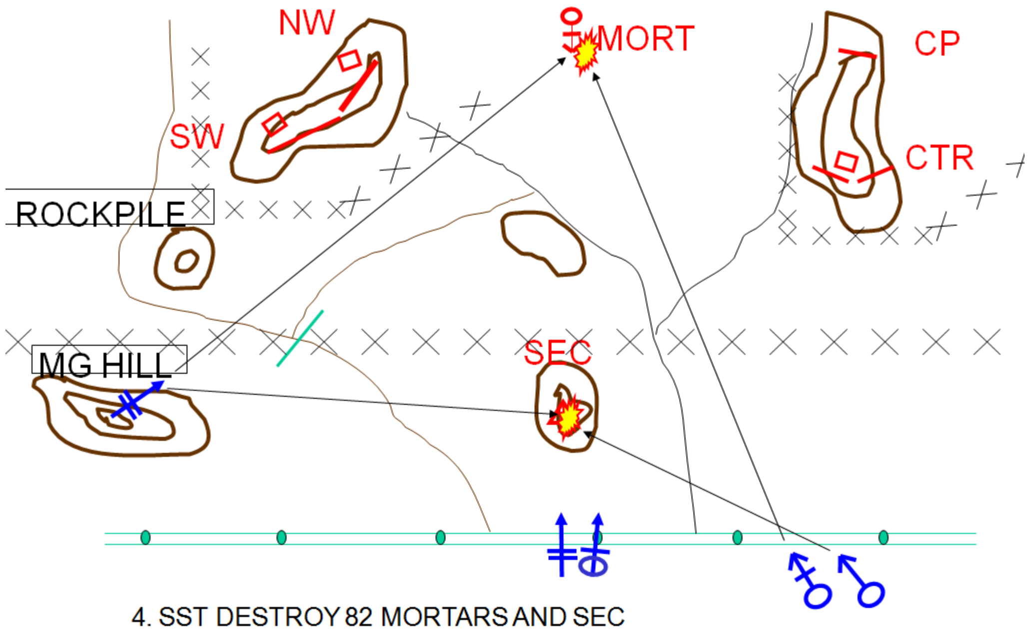
THIS PAGE INTENTIONALLY LEFT BLANK

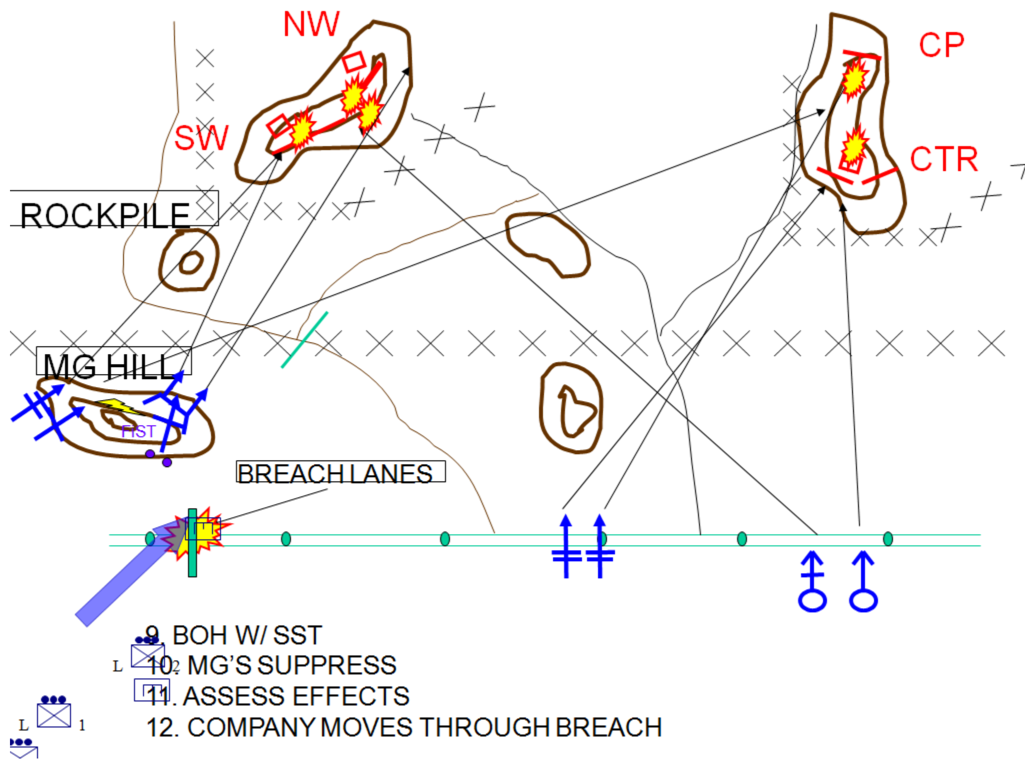
APPENDIX B:
Range 400 Example Attack

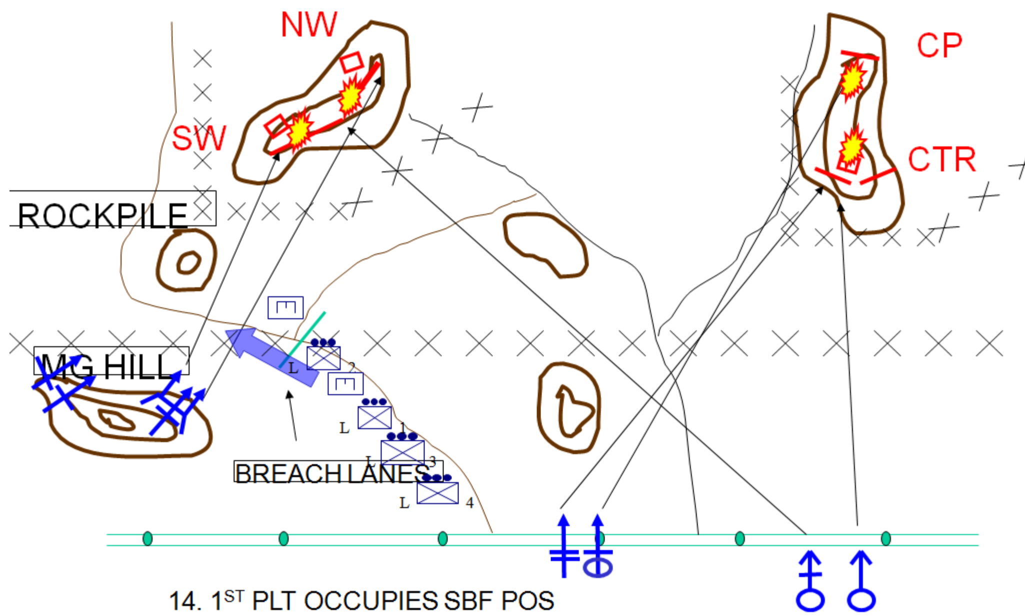
The following presentation illustrates an example of how a reinforced Marine infantry company conducted a R400 evolution at MCAGCC, 29 Palms, CA. If R400 is utilized in future studies, this example will give scenario designers a frame of reference for the overall tactics executed on R400. This example was derived from source [49].

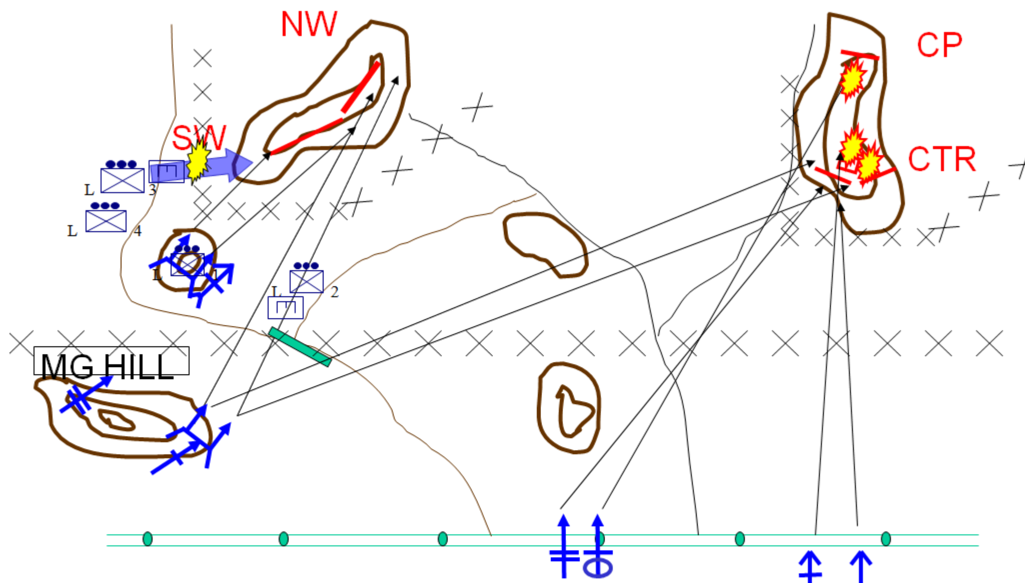



NW
MORT
CP
SW
CTR
ROCKPILE
MG HILL
SEC
BREACH LANES
MORT POS
TERRAIN MODEL
TAA
HMG POS
1. SST REPORT
2. GUNS UP MORT AND HMG ON PRIORITY TGTS

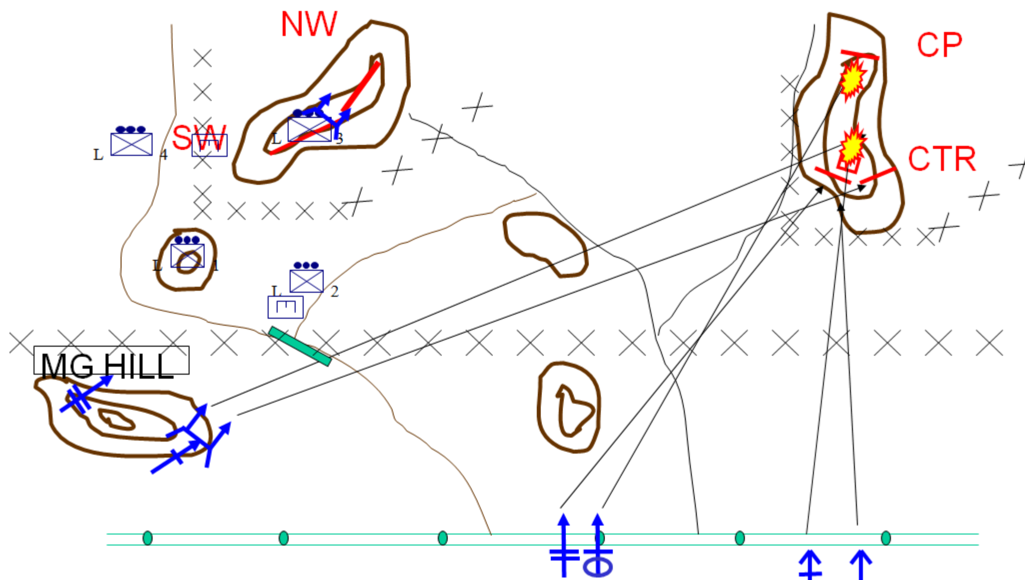

NW
MORT
CP
SW
CTR
ROCKPILE
MG HILL
SEC
4. SST DESTROY 82 MORTARS AND SEC


NW
CP
SW
CTR
ROCKPILE
MG HILL
FIST
BREACH LANES
9. BOH W/ SST
10. MG'S SUPPRESS
11. ASSESS EFFECTS
12. COMPANY MOVES THROUGH BREACH

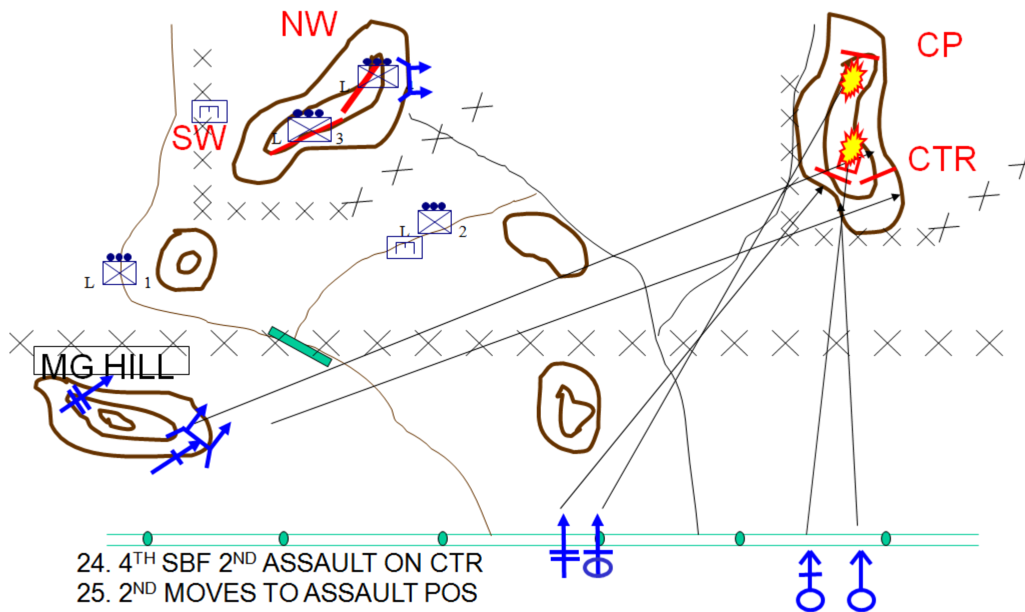

NW
CP
SW
CTR
ROCKPILE
MG HILL
BREACH LANES
14. 1ST PLT OCCUPIES SBF POS

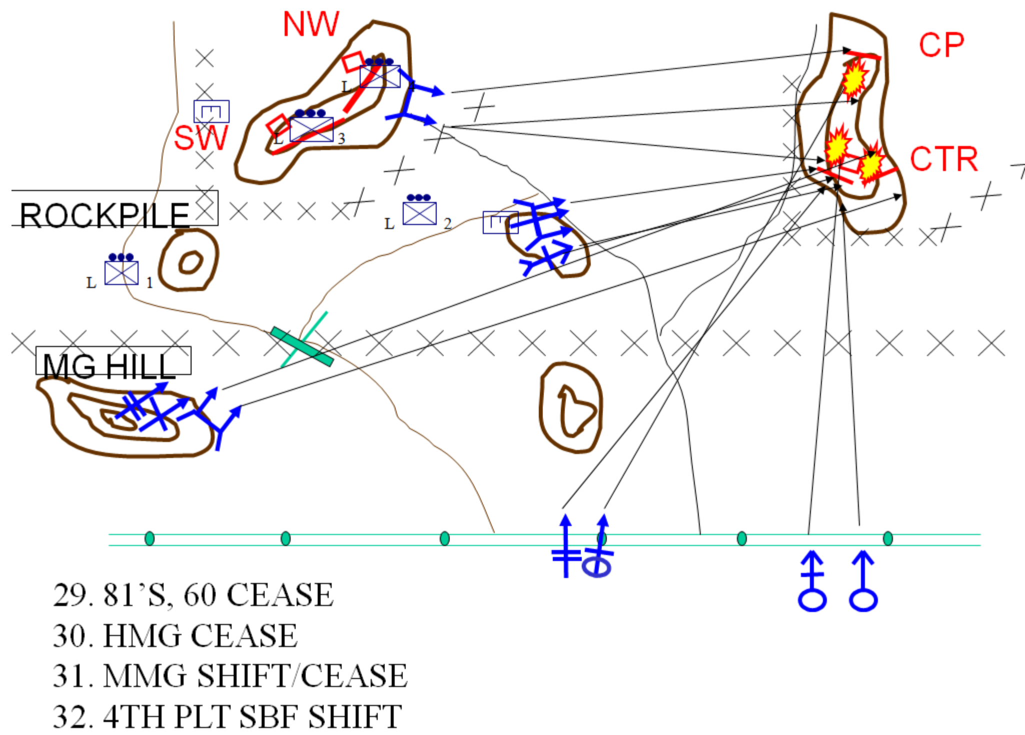


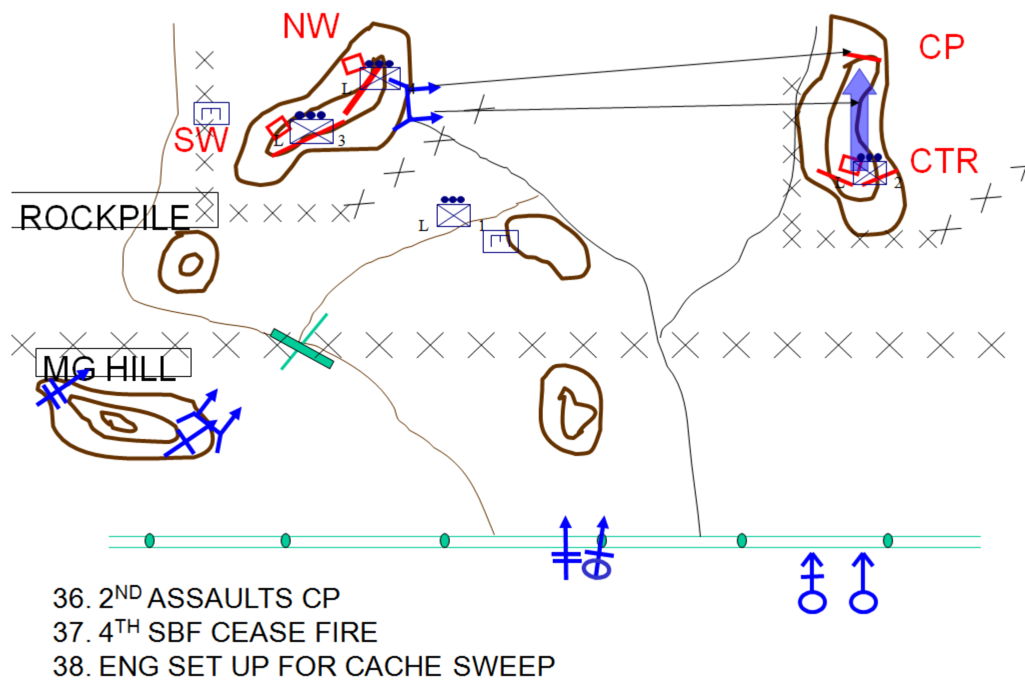
16. ENG BREACH WIRE WITH APOBS
17. 81'S SHIFT FIRE
18. 3RD PLT ASSAULTS SW
19. 1ST PLT CEASES FIRE
20. MMG SHIFT

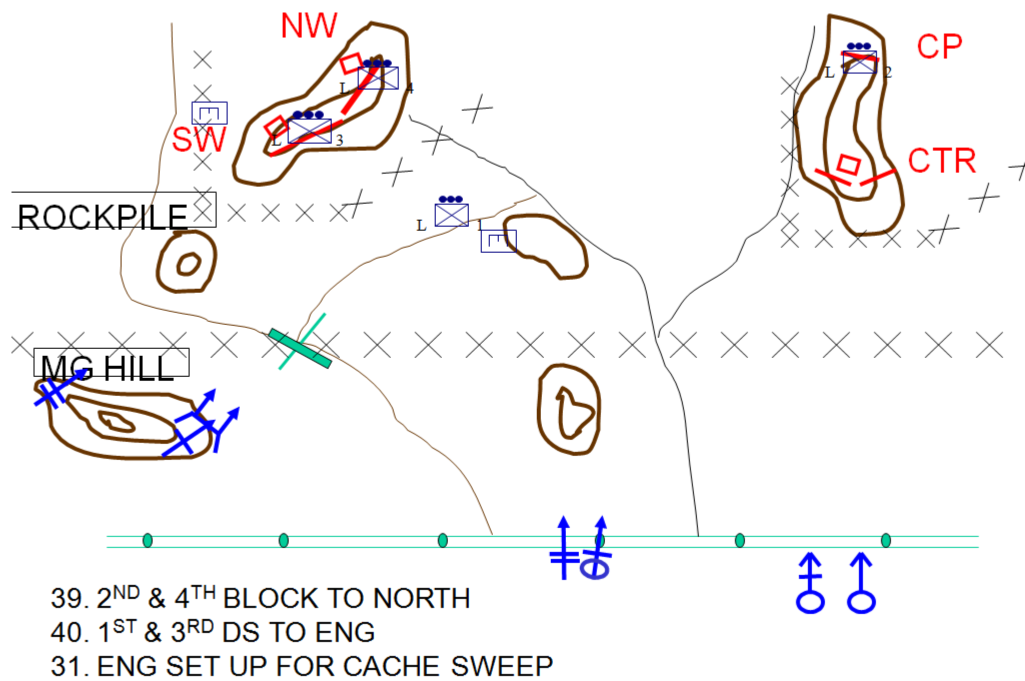


21. 1ST IS IN RESERVE BPT FOLLOW IN SUPPORT OF 2ND, REINFORCE 4TH, PROVIDE SECURITY FOR CASEVAC, SUPPORT ENGINEERS FOR CACHE SWEEP.


NW
SW
CP
CTR
MG HILL
24. 4TH SBF 2ND ASSAULT ON CTR
25. 2ND MOVES TO ASSAULT POS


NW
SW
CP
CTR
ROCKPILE
MG HILL
29. 81'S, 60 CEASE
30. HMG CEASE
31. MMG SHIFT/CEASE
32. 4TH PLT SBF SHIFT


NW
CP
SW
CTR
ROCKPILE
MG HILL
36. 2ND ASSAULTS CP
37. 4TH SBF CEASE FIRE
38. ENG SET UP FOR CACHE SWEEP


NW
CP
SW
CTR
ROCKPILE
MG HILL
39. 2ND & 4TH BLOCK TO NORTH
40. 1ST & 3RD DS TO ENG
31. ENG SET UP FOR CACHE SWEEP

APPENDIX C:
Scenario Initialization Files

The purpose of the scripts in this appendix is to initialize HTNs and affordances within the R400 and urban patrol scenarios.

C.1 Range 400 jump_start.py

This script adds HTNs to the goal stacks of entities in the R400 scenario.

```
1
2  from HTN import UtilityFuncsExp
3  from mtry.cxxi.model.HierarchicalTaskNetwork import GoalContainer
4  from cxxi.model.objects.holders import CMHolder
5  from java.util import ArrayList
6  import math
7
8
9
10 # print a debug message
11 print ""
12 print info.getMyAssignedName(),": JUMP START!"
13 print ""
14
15
16
17 # run the affordanceInitialization script to instantiate all affordances within the scenario
18 initializeAffordancesBoolean = True
19 if initializeAffordancesBoolean:
20   from HTN import r400AffordanceInitialization
21
22
23
24 GoalContainer.initAllEntities()
25
26
27
28 # CONTROL MEASURE NAMES OF INTEREST
29 FUPCenterPosCM = CMHolder.retrieveControlMeasureByName("FUPCenterPos")
30
31
32
33 # PLATOON COMMANDER NAME (DOES NOT ACTIVELY PARTICIPATE IN SCENARIO)
34 # "MarPltCO1"
35
36 # MAIN UNIT NAMES
37 # "Plt1Sqd1"
38 # "Plt1Sqd2"
39 # "MGSqd1"
40
41 # GUARDIAN ANGEL UNIT NAMES
42 # "Plt1Sqd1"
43
44
45 activateHTN = True
46
47 if activateHTN:
48
```

```
49
50
51   arrayListOfUnitNames = ArrayList()
52   arrayListOfUnitNames.add("Plt1Sqd1")
53   arrayListOfUnitNames.add("Plt1Sqd2")
54   arrayListOfUnitNames.add("MGSqd1")
55   UtilityFuncsExp.addGoal(
56     "MarPltCO1",
57     1.0,
58     "HTN/Trees/masterSecurityFormation.xml",
59     ["RECURSIVE", arrayListOfUnitNames, FUPCenterPosCM, 22.0, (0.0)*math.pi, (2.0)*math.pi, "CENTER", "CONVEX", "True"],
60     None)
61
62
63   arrayListOfUnitNames = ArrayList()
64   arrayListOfUnitNames.add("Plt1Sqd1")
65   UtilityFuncsExp.addGoal(
66     "MarPltCO1",
67     2000.0,
68     "HTN/Trees/masterSecurityFormation.xml",
69     ["RECURSIVE", arrayListOfUnitNames, FUPCenterPosCM, 45.0, (1.2)*math.pi, (1.7)*math.pi, "CENTER", "CONVEX", "True"],
70     None)
71
72
73
74 # print a debug message
75 print ""
76 print info.getMyAssignedName(),": JUMP START COMPLETE!"
77 print ""
```

C.2 r400AffordanceInitialization.py

This script initializes affordances within the R400 scenario.

```
1  from mtry.cxxi.model.HierarchicalTaskNetwork import GoalContainer
2  from cxxi.model.objects.holders import CMHolder
3  from java.util import ArrayList
4  from cxxi.model.behavior import OrderUtilities
5  import UtilityFuncs
6  from HTN import UtilityFuncsExp
7  from java.util import Set
8  from mtry.cxxi.model.affordances import affordanceUtilities
9  import math
10
11
12
13 # print a debug message
14 print ""
15 print "start affordanceInitialization"
16 print ""
17
18
19
20 # retrieve control measures from scenario file and assign to local variables
21
22 a1a = CMHolder.retrieveControlMeasureByName("affordance1a")
23 a2a = CMHolder.retrieveControlMeasureByName("affordance2a")
24 a2b = CMHolder.retrieveControlMeasureByName("affordance2b")
25 a2c = CMHolder.retrieveControlMeasureByName("affordance2c")
26 a3a = CMHolder.retrieveControlMeasureByName("affordance3a")
27 a4a = CMHolder.retrieveControlMeasureByName("affordance4a")
28
29 mgHillSecPosCenterWP = CMHolder.retrieveControlMeasureByName("mgHillSecPosCenter")
30
```

```
31
32
33 ################################################################################
34 # 5 STEP PROCESS TO SET UP SCENARIO AFFORDANCES
35 ################################################################################
36
37
38
39 # STEP 1 - SITS
40 ################################################################################
41 # Place (normal) waypoints in SITS scenario file that correspond to desired affordance locations
42 # Give these waypoints a name that is in accordance with your affordance naming convention (ie - affordance1a)
43 # If using HTNs that require locations as parameters, place create these waypoints as well
44 # If desired, the waypoints marking the affordance location can double as an HTN parameter
45
46
47
48 # STEP 2 - WRITE CODE TO INITIALIZE AFFORDANCES
49 ################################################################################
50 #
51 # THREE PART PROCESS
52 # A: create the affordance using affordanceUtilities.createNewAffordance() method
53 # B: create a list of parameters to be used with an HTN (if required)
54 # C: store the list of HTN parameters using affordanceUtilities.updateHtnParameterContainer() method (if required)
55 #
56 # EXPLANATION OF PARAMETERS USED WHEN CREATING A NEW AFFORDANCE
57 # affordanceUtilities.createNewAffordance(  wayPointNameFromSits where affordance is to be located,
58 #                 list of entity names who can access affordance (can be empty)
59 #                 list of unit names whose entities can access affordance (can be empty)
60 #                 affordanceValidBoolean (True or False),
61 #                 affordanceType (PERMANENT, ONESHOT, OR TEMPORARY),
62 #                 quickAccessClassifier (NONE, ALL, BLUE, RED, GREEN, etc...),
63 #                 time in seconds at which a TEMPORARY affordance becomes valid,
64 #                 time in seconds at which a TEMPORARY affordance becomes invalid,
65 #                 delay in seconds until a PROCEDURE affordance adds an HTN to an entity's goal stack
66 #                 rangeInMeters at which an entity can sense that an affordance exists,
67 #                 type of affordance (PROCEDURE or INFORMATION)
68 #                 data (for a PROCEDURE affordance this will usually be the location of an HTN)
69
70
71
72 # affordance1a - uses RECURSIVE version of masterSecurityFormation; all units that are part of "MGSqd1" participate in
      the formation
73
74 affordanceUtilities.createNewAffordance("affordance1a", [], [], "True", "ONESHOT", "BLUE", "0", "0", "60", "5", "
      PROCEDURE", "HTN/Trees/masterSecurityFormation.xml")
75
76 recursionDescriptor = "RECURSIVE"
77 unitNamesArrayList = ArrayList()
78 unitNamesArrayList.add("MGSqd1")
79 formationLocation = mgHillSecPosCenterWP
80 arcBetweenLocations = 10.0
81 formationOrientation = (0.0)*math.pi
82 angleToCover = (2.0)*math.pi
83 formationCenterDescriptor = "CENTER"
84 convexConcaveString = "CONVEX"
85 useIsCommanderBoolean = "False"
86 parameterList = [recursionDescriptor, unitNamesArrayList, formationLocation, arcBetweenLocations, formationOrientation,
      angleToCover, formationCenterDescriptor, convexConcaveString, useIsCommanderBoolean]
87
88 affordanceUtilities.updateHtnParameterContainer("affordance1a", parameterList)
89
90
91
92
93 # affordance2a - traditional use of SquadMove coupled with an affordance
```

```
94
95 affordanceUtilities.createNewAffordance("affordance2a", [], [], "True", "PERMANENT", "BLUE", "0", "0", "60", "5", "
       PROCEDURE", "HTN/Trees/SquadMove.xml")
96
97 parameterList = [a2a, a3a]
98
99 affordanceUtilities.updateHtnParameterContainer("affordance2a", parameterList)
100
101
102
103
104
105 # affordance2b - a modification of SquadMove; this enables one unit to trigger actions by another unit
106
107 affordanceUtilities.createNewAffordance("affordance2b", [], [], "True", "ONESHOT", "BLUE", "0", "0", "60", "5", "
       PROCEDURE", "HTN/Trees/SquadMove_MoveDesignatedUnits.xml")
108
109 unitNameToExecuteList = ArrayList()
110 unitNameToExecuteList.add("Plt1Sqd2")
111 unitNameToExecuteList.add("Plt1Sqd2Tm1")
112 unitNameToExecuteList.add("Plt1Sqd2Tm2")
113 unitNameToExecuteList.add("Plt1Sqd2Tm3")
114 parameterList = [unitNameToExecuteList, a1a, a4a]
115
116 affordanceUtilities.updateHtnParameterContainer("affordance2b", parameterList)
117
118
119
120
121
122 # affordance2c - a modification of SquadMove; this enables one unit to trigger actions by another unit
123
124 affordanceUtilities.createNewAffordance("affordance2c", [], [], "True", "ONESHOT", "BLUE", "0", "0", "300", "5", "
       PROCEDURE", "HTN/Trees/SquadMove_MoveDesignatedUnits.xml")
125
126 unitNameToExecuteList = ArrayList()
127 unitNameToExecuteList.add("MGSqd1")
128 unitNameToExecuteList.add("MGSqd1Tm1")
129 unitNameToExecuteList.add("MGSqd1Tm2")
130 parameterList = [unitNameToExecuteList, mgHillSecPosCenterWP, a4a]
131
132 affordanceUtilities.updateHtnParameterContainer("affordance2c", parameterList)
133
134
135
136
137
138 # affordance3a - uses NONRECURSIVE version of masterSecurityFormation; only designated units (no subordinates)
       participate in formation
139
140 affordanceUtilities.createNewAffordance("affordance3a", [], ["Plt1Sqd1"], "True", "ONESHOT", "NONE", "0", "0", "90", "5"
       , "PROCEDURE", "HTN/Trees/masterSecurityFormation.xml")
141
142 recursionDescriptor = "NONRECURSIVE"
143 unitNamesArrayList = ArrayList()
144 unitNamesArrayList.add("Plt1Sqd1")
145 unitNamesArrayList.add("Plt1Sqd1Tm1")
146 unitNamesArrayList.add("Plt1Sqd1Tm2")
147 unitNamesArrayList.add("Plt1Sqd1Tm3")
148 formationLocation = a3a
149 arcBetweenLocations = 5.0
150 formationOrientation = (0.1)*math.pi
151 angleToCover = (0.6)*math.pi
152 formationCenterDescriptor = "FLOT"
153 convexConcaveString = "CONVEX"
154 useIsCommanderBoolean = "True"
```

```
155 parameterList = [recursionDescriptor, unitNamesArrayList, formationLocation, arcBetweenLocations, formationOrientation,
        angleToCover, formationCenterDescriptor, convexConcaveString, useIsCommanderBoolean]
156
157 affordanceUtilities.updateHtnParameterContainer("affordance3a", parameterList)
158
159
160
161
162
163 # affordance4a - uses a modification of masterSecurityFormation; as units arrive at affordance, they are integrated into
        formation
164
165 affordanceUtilities.createNewAffordance("affordance4a", [], [], "True", "PERMANENT", "BLUE", "0", "0", "60", "5", "
        PROCEDURE", "HTN/Trees/securityFormation_byPhaseAffordanceMod.xml")
166
167 affordanceName = "affordance4a"
168 recursionDescriptor = "NONRECURSIVE"
169 unitNamesArrayList = ArrayList()
170 #unitNamesArrayList.add("someUnitName") # this value is left empty for this HTN
171 formationLocation = a4a
172 arcBetweenLocations = 5.0
173 formationOrientation = (0.1)*math.pi
174 angleToCover = (0.75)*math.pi
175 formationCenterDescriptor = "FLOT"
176 convexConcaveString = "CONVEX"
177 useIsCommanderBoolean = "True"
178 parameterList = [affordanceName, recursionDescriptor, unitNamesArrayList, formationLocation, arcBetweenLocations,
        formationOrientation, angleToCover, formationCenterDescriptor, convexConcaveString, useIsCommanderBoolean]
179
180 affordanceUtilities.updateHtnParameterContainer("affordance4a", parameterList)
181
182
183
184
185
186
187 # STEP 3 - INVERTED GATEKEEPER
188 ##############################################################################
189 # creates an invertedGateKeeper which is used in the HTN to tell entities what affordances to listen for
190 # this method must be run after the affordances have been initialized
191 # this method must be run before processAffordances.xml is added to each entity's goal stack
192 affordanceUtilities.createInvertedGateKeeper()
193
194
195
196
197
198 # STEP 4 - TELL ENTITIES TO LISTEN FOR AFFORDANCES
199 ##############################################################################
200 # enable designated entities to listen for affordances
201 # must be done for all entities that need to access affordances
202 # must be done after the invertedGateKeeper has been created
203 # NOTE: this affordance is added to an entity's "affordanceStack" and not its primary goal stack
204
205
206 listOfEntityNames = [] # left empty for the purpose of this demonstration
207 for entityName in listOfEntityNames:
208   UtilityFuncsExp.addSpecificGoal(
209       entityName,
210       1.0,
211       "HTN/Trees/processAffordances.xml",
212       "affordanceStack",
213       [],
214       None)
215
216
```

```
217
218 listOfUnitNames = ["Plt1Sqd1", "Plt1Sqd1Tm1", "Plt1Sqd1Tm2", "Plt1Sqd1Tm3", "MGSqd1", "MGSqd1Tm1", "MGSqd1Tm2", "
      Plt1Sqd2", "Plt1Sqd2Tm1", "Plt1Sqd2Tm2", "Plt1Sqd2Tm3"]
219 for unitName in listOfUnitNames:
220   UtilityFuncsExp.addSpecificGoalToUnit(
221       unitName,
222       1.0,
223       "HTN/Trees/processAffordances.xml",
224       "affordanceStack",
225       [],
226       None)
227
228
229
230
231
232 # STEP 5 - import this script into the scenario's jump start file
233 ####################################################################################
234
235
236
237
238 # print a debug message
239 print ""
240 print "affordanceInitialization complete"
241 print ""
```

C.3 Urban Patrol jump_start.py

This script adds HTNs to the goal stacks of entities in the urban patrol scenario.

```
1 from mtry.cxxi.model.HierarchicalTaskNetwork import GoalContainer
2 from cxxi.model.objects.holders import CMHolder
3 from java.util import ArrayList
4 from cxxi.model.behavior import OrderUtilities
5 import UtilityFuncs
6 from HTN import UtilityFuncsExp
7 from java.util import Set
8 from mtry.cxxi.model.affordances import affordanceUtilities
9 import math
10
11
12
13 # print a debug message
14 print ""
15 print info.getMyAssignedName(),": JUMP START!"
16 print ""
17
18
19
20 ################################################################################################
21 # CAUTION: if the scenario is changed to maneuver units down streets with numerous buildings
22 # the scenario will likely break.  This is because SquadMove.xml will activate its MoveInBoundingOverwatch.xml
23 # option.  masterSecurityFormation.xml does not currently work with this HTN.  The reason for this
24 # is because of the way that MoveInBoundingOverwatch.xml adjusts a unit's task organization
25 ################################################################################################
26
27
28
29
30 # run the affordanceInitialization script to instantiate all affordances within the scenario
31 initializeAffordancesBoolean = True
32 if initializeAffordancesBoolean:
33   from HTN import urbanPatrolAffordanceInitialization
```

```
34
35
36
37 # assign local variables
38
39 #r1_s = CMHolder.retrieveControlMeasureByName("road1_start")
40 #r1_e = CMHolder.retrieveControlMeasureByName("road1_end")
41 r2_s = CMHolder.retrieveControlMeasureByName("road2_start")
42 r2_e = CMHolder.retrieveControlMeasureByName("road2_end")
43 r3_t = CMHolder.retrieveControlMeasureByName("road3_top")
44 r3_b = CMHolder.retrieveControlMeasureByName("road3_bottom")
45
46 namePlatoonCommander = "DEFAULT_TEAM/B_1MAN_969"
47
48
49
50 # assign HTNs to Platoon Commander in order to start the scenario
51
52 demoPlatoonCommander = True
53
54 goalPath = "HTN/Trees/SquadMove.xml"
55
56
57 if demoPlatoonCommander:
58   UtilityFuncsExp.addGoal(
59       namePlatoonCommander,
60       1.0,
61       goalPath,
62       [r2_s, r2_e],
63       None)
64
65   UtilityFuncsExp.addGoal(
66       namePlatoonCommander,
67       70.0,
68       goalPath,
69       [r2_e, r3_t],
70       None)
71
72   UtilityFuncsExp.addGoal(
73       namePlatoonCommander,
74       125.0,
75       goalPath,
76       [r3_t, r3_b],
77       None)
78
79
80
81 # print a debug message
82 print ""
83 print info.getMyAssignedName(),": JUMP START COMPLETE!"
84 print ""
```

C.4 urbanPatrolAffordanceInitialization.py

This script initializes affordances within the urban patrol scenario.

```
1 from mtry.cxxi.model.HierarchicalTaskNetwork import GoalContainer
2 from cxxi.model.objects.holders import CMHolder
3 from java.util import ArrayList
4 from cxxi.model.behavior import OrderUtilities
5 import UtilityFuncs
6 from HTN import UtilityFuncsExp
7 from java.util import Set
8 from mtry.cxxi.model.affordances import affordanceUtilities
```

```
9  import math
10
11
12
13 # print a debug message
14 print ""
15 print "start affordanceInitialization"
16 print ""
17
18
19
20 # assign local variables
21
22 #r1_s = CMHolder.retrieveControlMeasureByName("road1_start")
23 #r1_e = CMHolder.retrieveControlMeasureByName("road1_end")
24 r2_s = CMHolder.retrieveControlMeasureByName("road2_start")
25 r2_e = CMHolder.retrieveControlMeasureByName("road2_end")
26 r3_t = CMHolder.retrieveControlMeasureByName("road3_top")
27 r3_b = CMHolder.retrieveControlMeasureByName("road3_bottom")
28 d_s = CMHolder.retrieveControlMeasureByName("demo_start")
29
30
31 platoonCommander = "DEFAULT_TEAM/B_1MAN_969"
32 commanderSquad1 = "DEFAULT_TEAM/B_1MAN_948"
33 commanderSquad2 = "DEFAULT_TEAM/B_1MAN_959"
34
35 nameSquad1 = "INF_PLATOON_1/INF_SQUAD_1"
36 nameSquad2 = "INF_PLATOON_1/INF_SQUAD_2"
37
38
39
40 ########################################################################################
41 # if desired, the lead squad can be changed to nameSquad2
42 # the scenario was designed so that it would work regardless of which squad moved first
43 leadSquadForDemonstration = nameSquad1
44 ########################################################################################
45
46
47
48
49 ####################################################################################
50 # 5 STEP PROCESS TO SET UP SCENARIO AFFORDANCES
51 ####################################################################################
52
53
54
55
56 # STEP 1 - SITS
57 ####################################################################################
58 # Place (normal) waypoints in SITS scenario file that correspond to desired affordance locations
59 # Give these waypoints a name that is in accordance with your affordance naming convention (ie - affordance1a)
60 # If using HTNs that require locations as parameters, place create these waypoints as well
61 # If desired, the waypoints marking the affordance location can double as an HTN parameter
62
63
64
65
66 # STEP 2 - WRITE CODE TO INITIALIZE AFFORDANCES
67 ####################################################################################
68 #
69 # THREE PART PROCESS
70 # A: create the affordance using affordanceUtilities.createNewAffordance() method
71 # B: create a list of parameters to be used with an HTN (if required)
72 # C: store the list of HTN parameters using affordanceUtilities.updateHtnParameterContainer() method (if required)
73 #
74 # EXPLANATION OF PARAMETERS USED WHEN CREATING A NEW AFFORDANCE
```

```
75 # affordanceUtilities.createNewAffordance(  wayPointNameFromSits where affordance is to be located,
76 #                 list of entity names who can access affordance (can be empty)
77 #                 list of unit names whose entities can access affordance (can be empty)
78 #                 affordanceValidBoolean (True or False),
79 #                 affordanceType (PERMANENT, ONESHOT, OR TEMPORARY),
80 #                 quickAccessClassifier (NONE, ALL, BLUE, RED, GREEN, etc...),
81 #                 time in seconds at which a TEMPORARY affordance becomes valid,
82 #                 time in seconds at which a TEMPORARY affordance becomes invalid,
83 #                 delay in seconds until a PROCEDURE affordance adds an HTN to an entity's goal stack
84 #                 rangeInMeters at which an entity can sense that an affordance exists,
85 #                 type of affordance (PROCEDURE or INFORMATION)
86 #                 data (for a PROCEDURE affordance this will usually be the location of an HTN)
87
88
89
90
91 # affordance0a
92
93 affordanceUtilities.createNewAffordance("affordance0a", [commanderSquad1, commanderSquad2], [], "True", "ONESHOT", "NONE
       ", "0", "0", "0.1", "10", "PROCEDURE", "HTN/Trees/changeAffordanceValidBoolean.xml")
94
95 affordancesToChangeList = ArrayList()
96 affordancesToChangeList.add("affordance1a")
97 affordancesToChangeList.add("affordance1b")
98 affordancesToChangeList.add("affordance2b")
99 affordancesToChangeList.add("affordance2c")
100 affordancesToChangeList.add("affordance2d")
101 affordancesToChangeList.add("affordance3a")
102 affordancesToChangeList.add("affordance3b")
103 affordancesToChangeList.add("affordance3c")
104 affordancesToChangeList.add("affordance4a")
105 affordancesToChangeList.add("affordance4b")
106 parameterList = ["SWITCH", affordancesToChangeList]
107
108 affordanceUtilities.updateHtnParameterContainer("affordance0a", parameterList)
109
110
111
112
113 # affordance1a
114
115 affordanceUtilities.createNewAffordance("affordance1a", [commanderSquad1, commanderSquad2], [], "False", "TEMPORARY", "
       NONE", "0", "350", "50", "30", "PROCEDURE", "HTN/Trees/SquadMove_OtherUnitsInArrayList.xml")
116
117 unitNameToExecuteList = ArrayList()
118 unitNameToExecuteList.add(nameSquad1)
119 unitNameToExecuteList.add(nameSquad2)
120 parameterList = [unitNameToExecuteList, r2_s, r2_e]
121
122 affordanceUtilities.updateHtnParameterContainer("affordance1a", parameterList)
123
124
125
126
127 # affordance1b
128
129 affordanceUtilities.createNewAffordance("affordance1b", [], [nameSquad1, nameSquad2], "False", "ONESHOT", "NONE", "0", "
       0", "1", "10", "PROCEDURE", "HTN/Trees/securityFormation_byProcessingUnit.xml")
130
131 recursionDescriptor = "NONRECURSIVE"
132 unitNamesArrayList = ArrayList()
133 #unitNamesArrayList.add(someUnitName) # this field left empty for this HTN
134 formationLocation = r2_s
135 arcBetweenLocations = 5.0
136 formationOrientation = (1.05)*math.pi
137 angleToCover = (0.55)*math.pi
```

```
138 formationCenterDescriptor = "FLOT"
139 convexConcaveString = "CONCAVE"
140 useIsCommanderBoolean = "True"
141 parameterList = [recursionDescriptor, unitNamesArrayList, formationLocation, arcBetweenLocations, formationOrientation,
        angleToCover, formationCenterDescriptor, convexConcaveString, useIsCommanderBoolean]
142
143 affordanceUtilities.updateHtnParameterContainer("affordance1b", parameterList)
144
145
146
147
148 # affordance2a
149
150 affordanceUtilities.createNewAffordance("affordance2a", [commanderSquad1, commanderSquad2], [], "False", "ONESHOT", "
        NONE", "0", "0", "1", "15", "PROCEDURE", "HTN/Trees/SquadMove.xml")
151
152 parameterList = [r2_e, r3_t]
153
154 affordanceUtilities.updateHtnParameterContainer("affordance2a", parameterList)
155
156
157
158
159 # affordance2b
160
161 affordanceUtilities.createNewAffordance("affordance2b", [], [], "False", "ONESHOT", "BLUE", "0", "0", "1", "50", "
        PROCEDURE", "HTN/Trees/securityFormation_byProcessingUnit.xml")
162
163 recursionDescriptor = "NONRECURSIVE"
164 unitNamesArrayList = ArrayList()
165 formationLocation = r2_e
166 arcBetweenLocations = 5.0
167 formationOrientation = (1.9)*math.pi
168 angleToCover = (0.05)*math.pi
169 formationCenterDescriptor = "FLOT"
170 convexConcaveString = "CONVEX"
171 useIsCommanderBoolean = "True"
172 parameterList = [recursionDescriptor, unitNamesArrayList, formationLocation, arcBetweenLocations, formationOrientation,
        angleToCover, formationCenterDescriptor, convexConcaveString, useIsCommanderBoolean]
173
174 affordanceUtilities.updateHtnParameterContainer("affordance2b", parameterList)
175
176
177
178
179
180 # affordance2c
181
182 affordanceUtilities.createNewAffordance("affordance2c", [], [], "False", "ONESHOT", "BLUE", "0", "0", "50", "15", "
        PROCEDURE", "HTN/Trees/SquadMove_OtherUnitsInArrayList.xml")
183
184 unitNameToExecuteList = ArrayList()
185 unitNameToExecuteList.add(nameSquad1)
186 unitNameToExecuteList.add(nameSquad2)
187 parameterList = [unitNameToExecuteList, r2_s, r2_e]
188
189 affordanceUtilities.updateHtnParameterContainer("affordance2c", parameterList)
190
191
192
193
194 # affordance2d
195
196 affordanceUtilities.createNewAffordance("affordance2d", [], [], "False", "ONESHOT", "ALL", "0", "0", "25", "15", "
        PROCEDURE", "HTN/Trees/changeAffordanceValidBoolean.xml")
197
```

```
198 affordancesToChangeList = ArrayList()
199 affordancesToChangeList.add("affordance2a")
200 parameterList = ["SWITCH", affordancesToChangeList]
201
202 affordanceUtilities.updateHtnParameterContainer("affordance2d", parameterList)
203
204
205
206
207
208 # affordance3a
209
210 affordanceUtilities.createNewAffordance("affordance3a", [commanderSquad1, commanderSquad2], [], "False", "TEMPORARY", "
        NONE", "560", "99999", "1", "25", "PROCEDURE", "HTN/Trees/SquadMove.xml")
211
212 parameterList = [r3_t, r3_b]
213
214 affordanceUtilities.updateHtnParameterContainer("affordance3a", parameterList)
215
216
217
218
219
220 # affordance3b
221
222 affordanceUtilities.createNewAffordance("affordance3b", [], [], "False", "ONESHOT", "BLUE", "0", "0", "1", "5", "
        PROCEDURE", "HTN/Trees/securityFormation_byProcessingUnit.xml")
223
224 recursionDescriptor = "NONRECURSIVE"
225 unitNamesArrayList = ArrayList()
226 formationLocation = r3_t
227 arcBetweenLocations = 5.0
228 formationOrientation = (1.6)*math.pi
229 angleToCover = (1.35)*math.pi
230 formationCenterDescriptor = "CENTER"
231 convexConcaveString = "CONVEX"
232 useIsCommanderBoolean = "True"
233 parameterList = [recursionDescriptor, unitNamesArrayList, formationLocation, arcBetweenLocations, formationOrientation,
        angleToCover, formationCenterDescriptor, convexConcaveString, useIsCommanderBoolean]
234
235 affordanceUtilities.updateHtnParameterContainer("affordance3b", parameterList)
236
237
238
239
240 # affordance3c
241
242 affordanceUtilities.createNewAffordance("affordance3c", [], [], "False", "ONESHOT", "BLUE", "0", "0", "50", "15", "
        PROCEDURE", "HTN/Trees/SquadMove_OtherUnitsInArrayList.xml")
243
244 unitNameToExecuteList = ArrayList()
245 unitNameToExecuteList.add(nameSquad1)
246 unitNameToExecuteList.add(nameSquad2)
247 parameterList = [unitNameToExecuteList, r2_e, r3_t]
248
249 affordanceUtilities.updateHtnParameterContainer("affordance3c", parameterList)
250
251
252
253
254 # affordance4a
255
256 affordanceUtilities.createNewAffordance("affordance4a", [], [], "False", "PERMANENT", "BLUE", "0", "0", "30", "5", "
        PROCEDURE", "HTN/Trees/securityFormation_byPhaseAffordanceMod.xml")
257
258 affordanceName = "affordance4a"
```

```
259 recursionDescriptor = "NONRECURSIVE"
260 unitNamesArrayList = ArrayList()
261 formationLocation = r3_b
262 arcBetweenLocations = 5.0
263 formationOrientation = (1.0)*math.pi
264 angleToCover = (2.0)*math.pi
265 formationCenterDescriptor = "CENTER"
266 convexConcaveString = "CONVEX"
267 useIsCommanderBoolean = "True"
268 parameterList = [affordanceName, recursionDescriptor, unitNamesArrayList, formationLocation, arcBetweenLocations,
        formationOrientation, angleToCover, formationCenterDescriptor, convexConcaveString, useIsCommanderBoolean]
269
270 affordanceUtilities.updateHtnParameterContainer("affordance4a", parameterList)
271
272
273
274
275
276 # affordance4b
277
278 affordanceUtilities.createNewAffordance("affordance4b", [], [], "False", "ONESHOT", "BLUE", "0", "0", "50", "10", "
        PROCEDURE", "HTN/Trees/SquadMove_OtherUnitsInArrayList.xml")
279
280 unitNameToExecuteList = ArrayList()
281 unitNameToExecuteList.add(nameSquad1)
282 unitNameToExecuteList.add(nameSquad2)
283 parameterList = [unitNameToExecuteList, r3_t, r3_b]
284
285 affordanceUtilities.updateHtnParameterContainer("affordance4b", parameterList)
286
287
288
289
290
291 # affordance4c
292
293 affordanceUtilities.createNewAffordance("affordance4c", [platoonCommander], [], "True", "ONESHOT", "NONE", "0", "0", "
        120", "10", "PROCEDURE", "HTN/Trees/SquadMove_MoveDesignatedUnits.xml")
294
295 unitNameToExecuteList = ArrayList()
296 unitNameToExecuteList.add(leadSquadForDemonstration)
297 parameterList = [unitNameToExecuteList, d_s, r2_s]
298
299 affordanceUtilities.updateHtnParameterContainer("affordance4c", parameterList)
300
301
302
303
304
305 # affordance4d
306
307 affordanceUtilities.createNewAffordance("affordance4d", [platoonCommander], [], "True", "PERMANENT", "NONE", "0", "0", "
        90", "10", "PROCEDURE", "HTN/Trees/stopProcessAffordancesHTN.xml")
308
309 parameterList = [] # no parameters needed for this HTN
310
311 affordanceUtilities.updateHtnParameterContainer("affordance4d", parameterList)
312
313
314
315
316
317 # affordance4e
318
319 affordanceUtilities.createNewAffordance("affordance4e", [platoonCommander], [], "True", "ONESHOT", "NONE", "0", "0", "1"
        , "5", "PROCEDURE", "HTN/Trees/masterSecurityFormation.xml")
```

```
320
321 recursionDescriptor = "NONRECURSIVE"
322 unitNamesArrayList = ArrayList()
323 unitNamesArrayList.add(nameSquad1)
324 unitNamesArrayList.add(nameSquad2)
325 formationLocation = d_s
326 arcBetweenLocations = 4.0
327 formationOrientation = (1.0)*math.pi
328 angleToCover = (2.0)*math.pi
329 formationCenterDescriptor = "CENTER"
330 convexConcaveString = "CONVEX"
331 useIsCommanderBoolean = "True"
332 parameterList = [recursionDescriptor, unitNamesArrayList, formationLocation, arcBetweenLocations, formationOrientation,
        angleToCover, formationCenterDescriptor, convexConcaveString, useIsCommanderBoolean]
333
334 affordanceUtilities.updateHtnParameterContainer("affordance4e", parameterList)
335
336
337
338
339 # affordance4f
340
341 affordanceUtilities.createNewAffordance("affordance4f", [platoonCommander], [], "True", "ONESHOT", "NONE", "0", "0", "60
        ", "5", "PROCEDURE", "HTN/Trees/removeUnitFromAffSecForm.xml")
342
343 wpNameInSits = "affordance4e"
344 listOfUnitNamesToRemove = ArrayList()
345 listOfUnitNamesToRemove.add(leadSquadForDemonstration)
346 controlMeasureToMoveTo = d_s
347 newFormationParameterBoolean = "True"
348 newArcBetweenLocations = 10.0
349 parameterList = [wpNameInSits, listOfUnitNamesToRemove, controlMeasureToMoveTo, newFormationParameterBoolean,
        newArcBetweenLocations]
350
351 affordanceUtilities.updateHtnParameterContainer("affordance4f", parameterList)
352
353
354
355
356
357
358 # STEP 3 - INVERTED GATEKEEPER
359 ################################################################################
360 # creates an invertedGateKeeper which is used in the HTN to tell entities what affordances to listen for
361 # this method must be run after the affordances have been initialized
362 # this method must be run before processAffordances.xml is added to each entity's goal stack
363 affordanceUtilities.createInvertedGateKeeper()
364
365
366
367
368
369
370 # STEP 4 - TELL ENTITIES TO LISTEN FOR AFFORDANCES
371 ################################################################################
372 # enable designated entities to listen for affordances
373 # must be done for all entities that need to access affordances
374 # must be done after the invertedGateKeeper has been created
375 # NOTE: this affordance is added to an entity's "affordanceStack" and not its primary goal stack
376
377
378 listOfEntityNames = [commanderSquad1, commanderSquad2, platoonCommander]
379 for entityName in listOfEntityNames:
380   UtilityFuncsExp.addSpecificGoal(
381       entityName,
382       1.0,
```

```
383         "HTN/Trees/processAffordances.xml",
384         "affordanceStack",
385         [],
386         None)
387
388
389 listOfUnitNames = []
390 for unitName in listOfUnitNames:
391     UtilityFuncsExp.addSpecificGoalToUnit(
392         unitName,
393         1.0,
394         "HTN/Trees/processAffordances.xml",
395         "affordanceStack",
396         [],
397         None)
398
399
400
401
402 # STEP 5 - import this script into the scenario's jump start file
403 ###################################################################################
404
405
406
407
408 # print a debug message
409 print ""
410 print "affordanceInitialization complete"
411 print ""
```

APPENDIX D:
Affordance Files

The following files are responsible for creating the affordance functionality within COMBATXXI.

D.1 affordanceUtilities.java

This script contains functions that are utilized by the other affordance files.

```
1  package mtry.cxxi.model.affordances;
2
3  import cxxi.model.knowledge.group.OperationalGroup;
4  import cxxi.model.knowledge.group.holders.NewUnitHolder;
5  import cxxi.model.objects.entity.Entity;
6  import cxxi.support.UniqueIDFactory;
7  import cxxi.model.objects.features.CMWayPoint;
8  import cxxi.model.objects.features.ControlMeasure;
9  import cxxi.model.objects.holders.CMHolder;
10 import cxxi.model.physics.geometry.Location;
11 import cxxi.support.types.SideType;
12 import cxxi.support.types.WayPointType;
13 import java.util.ArrayList;
14 import java.util.HashMap;
15 import java.util.List;
16 import java.util.Set;
17 import simkit.Schedule;
18
19
20 public class affordanceUtilities {
21
22     // these variables are "instantiated" when the first "affordanceUtilities" method is called by a python script
23     // these maps are used to store information which is necessary to make the affordances work
24     public static HashMap<String, Object> workingTagContainer= new HashMap<>(); // holds the "data" that corresponds to
         an affordance
25     public static HashMap<String, List> workingGateKeeper = new HashMap<>(); // holds a listOfEntitiesWithAccess list
         for each affordance uniqueID
26     public static HashMap<String, HashMap> workingTagAttributeContainer = new HashMap<>(); // holds attributes that
         affect how each affordance functions
27     public static HashMap<String, List> workingInvertedGateKeeper = new HashMap<>(); // holds a list of affordances that
         each entity can access
28     public static HashMap<String, List> htnParameterContainer = new HashMap<>(); // stores HTN parameters
29
30
31
32
33
34     /**
35      * Adds an entry to the gateKeeper, this function is called in the corresponding python library
36      * @param uniqueID is uniqueID of the referenced affordance waypoint
37      * @param listOfEntitiesWithAccess is a list of entity IDs that are allowed to access the tag's data
38      */
39     public static void addGateKeeperEntry(String uniqueID, List listOfEntitiesWithAccess){
40         workingGateKeeper.put(uniqueID, listOfEntitiesWithAccess);
41     }
42
43
44
45
46
```

```
47      /**
48       * Method designed for being called from python to create an affordance control measure
49       * at the specified location
50       *
51       * @param wpNameInSits this string corresponds to the waypoint name in Sits (such as "affordance1a")
52       * @param loc Location location for control measure
53       * @param affordanceValidBoolean "True" or "False" - determines if an affordance can be accessed at all
54       * @param affordanceType ONESHOT, PERMANENT, TEMPORARY
55       * @param quickAccessClassifier NONE, ALL, BLUE, RED, etc...
56       * @param validTime sim time (in seconds) at which a temporary affordance can be accessed
57       * @param cancelTime sim time (in seconds) at which a temporary affordance can no longer be accessed
58       * @param delay delay (in seconds) until a procedural affordance executes an HTN
59       * @param rangeInMeters how close an entity needs to be to try to access an affordance
60       * @param dataType INFORMATION, PROCEDURE
61       * @param data the object that holds the actual information or procedure
62       *
63       * @return affordance waypoint's unique ID
64       */
65      public static String affordanceAddControlMeasure(
66              Location loc,
67              String wpNameInSits,
68              String affordanceValidBoolean,
69              String affordanceType,
70              String quickAccessClassifier,
71              String validTime,
72              String cancelTime,
73              String delay,
74              String rangeInMeters,
75              String dataType,
76              Object data)
77      {
78
79          String dataPlaceHolder = "None";
80          if (dataType.equalsIgnoreCase("Procedure")){
81              dataPlaceHolder = data.toString();
82          }
83
84          // creates a string that corresponds to a waypoint's name
85          String affordanceName = wpNameInSits+":"+affordanceValidBoolean+":"+affordanceType+":"+quickAccessClassifier+":"
        +validTime+":"+cancelTime+":"+delay+":"+rangeInMeters+":"+dataType+":"+dataPlaceHolder;
86
87          CMWayPoint wp = new CMWayPoint(
88                  loc,
89      affordanceName,
90                  //ent.getSide(),
91                  SideType.UNKNOWN, // SideType of UNKNOWN because it doesn't matter
92                  WayPointType.AFFORDANCE,
93                  Integer.parseInt(UniqueIDFactory.getStringID()));
94
95          CMHolder.addControlMeasure(wp);
96
97          // sets radius waypoint at which an entity will sense an affordance exists (meters)
98          wp.setTolerance(Double.valueOf(rangeInMeters));
99
100         // updates the workingTagAttributeContainer and the workingTagContainer
101         String wpID = wp.getID();
102         HashMap mapOfTagAttributes = affordanceUtilities.formatWayPointName(affordanceName);
103         workingTagAttributeContainer.put(wpID, mapOfTagAttributes);
104         workingTagContainer.put(wpID, data);
105
106         return wpID;
107     }
108
109
110
111
```

```
112
113
114
115    /**
116     * This function changes the affordanceValidBoolean to false; this is important to do after
117     * a Temp affordance has "timed out" or after a oneShot affordance has been used
118     * Important to note, this does not change the name that is included in the waypoint, this only
119     * changes the affordanceValidBoolean that is stored in the workingTagAttributeContainer
120
121     * @param uniqueID
122     */
123    public static void changeAffordanceValidBooleanToFalse(String uniqueID){
124        String affordanceName = affordanceUtilities.getFullAffordanceName(uniqueID);
125        HashMap mapOfTagAttributes = affordanceUtilities.formatWayPointName(affordanceName);
126        // this replaces the current "True" value of affordanceValidBoolean to "False"
127        mapOfTagAttributes.put("affordanceValidBoolean", "False");
128        workingTagAttributeContainer.put(uniqueID, mapOfTagAttributes);
129    }
130
131
132
133
134
135
136
137
138    /**
139     * This function changes the affordanceValidBoolean to True.
140     * Important to note, this does not change the name that is included in the waypoint, this only
141     * changes the affordanceValidBoolean that is stored in the workingTagAttributeContainer
142
143     * @param uniqueID
144     */
145    public static void changeAffordanceValidBooleanToTrue(String uniqueID){
146        String affordanceName = affordanceUtilities.getFullAffordanceName(uniqueID);
147        HashMap mapOfTagAttributes = affordanceUtilities.formatWayPointName(affordanceName);
148        // this replaces the current "False" value of affordanceValidBoolean to "True"
149        mapOfTagAttributes.put("affordanceValidBoolean", "True");
150        workingTagAttributeContainer.put(uniqueID, mapOfTagAttributes);
151    }
152
153
154
155
156
157
158
159    /**
160     * this function creates an invertedGateKeeper; this object is used to tell
161     * entities what affordances they should be listening for
162     */
163    public static void createInvertedGateKeeper(){
164        workingInvertedGateKeeper.clear();
165        for (String uniqueID : workingGateKeeper.keySet()){
166            List listOfEntities = workingGateKeeper.get(uniqueID);
167            for (Object testEntity : listOfEntities){
168                String testEntityString = (String)testEntity;
169                if (workingInvertedGateKeeper.containsKey(testEntityString)){
170                    List listToAddTo = workingInvertedGateKeeper.remove(testEntityString);
171                    listToAddTo.add(uniqueID);
172                    workingInvertedGateKeeper.put(testEntityString, listToAddTo);
173                }
174                else if (!workingInvertedGateKeeper.containsKey(testEntityString)){
175                    ArrayList newList = new ArrayList<>();
176                    newList.add(uniqueID);
177                    workingInvertedGateKeeper.put(testEntityString, newList);
```

```
178                 }
179                 else{
180                     System.out.print("AN ERROR OCCURRED WHILE CREATING THE invertedGateKeeper");
181                 }
182             }
183         }
184     }
185
186
187
188
189
190
191
192     /**
193      *
194      * @param wpNameInSits
195      * @param listOfEntitiesWithAccess
196      * @param affordanceValidBoolean
197      * @param affordanceType
198      * @param quickAccessClassifier
199      * @param validTime
200      * @param cancelTime
201      * @param delay
202      * @param rangeInMeters
203      * @param dataType
204      * @param data
205      * @return
206      */
207     public static String createNewAffordance(   String wpNameInSits,
208                                                 List listOfEntitiesWithAccess,
209                                                 List listOfUnitsWithAccess,
210                                                 String affordanceValidBoolean,
211                                                 String affordanceType,
212                                                 String quickAccessClassifier,
213                                                 String validTime,
214                                                 String cancelTime,
215                                                 String delay,
216                                                 String rangeInMeters,
217                                                 String dataType,
218                                                 Object data){
219
220         // creates a new affordance waypoint (this line creates the WP, makes an entry in the tagAttributeContainer,
        makes entry in tagContainer, and returns the unique ID for later use)
221     ControlMeasure currentControlMeasure = CMHolder.retrieveControlMeasureByName(wpNameInSits);
222     String uniqueID = affordanceAddControlMeasure(  currentControlMeasure.getLocation(),
223                                                     wpNameInSits,
224                                                     affordanceValidBoolean,
225                                                     affordanceType,
226                                                     quickAccessClassifier,
227                                                     validTime,
228                                                     cancelTime,
229                                                     delay,
230                                                     rangeInMeters,
231                                                     dataType,
232                                                     data);
233
234         List<String> listOfEntityNames = getArrayListOfEntityNames(listOfUnitsWithAccess);
235
236         for (String entityName : listOfEntityNames){
237             listOfEntitiesWithAccess.add(entityName);
238         }
239
240         // create appropriate entry in gateKeeper
241     addGateKeeperEntry(uniqueID, listOfEntitiesWithAccess);
242
```

```
243         boolean printBasicAffordanceInfoBoolean = true;
244    if (printBasicAffordanceInfoBoolean){
245             System.out.println("affordance uniqueID = "+uniqueID);
246             System.out.println("affordance data = "+data.toString());
247             System.out.println("listOfEntitiesWithAccess = "+listOfEntitiesWithAccess.toString());
248             System.out.println("quickAccessClassifier = "+quickAccessClassifier);
249             System.out.println();
250
251         }
252
253         return uniqueID;
254     }
255
256
257
258
259
260
261
262  /**
263   * determines if a uniqueID corresponds to an Affordance (used to determine if ControlMeasure is an Affordance)
264   * used in processAffordancesTree
265   * @param uniqueID
266   * @return
267   */
268     public static boolean determineIfAffordance(String uniqueID){
269         return workingTagContainer.containsKey(uniqueID);
270     }
271
272
273
274
275
276
277
278
279
280 // BEGIN entityAttemptToAccessAffordance FUNCTION
281 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
282 ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
283
284     /**
285      * this block of code determines whether or not an entity is allowed to access affordance data
286      * @param ent
287      * @param uniqueID
288      * @return affordance data if the entity has access (null if the entity does not have access)
289      */
290     public static Object entityAttemptToAccessAffordance(Entity ent, String uniqueID){
291
292
293         //////////////////////////////////////////////////////////////////////////////////////////
294         // this line checks whether or not an affordance is valid; if not, the method immediately exits
295         // this is mostly applicable to oneShots that have been used or temps that have "timed out"
296         String affordanceValidBoolean = affordanceUtilities.getAffordanceAttribute(uniqueID, "affordanceValidBoolean");
297         if (affordanceValidBoolean.equalsIgnoreCase("False")) {
298             return null;
299         }
300
301
302         //////////////////////////////////////////////////////////////////////////////////////////
303         // set "local" variables that will be used within this function
304         boolean accessGranted = false;
305         boolean isAffordanceTemp = false;
306
307
308         //////////////////////////////////////////////////////////////////////////////////////////
```

```
309         // collects and organizes information about the affordance, and the entity attempting to access the affordance
310
311         // determines information concerning the entity that is trying to access the affordance
312         SideType thisEntitySideType = ent.getSide();
313         String thisEntityName = ent.getAssignedName();
314
315         // this block of code retrieves and formats the tag's name
316         ControlMeasure affordanceAttemptingToAccess = CMHolder.retrieveControlMeasureByID(uniqueID);
317
318         String quickAccessClassifier = affordanceUtilities.getAffordanceAttribute(uniqueID, "quickAccessClassifier");
319         String affordanceType = affordanceUtilities.getAffordanceAttribute(uniqueID, "affordanceType");
320
321         if (affordanceType.equalsIgnoreCase("TEMPORARY")){
322             isAffordanceTemp = true;
323         }
324
325
326         ///////////////////////////////////////////////////////////////////////////////////////////////
327         // determines if temp affordances are valid, if not, access to affordance data is denied
328         // and the entire entityAttemptToAccessAffordance function exits.  Additionally, if the affordance has
329         // timed out, it changes the affordanceValidBoolean to false
330         if (isAffordanceTemp) {
331
332             Double currentSimTime = Schedule.getSimTime(); // use this value to test if affordance is valid
333             Double validTime = Double.valueOf(affordanceUtilities.getAffordanceAttribute(uniqueID, "validTime"));
334             Double cancelTime = Double.valueOf(affordanceUtilities.getAffordanceAttribute(uniqueID, "cancelTime"));
335
336             if (currentSimTime < validTime){
337                 System.out.println("");
338                 System.out.println("TEMPORARY affordance is not valid yet; ACCESS DENIED");
339                 System.out.println("");
340                 return null;
341             }
342
343             if (currentSimTime >= cancelTime){
344                 affordanceUtilities.changeAffordanceValidBooleanToFalse(uniqueID);
345                 System.out.println("");
346                 System.out.println("TEMPORARY affordance has timed out; ACCESS DENIED and affordanceValidBoolean changed
        to False");
347                 System.out.println("");
348                 return null;
349             }
350         }
351
352
353         ///////////////////////////////////////////////////////////////////////////////////////////
354         // this block determines if an entity can access an affordance
355
356         // if the quickAccessClassifier value is "ALL" then it grants the entity access to the data
357         if (quickAccessClassifier.equalsIgnoreCase("ALL")) {
358             accessGranted = true;
359         }
360
361         // determines if entity attempting to access matches the quickAccessClassifier value
362         else if (thisEntitySideType.toString().equalsIgnoreCase(quickAccessClassifier)){
363             accessGranted = true;
364         }
365
366         // if entity does not match quickAccessClassifierValue, it checks if entity is on the tag's GateKeeper access
        list
367         else{
368             List listToCheck = workingGateKeeper.get(uniqueID);
369             if (listToCheck.contains(thisEntityName)) {
370                 accessGranted = true;
371             }
372         }
```

```
373
374
375         ///////////////////////////////////////////////////////////////////////////////////////////////////
376         // returns the data if access has been granted
377         if (accessGranted){
378             return workingTagContainer.get(uniqueID);
379         }
380         else {
381             return null;
382         }
383     }
384 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////
385 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////
386 // end entityAttemptToAccessAffordance function
387
388
389
390
391
392
393
394     /**
395      * this function formats an affordance name (in string format) into a hashmap which is easier to access
396      * this is the primary function to update if one changes the format of an affordance's name
397      *
398      * @param wayPointNameString
399      * @return returns a HashMap that has organized the name of a waypoint
400      */
401     public static HashMap formatWayPointName(String wayPointNameString){
402
403         HashMap<String, Object> formattedWayPointName= new HashMap<>();
404
405         String stringToParse = wayPointNameString;
406
407         String[] arrayOfStrings = stringToParse.split(":");
408
409         String wpNameInSits = arrayOfStrings[0];
410         String affordanceValidBoolean = arrayOfStrings[1];
411         String affordanceType = arrayOfStrings[2];
412         String quickAccessClassifier = arrayOfStrings[3];
413         String validTime = arrayOfStrings[4];
414         String cancelTime = arrayOfStrings[5];
415         String delay = arrayOfStrings[6];
416         String rangeInMeters = arrayOfStrings[7];
417         String dataType = arrayOfStrings[8];
418         String data = arrayOfStrings[9];
419
420         formattedWayPointName.put("wpNameInSits", wpNameInSits);
421         formattedWayPointName.put("affordanceValidBoolean", affordanceValidBoolean);
422         formattedWayPointName.put("affordanceType", affordanceType);
423         formattedWayPointName.put("quickAccessClassifier", quickAccessClassifier);
424         formattedWayPointName.put("validTime", validTime);
425         formattedWayPointName.put("cancelTime", cancelTime);
426         formattedWayPointName.put("delay", delay);
427         formattedWayPointName.put("rangeInMeters", rangeInMeters);
428         formattedWayPointName.put("dataType", dataType);
429         formattedWayPointName.put("data", data);
430
431         return formattedWayPointName;
432     }
433
434
435
436
437
438
```

```
439
440 /**
441  * This affordanceUtilities "internal" function gets an attribute for a specific affordance
442  * @param uniqueID
443  * @param attributeType
444  * @return the desired affordance attribute
445  */
446     public static String getAffordanceAttribute(String uniqueID, String attributeType){
447         String affordanceAttribute = workingTagAttributeContainer.get(uniqueID).get(attributeType).toString();
448         return affordanceAttribute;
449     }
450
451
452
453
454
455
456
457
458     /**
459      * this is an important function that aids in letting an HTN change an specific affordance's affordanceValidBoolean
460      * @param wayPointNameInSits
461      * @return an affordance's uniqueID
462      */
463     public static String getAffordanceUniqueID(String wayPointNameInSits){
464
465         Set setOfAffordanceUniqueIDs = getSetOfAffordanceUniqueIDs();
466         String uniqueID = null;
467
468         for (Object value : setOfAffordanceUniqueIDs) {
469             String testUniqueID = value.toString();
470             String testAttribute = getAffordanceAttribute(testUniqueID, "wpNameInSits");
471             if (testAttribute.equalsIgnoreCase(wayPointNameInSits)){
472                 uniqueID = testUniqueID;
473                 break;
474             }
475         }
476
477         if (uniqueID == null){
478             System.out.println("AN ERROR OCCURRED IN GETAFFORDANCEUNIQUEID");
479         }
480
481         return uniqueID;
482     }
483
484
485
486
487
488
489
490     public static ArrayList<String> getArrayListOfEntityNames(List<String> listOfUnitNames){
491
492         ArrayList<String> arrayListOfEntityNames = new ArrayList<>();
493
494         for (String unitName : listOfUnitNames){
495             OperationalGroup unit = NewUnitHolder.retrieveUnitByName(unitName);
496             List<Entity> listOfEntities = unit.getAllUnitMembers();
497             for (Entity currentEntity : listOfEntities){
498                 String currentEntityName = currentEntity.getAssignedName();
499                 arrayListOfEntityNames.add(currentEntityName);
500             }
501         }
502
503         return arrayListOfEntityNames;
504     }
```

```
505
506
507
508
509
510
511     /**
512      * used in processAffordancesTree, haveEntityListenForAffordances node
513      * @param uniqueID
514      * @return
515      */
516     public static ControlMeasure getControlMeasure(String uniqueID){
517         ControlMeasure currentCM = CMHolder.retrieveControlMeasure(uniqueID);
518         return currentCM;
519     }
520
521
522
523
524
525
526
527
528   /**
529    * gets an affordances's full name
530    * this is used to change the value of an affordanceValidBoolean
531    * @param uniqueID
532    * @return
533    */
534     public static String getFullAffordanceName(String uniqueID){
535         ControlMeasure workingCM = CMHolder.retrieveControlMeasure(uniqueID);
536         String affordanceName = workingCM.getName();
537         return affordanceName;
538     }
539
540
541
542
543
544
545
546     /**
547      * this is used in processAffordancesTree haveEntitiesListenForAffordances node
548      * @param entity
549      * @return
550      */
551     public static List getListOfAffordancesEntityCanAccess(String entity){
552         return workingInvertedGateKeeper.get(entity);
553     }
554
555
556
557
558
559
560
561    /**
562     * this is used to get HTN parameters
563     * IMPORTANT TO NOTE THAT THIS IS A DEEP COPY OF THE HTN PARAMETERS
564     * @param wpNameInSits
565     * @return parameterList
566     */
567     public static List getListOfHtnParameters(String wpNameInSits){
568         List parameterList = htnParameterContainer.get(wpNameInSits);
569         //System.out.println("PARAMETER LIST FROM JAVA");
570         //System.out.println(parameterList.toString());
```

```
571         return parameterList;
572     }
573
574
575
576
577
578
579
580     /**
581      * used in processAffordanceHTN (haveEntityListenForAffordances node)
582      * used in conjunction with getAffordanceUniqueID (optional usage here)
583      * @return setOfAffordanceUniqueIDs
584      */
585
586     public static Set getSetOfAffordanceUniqueIDs(){
587         Set setOfAffordanceUniqueIDs = workingGateKeeper.keySet();
588         return setOfAffordanceUniqueIDs;
589     }
590
591
592
593
594
595
596
597 /**
598  * this is used to initially set and later update (if required) HTN parameters
599  * @param wpNameInSits
600  * @param listOfParameters
601  */
602     public static void updateHtnParameterContainer(String wpNameInSits, List listOfParameters){
603         htnParameterContainer.put(wpNameInSits, listOfParameters);
604         //System.out.println("updateHtnParameterContainer method is complete");
605         //System.out.println(htnParameterContainer.toString());
606     }
607
608
609
610
611
612
613 }
```

D.2 changeAffordanceValidBoolean.xml

The primary function of this HTN is to change an affordance's "affordanceValidBoolean." When used as part of a procedural affordance, this feature can be used to activate, reactivate, or deactivate designated affordances.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="
      changeAffordanceValidBoolean" Type="DEFAULT"><Parent>null</Parent><DataMap><DataKey>typeChange,java.lang.String</
      DataKey><DataKey>affordancesToChangeList,java.util.ArrayList</DataKey></DataMap><Code IsFile="false"/><Import/><
      HTNNode AllowMsg="true" BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>changeAffordanceValidBoolean</Parent
      ><Code IsFile="false">if _gt_activeNode.getVar("isInited") == None:
2   _gt_activeNode.putVar("isInited", 1)
3   _htn_precon_ret=1
4 </Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="makeChanges" Type="GOAL"><Parent>initInfo</Parent><Code
      IsFile="false">from cxxi.model.behavior import OrderUtilities
5 import UtilityFuncs
6 from HTN import UtilityFuncsExp
7 from java.util import Set
```

```
8 from mtry.cxxi.model.affordances import affordanceUtilities
9 import copy
10
11
12
13 typeChange = _gt_activeNode.getParam("typeChange")
14 affordancesToChangeList = _gt_activeNode.getParam("affordancesToChangeList")
15
16
17 # changes affordanceValidBooleans to the opposite of their current value
18 if typeChange == "SWITCH":
19   for wpNameInSits in affordancesToChangeList:
20     wpNameInSitsString = str(wpNameInSits)
21     uniqueID = affordanceUtilities.getAffordanceUniqueID(wpNameInSitsString)
22     affordanceValidBoolean = affordanceUtilities.getAffordanceAttribute(uniqueID, "affordanceValidBoolean")
23     if affordanceValidBoolean == "True":
24       affordanceUtilities.changeAffordanceValidBooleanToFalse(uniqueID)
25       printMessage("changeAffordanceValidBoolian uniqueID "+str(uniqueID)+" changed to False", True)
26     elif affordanceValidBoolean == "False":
27       affordanceUtilities.changeAffordanceValidBooleanToTrue(uniqueID)
28       printMessage("changeAffordanceValidBoolian uniqueID "+str(uniqueID)+" changed to True", True)
29     else:
30       printMessage("!!!!! ERROR OCCURRED IN CHANGEAFFORDANCEVALIDBOOLEAN HTN", True)
31
32
33 # changes all affordanceValidBooleans to True (regardless of their current value)
34 elif typeChange == "ALLTRUE":
35   for wpNameInSits in affordancesToChangeList:
36     wpNameInSitsString = str(wpNameInSits)
37     uniqueID = affordanceUtilities.getAffordanceUniqueID(wpNameInSitsString)
38     affordanceValidBoolean = affordanceUtilities.getAffordanceAttribute(uniqueID, "affordanceValidBoolean")
39     affordanceUtilities.changeAffordanceValidBooleanToTrue(uniqueID)
40     printMessage("changeAffordanceValidBoolian uniqueID "+str(uniqueID)+" changed to True", True)
41
42
43 # changes all affordanceValidBooleans to False (regardless of their current value)
44 elif typeChange == "ALLFALSE":
45   for wpNameInSits in affordancesToChangeList:
46     wpNameInSitsString = str(wpNameInSits)
47     uniqueID = affordanceUtilities.getAffordanceUniqueID(wpNameInSitsString)
48     affordanceValidBoolean = affordanceUtilities.getAffordanceAttribute(uniqueID, "affordanceValidBoolean")
49     affordanceUtilities.changeAffordanceValidBooleanToFalse(uniqueID)
50     printMessage("changeAffordanceValidBoolian uniqueID "+str(uniqueID)+" changed to False", True)
51
52
53 else:
54   printMessage("", False)
55   printMessage("error in changeAffordanceValidBoolean.xml", False)
56   printMessage("", False)</Code><Import/></HTNNode></HTNNode></HTNNode>
```

D.3 processAffordances.xml

When added to an entity's goal stack, this HTN enables entities to detect and process affordances.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="processAffordances"
      Type="DEFAULT"><Parent>null</Parent><Code IsFile="false"/><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="
      initInfo" Type="DEFAULT"><Parent>processAffordances</Parent><Code IsFile="false">if _gt_activeNode.getVar("
      isInited") == None:
2   _gt_activeNode.putVar("isInited", 1)
3   _htn_precon_ret=1
4 </Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="addReplanTriggers" Type="DEFAULT"><Parent>initInfo</Parent
      ><Code IsFile="false"># model events
```

```
5 goalContainer.getCurrentExecutingStack().addReplanTrigger("AtAControlMeasure")
6
7 # HTN events
8 goalContainer.getCurrentExecutingStack().addReplanTrigger("GoalTracker_StopAffordanceTree")</Code><Import/></HTNNode><
      HTNNode AllowMsg="false" BreakPoints="" Name="haveEntityListenForAffordances" Type="DEFAULT"><Parent>initInfo</
      Parent><Code IsFile="false">from cxxi.model.behavior import OrderUtilities
9 import UtilityFuncs
10 from HTN import UtilityFuncsExp
11 from java.util import Set
12 from mtry.cxxi.model.affordances import affordanceUtilities
13
14
15 # this tells entities to listen for affordances they have access to and that do not use the quickAccessClassifier
      feature
16 listOfAffordancesEntityCanAccess = affordanceUtilities.getListOfAffordancesEntityCanAccess(info.getMyAssignedName())
17 if listOfAffordancesEntityCanAccess:
18   printMessage("", False)
19   printMessage("listOfAffordancesEntityCanAccess (not including quickAccessClassifier affordances) = "+str(
      listOfAffordancesEntityCanAccess), True)
20   printMessage("", False)
21   for uniqueID in listOfAffordancesEntityCanAccess:
22     currentCM = affordanceUtilities.getControlMeasure(uniqueID)
23     state.addControlMeasure(currentCM.getName())
24
25
26 # this tell entities to listen for affordances that utilize the appropriate quickAccessClassifier feature
27 setOfAffordanceUniqueIDs = affordanceUtilities.getSetOfAffordanceUniqueIDs()
28 for uniqueID in setOfAffordanceUniqueIDs:
29   affordanceQuickAccessClassifier = affordanceUtilities.getAffordanceAttribute(uniqueID, "quickAccessClassifier")
30   if affordanceQuickAccessClassifier == "ALL":
31     currentCM = affordanceUtilities.getControlMeasure(uniqueID)
32     state.addControlMeasure(currentCM.getName())
33   elif affordanceQuickAccessClassifier == str(info.getMySide()):
34     currentCM = affordanceUtilities.getControlMeasure(uniqueID)
35     state.addControlMeasure(currentCM.getName())</Code><Import/></HTNNode><HTNNode AllowMsg="false" BreakPoints="" Name=
      "endInit" Type="INTERRUPT"><Parent>initInfo</Parent><Code IsFile="false">printMessage("processAffordancesTree INIT
       FINISHED", True)
36 printMessage("", False)</Code><Import/></HTNNode></HTNNode><HTNNode AllowMsg="true" BreakPoints="" Name="events" Type="
      DEFAULT"><Parent>processAffordances</Parent><Code IsFile="false">_htn_precon_ret=1</Code><Import/><HTNNode
      AllowMsg="true" BreakPoints="" Name="isGoalTrackerEvent" Type="DEFAULT"><Parent>events</Parent><Code IsFile="false
      ">if state.getLastTrigger().startswith("doGoalTracker_"):
37   _htn_precon_ret=1</Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="isStopAffordanceTree" Type="DEFAULT"><
      Parent>isGoalTrackerEvent</Parent><Code IsFile="false">if state.getLastTrigger() == "
      doGoalTracker_StopAffordanceTree":
38   _htn_precon_ret=1</Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="stopAffordanceTree" Type="GOAL"><Parent
      >isStopAffordanceTree</Parent><Code IsFile="false"># this node provides the option of having an entity stop
      listening for affordances
39
40 printMessage("", False)
41 printMessage("ENTITY HAS STOPPED ITS AFFORDANCE TREE", True)
42 printMessage("", False)</Code><Import/></HTNNode></HTNNode></HTNNode><HTNNode AllowMsg="true" BreakPoints="" Name="
      modelEvents" Type="DEFAULT"><Parent>events</Parent><Code IsFile="false">_htn_precon_ret=1</Code><Import/><HTNNode
      AllowMsg="true" BreakPoints="" Name="isAtAControlMeasure" Type="DEFAULT"><Parent>modelEvents</Parent><Code IsFile=
      "false">if state.getLastTrigger() == "doAtAControlMeasure":
43   _htn_precon_ret=1</Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="isAffordance" Type="DEFAULT"><Parent>
      isAtAControlMeasure</Parent><Code IsFile="false">from cxxi.model.behavior import OrderUtilities
44 import UtilityFuncs
45 from HTN import UtilityFuncsExp
46 from java.util import Set
47 from mtry.cxxi.model.affordances import affordanceUtilities
48
49
50 # this code is triggered because an entity has come within range of a control measure
51 # this code determines if that control measure is an affordance
52
53 currentCM = state.getLastTriggerParams()[0]
```

```
54 uniqueID = currentCM.getID()
55
56 if affordanceUtilities.determineIfAffordance(uniqueID):
57   _gt_activeNode.putVar("currentCM", currentCM)
58   _gt_activeNode.putVar("uniqueID", uniqueID)
59   _htn_precon_ret=1</Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="isEntityHaveAccessToAffordance" Type="
       DEFAULT"><Parent>isAffordance</Parent><Code IsFile="false">from cxxi.model.behavior import OrderUtilities
60 import UtilityFuncs
61 from HTN import UtilityFuncsExp
62 from java.util import Set
63 from mtry.cxxi.model.affordances import affordanceUtilities
64
65
66 uniqueID = _gt_activeNode.getVar("uniqueID")
67
68
69 # checks to see if an entity can access an affordance
70 # if access is access is not granted, then affordanceData is null
71 affordanceData = affordanceUtilities.entityAttemptToAccessAffordance(info.getMySelf(), uniqueID)
72
73 _gt_activeNode.putVar("affordanceData", affordanceData)
74
75 if affordanceData:
76   _htn_precon_ret=1
77
78 elif not affordanceData:
79   printMessage("", False)
80   printMessage("access denied to affordance number"+str(uniqueID), True)
81   printMessage("", False)
82
83 else:
84   printMessage("ERROR IN isEntityHaveAccessToAffordance", True)
85 </Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="executeAffordance" Type="DEFAULT"><Parent>
       isEntityHaveAccessToAffordance</Parent><Code IsFile="false">from cxxi.model.behavior import OrderUtilities
86 import UtilityFuncs
87 from HTN import UtilityFuncsExp
88 from java.util import Set
89 from mtry.cxxi.model.affordances import affordanceUtilities
90 import copy
91
92
93
94 # gets information about the affordance that will be used in its execution
95 uniqueID = _gt_activeNode.getVar("uniqueID")
96 dataType = affordanceUtilities.getAffordanceAttribute(uniqueID, "dataType")
97 affordanceType = affordanceUtilities.getAffordanceAttribute(uniqueID, "affordanceType")
98 wpNameInSits = affordanceUtilities.getAffordanceAttribute(uniqueID, "wpNameInSits")
99
100
101 printMessage("", False)
102 printMessage("", False)
103 printMessage("ACCESS GRANTED TO AFFORDANCE", True)
104 printMessage("affordanceType = "+str(affordanceType), True)
105 printMessage("wpNameInSits = "+str(wpNameInSits), True)
106
107
108
109
110 # this block of code runs if the Affordance is of type PROCEDURE
111 if dataType == "PROCEDURE":
112
113   currentParameterList = affordanceUtilities.getListOfHtnParameters(wpNameInSits)
114   shallowCopyCurrentParameterList = copy.copy(currentParameterList)
115   delay = affordanceUtilities.getAffordanceAttribute(uniqueID, "delay")
116   delayFloat = float(delay)
117   affordanceData = _gt_activeNode.getVar("affordanceData")
```

```
118   goalPath = affordanceData
119
120   printMessage("executing a procedural affordance", True)
121   printMessage("goalPath = "+str(goalPath), True)
122   printMessage("affordance parameter list = "+str(currentParameterList), True)
123
124   UtilityFuncsExp.addGoal(
125     info.getMyAssignedName(),
126     delayFloat,
127     goalPath,
128     shallowCopyCurrentParameterList,
129     None)
130
131   if affordanceType == "ONESHOT":
132     affordanceUtilities.changeAffordanceValidBooleanToFalse(uniqueID)
133     printMessage("AFFORDANCE VALID BOOLEAN CHANGED TO FALSE", True)
134
135
136
137
138 # this block of (yet to be written) code runs if the Affordance is of type INFORMATION
139 elif dataType == "INFORMATION":
140   printMessage("attempting to execute an informational affordance", True)
141
142
143
144
145 else:
146   printMessage("ERROR IN executeAffordance", True)</Code><Import/></HTNNode></HTNNode><HTNNode AllowMsg="false"
      BreakPoints="" Name="stopListeningForAffordance" Type="DEFAULT"><Parent>isAffordance</Parent><Code IsFile="false">
      # stops entity from listening for this Affordance
147 # IMPORTANT TO REALIZE REPERCUSSIONS OF THIS CODE
148 # IF AN ENTITY COMES ACROSS AN AFFORDANCE ONCE AND IS DENIED ACCESS,
149 # THEN THE ENTITY WILL NO LONGER BE LISTENING FOR THE AFFORDANCE
150
151 currentCM = _gt_activeNode.getVar("currentCM")
152 state.removeControlMeasure(currentCM.getName())
153
154 printMessage("stopListeningForAffordance node finished", True)</Code><Import/></HTNNode><HTNNode AllowMsg="false"
      BreakPoints="" Name="noWorkLeft" Type="INTERRUPT"><Parent>isAffordance</Parent><Code IsFile="false">printMessage("
      no work left", True)</Code><Import/></HTNNode></HTNNode></HTNNode></HTNNode></HTNNode></HTNNode>
```

D.4 stopProcessAffordancesHTN.xml

When added to an entity's goal stack, this HTN brings the entity's *processAffordances* HTN to a goal state, thus preventing the entity from detecting and accessing additional affordances.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="
      stopProcessAffordancesHTN" Type="DEFAULT"><Parent>null</Parent><Code IsFile="false"/><Import/><HTNNode AllowMsg="
      true" BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>stopProcessAffordancesHTN</Parent><Code IsFile="false"
      >if _gt_activeNode.getVar("isInited") == None:
2    _gt_activeNode.putVar("isInited", 1)
3    _htn_precon_ret=1
4  </Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="fireStopAffordanceTree" Type="GOAL"><Parent>initInfo</
      Parent><Code IsFile="false">from HTN import UtilityFuncsExp
5
6  printMessage("", False)
7  printMessage("fireStopAffordanceTree node activated", True)
8  printMessage("", False)
9
10 UtilityFuncsExp.scheduleEvent(
```

```
11     info.getMyAssignedName(),
12     "GoalTracker_StopAffordanceTree",
13     0.001,
14     None)
15
16
17
18 </Code><Import/></HTNNode></HTNNode></HTNNode>
```

D.5 templateAffordanceInitialization.py

This script can be used as a template for the creation of scenario-specific *affordanceInitialization* files.

```
1  from mtry.cxxi.model.HierarchicalTaskNetwork import GoalContainer
2  from cxxi.model.objects.holders import CMHolder
3  from java.util import ArrayList
4  from cxxi.model.behavior import OrderUtilities
5  import UtilityFuncs
6  from HTN import UtilityFuncsExp
7  from java.util import Set
8  from mtry.cxxi.model.affordances import affordanceUtilities
9  import math
10
11
12
13 # print a debug message
14 print ""
15 print "start affordanceInitialization"
16 print ""
17
18
19
20 ####################################################################################
21 # 5 STEP PROCESS TO SET UP SCENARIO AFFORDANCES
22 ####################################################################################
23
24
25
26 # STEP 1 - SITS
27 ####################################################################################
28 # Place (normal) waypoints in SITS scenario file that correspond to desired affordance locations
29 # Give these waypoints a name that is in accordance with your affordance naming convention (ie - affordance1a)
30 # If using HTNs that require locations as parameters, place create these waypoints as well
31 # If desired, the waypoints marking the affordance location can double as an HTN parameter
32
33
34
35 # STEP 2 - WRITE CODE TO INITIALIZE AFFORDANCES
36 ####################################################################################
37 #
38 # THREE PART PROCESS
39 # A: create the affordance using affordanceUtilities.createNewAffordance() method
40 # B: create a list of parameters to be used with an HTN (if required)
41 # C: store the list of HTN parameters using affordanceUtilities.updateHtnParameterContainer() method (if required)
42 #
43 # EXPLANATION OF PARAMETERS USED WHEN CREATING A NEW AFFORDANCE
44 # affordanceUtilities.createNewAffordance(  wayPointNameFromSits where affordance is to be located,
45 #                  list of entity names who can access affordance (can be empty)
46 #                  list of unit names whose entities can access affordance (can be empty)
47 #                  affordanceValidBoolean (True or False),
48 #                  affordanceType (PERMANENT, ONESHOT, OR TEMPORARY),
49 #                  quickAccessClassifier (NONE, ALL, BLUE, RED, GREEN, etc...),
```

```
50 #                  time in seconds at which a TEMPORARY affordance becomes valid,
51 #                  time in seconds at which a TEMPORARY affordance becomes invalid,
52 #                  delay in seconds until a PROCEDURE affordance adds an HTN to an entity's goal stack
53 #                  rangeInMeters at which an entity can sense that an affordance exists,
54 #                  type of affordance (PROCEDURE or INFORMATION)
55 #                  data (for a PROCEDURE affordance this will usually be the location of an HTN)
56
57
58
59 # STEP 3 - INVERTED GATEKEEPER
60 ################################################################################
61 # creates an invertedGateKeeper which is used in the HTN to tell entities what affordances to listen for
62 # this method must be run after the affordances have been initialized
63 # this method must be run before processAffordances.xml is added to each entity's goal stack
64 affordanceUtilities.createInvertedGateKeeper()
65
66
67
68 # STEP 4 - TELL ENTITIES TO LISTEN FOR AFFORDANCES
69 ################################################################################
70 # enable designated entities to listen for affordances
71 # must be done for all entities that need to access affordances
72 # must be done after the invertedGateKeeper has been created
73 # NOTE: this affordance is added to an entity's "affordanceStack" and not its primary goal stack
74
75
76 listOfEntityNames = []
77 for entityName in listOfEntityNames:
78   UtilityFuncsExp.addSpecificGoal(
79       entityName,
80       1.0,
81       "HTN/Trees/processAffordances.xml",
82       "affordanceStack",
83       [],
84       None)
85
86
87
88 listOfUnitNames = []
89 for unitName in listOfUnitNames:
90   UtilityFuncsExp.addSpecificGoalToUnit(
91       unitName,
92       1.0,
93       "HTN/Trees/processAffordances.xml",
94       "affordanceStack",
95       [],
96       None)
97
98
99
100 # STEP 5 - import this script into the scenario's jump start file
101 ################################################################################
102
103
104
105 # print a debug message
106 print ""
107 print "affordanceInitialization complete"
108 print ""
```

APPENDIX E:
Core Security Formation Behavior Files

These are the primary files utilized to create the security formation behavior.

E.1 securityFormationLibrary.py

This script contains functions that are utilized by the security formation behavior files.

```
1  import math
2  from java.util import Vector
3  from HTN import UtilityFuncsExp
4  from cxxi.model.knowledge.group.holders import NewUnitHolder
5
6
7
8  # assigns the appropriate moveOrderHTN to units contained in the 'orderedMasterDapVector'
9  # this function is called in 'masterSecFormV1.xml' and in 'unitMoveOrderHTN.xml'
10 def assignSecurityFormationPositionsToUnits(orderedMasterDapVector, perimeterLocationsVector,
       directionsToLocationsVector, unitFirstPositionNumber, debugBoolean):
11   nextUnitFirstPositionNumber = unitFirstPositionNumber
12
13   if debugBoolean:
14     print "orderedMasterDapVector"
15     print orderedMasterDapVector
16
17   # this loop cycles through each DAP in the orderedMasterDapVector
18   for i in range(len(orderedMasterDapVector)):
19
20     # determines the locations the unit is responsible for occupying
21     # these locations were originated within masterSecurityFormation.xml
22     currentUnitFirstPositionNumber = nextUnitFirstPositionNumber
23     currentUnitName = orderedMasterDapVector[i][0]
24     currentUnitParentValue = orderedMasterDapVector[i][1]
25     numberEntitiesInCurrentUnit = orderedMasterDapVector[i][2]
26     nextUnitFirstPositionNumber = currentUnitFirstPositionNumber + numberEntitiesInCurrentUnit
27
28     currentUnit = NewUnitHolder.retrieveUnitByName(currentUnitName) # of type OrganizationalUnit
29
30     # this collects information on the unit itself (does not include subordinate units)
31     currentUnitMembers = currentUnit.getMembers()
32     numberMembersInCurrentUnit = len(currentUnitMembers)
33
34     # this collects information on the entire unit (including subordinate units)
35     totalCurrentUnitMembers = currentUnit.getAllUnitMembers()
36     totalNumberMembersInCurrentUnit = len(totalCurrentUnitMembers)
37
38     if debugBoolean:
39       print "CURRENT UNIT NAME"
40       print currentUnitName
41       print "NUMBER OF MEMBERS IN CURRENT UNIT"
42       print numberMembersInCurrentUnit
43       print "TOTAL NUMBER OF MEMBERS IN CURRENT UNIT"
44       print totalNumberMembersInCurrentUnit
45
46
47     # a PARENT unit is by its nature a SIMPLE unit
48     # if the PARENT unit has members in it, then it will execute simpleUnitMoveOrderHTN.xml
49     if currentUnitParentValue == "PARENT" and numberMembersInCurrentUnit > 0:
```

```
50
51          firstEntityFromCurrentUnitMembers = currentUnitMembers[0]
52          entityName = firstEntityFromCurrentUnitMembers.getAssignedName()
53
54          UtilityFuncsExp.addGoal(
55              str(entityName),
56              0.1,
57              "HTN/Trees/simpleUnitSecurityFormation.xml",
58              [perimeterLocationsVector, directionsToLocationsVector, currentUnitFirstPositionNumber],
59              None)
60
61
62      elif currentUnitParentValue == "PARENT" and not numberMembersInCurrentUnit > 0 and debugBoolean:
63          print "simpleUnitMoveOrderHTN not added to parent unit because parent unit was destroyed
        ################################"
64
65
66      # a CHILD unit could be either SIMPLE or COMPLEX
67      # if the CHILD unit has members in it, then it will execute unitMoveOrderHTN.xml
68      elif currentUnitParentValue == "CHILD" and totalNumberMembersInCurrentUnit > 0:
69
70          firstEntityFromTotalCurrentUnitMembers = totalCurrentUnitMembers[0]
71          entityName = firstEntityFromTotalCurrentUnitMembers.getAssignedName()
72
73          UtilityFuncsExp.addGoal(
74              str(entityName),
75              0.1,
76              "HTN/Trees/unitSecurityFormation.xml",
77              [perimeterLocationsVector, directionsToLocationsVector, currentUnitFirstPositionNumber, currentUnitName],
78              None)
79
80
81      elif currentUnitParentValue == "CHILD" and not totalNumberMembersInCurrentUnit > 0 and debugBoolean:
82          print "unitMoveOrderHTN not added to child unit because entire unit was destroyed
        ####################################"
83
84
85
86
87
88
89
90
91  # angles must be entered in radians
92  def calculateSecurityFormationPositions(arcBetweenPersonnel, formationOrientation, angleToCover, numberActivePersonnel,
        debugBoolean):
93
94          degreesInCircle = 360
95          radiansInCircle = 2 * math.pi
96
97          if debugBoolean:
98                  print("FUNCTION PARAMETERS")
99                  print("arcBetweenPersonnel =",arcBetweenPersonnel)
100                 print("formationOrientation =",formationOrientation)
101                 print("angleToCover =",angleToCover)
102                 print("numberActivePersonnel =",numberActivePersonnel)
103                 print()
104
105         atcPercentOfCircle = angleToCover/radiansInCircle
106
107         scaledNumberActivePersonnel = numberActivePersonnel / atcPercentOfCircle
108         #print("scaledNumberActivePersonnel =",scaledNumberActivePersonnel)
109
110         circleCircumference = scaledNumberActivePersonnel * arcBetweenPersonnel
111         #print("circleCircumference =",circleCircumference)
112
```

```
113         circleRadius = circleCircumference / (2*math.pi)
114         #print("circleRadius = ",circleRadius)
115 
116         individualAngleResponsibility = angleToCover / numberActivePersonnel
117         #print("individualAngleResponsibility =",individualAngleResponsibility)
118 
119         directionToCounterClockwiseLimit = formationOrientation - (angleToCover/2)
120         #print("directionToCounterClockwiseLimit =",directionToCounterClockwiseLimit)
121 
122         directionToFirstPosition = directionToCounterClockwiseLimit + (individualAngleResponsibility/2)
123         #print("directionToFirstPosition =",directionToFirstPosition)
124 
125         if debugBoolean:
126                 print()
127                 print("LIST POSITION INFORMATION")
128         listOfPolarPlotsToPositions = []
129         for i in range(numberActivePersonnel):
130                 directionToCurrentPosition = directionToCounterClockwiseLimit + (individualAngleResponsibility/2) + i*(
        individualAngleResponsibility)
131                 if directionToCurrentPosition < 0:
132                         directionToCurrentPosition = directionToCurrentPosition + 2*math.pi
133                 polarToCurrentPosition = [directionToCurrentPosition, circleRadius]
134                 if debugBoolean:
135                         print("polar plot to position",i+1,"is",polarToCurrentPosition)
136                 listOfPolarPlotsToPositions.append(polarToCurrentPosition)
137 
138         if debugBoolean:
139                 print()
140                 print()
141 
142         return listOfPolarPlotsToPositions
143 
144 
145 
146 
147 
148 
149 
150 # the "dap" or "decision assignment profile" was created to store relevant information about units
151 # this "information holder" will be important to the process of intelligently assigning units to positions that suit
        their capabilities
152 # currently it only includes the number of personnel in the unit and whether or not it is a Parent or Child unit
153 # eventually, I could see adding information about what type of weapons the unit has, ammunition levels, entity health,
        etc...
154 # MIGHT WANT TO THINK ABOUT CREATING 'GETTERS' FOR DAP VALUES
155 def createUnitDecisionAssignmentProfile(currentUnit, parentEnum):
156   currentUnitName = currentUnit.getName()
157   if parentEnum == "PARENT":
158     numberEntitiesInCurrentUnit = determineNumberEntitiesInSimpleUnit(currentUnit)
159   elif parentEnum == "CHILD":
160     numberEntitiesInCurrentUnit = determineNumberEntitiesInUnit(currentUnit)
161   else:
162     return "ERROR IN DETERMINING DECISION ASSIGNMENT PROFILE"
163   unitDapVector = Vector()
164   unitDapVector.add(currentUnitName)
165   unitDapVector.add(parentEnum)
166   unitDapVector.add(numberEntitiesInCurrentUnit)
167   return unitDapVector
168 
169 
170 
171 
172 
173 
174 
175 
```

```
176 # used to determine number of entities in a simple unit (unit with no sub-units)
177 # there are simpler ways to accomplish this such as using the 'getMembers' command
178 # this script is being kept because it could eventually be used to help build a unit's Decision Assignment Profile
179 def determineNumberEntitiesInSimpleUnit(currentUnit):
180
181   #currentUnitEntities = currentUnit.getAllEntities() # need to verify that this is the correct call
182   currentUnitEntities = currentUnit.getMembers() # using this one at Imre's suggestion
183
184   numberCurrentUnitEntities = len(currentUnitEntities)
185   return numberCurrentUnitEntities
186
187
188
189
190
191
192
193 # used to determine number of entities in a unit which is potentially complex (has sub-units); heavy recursion here
194 # there are simpler ways to accomplish this such as using the 'getAllUnitMembers' command
195 # this script is being kept because it could eventually be used to help build a unit's Decision Assignment Profile
196 def determineNumberEntitiesInUnit(currentUnit):
197   totalEntitiesInCurrentUnit = 0
198   currentUnitGroups = currentUnit.getGroups()
199   numberGroupsInCurrentUnit = len(currentUnitGroups)
200   if numberGroupsInCurrentUnit > 0:
201     for i in range(numberGroupsInCurrentUnit):
202       totalEntitiesInThisSubUnit = 0
203       if len(currentUnitGroups[i].getGroups()) > 0:
204         totalEntitiesInThisSubUnit = determineNumberEntitiesInUnit(currentUnitGroups[i])
205       else:
206         totalEntitiesInThisSubUnit = determineNumberEntitiesInSimpleUnit(currentUnitGroups[i])
207       totalEntitiesInCurrentUnit = totalEntitiesInCurrentUnit + totalEntitiesInThisSubUnit
208     totalEntitiesInCurrentUnit = totalEntitiesInCurrentUnit + determineNumberEntitiesInSimpleUnit(currentUnit)
209   elif numberGroupsInCurrentUnit == 0:
210     totalEntitiesInCurrentUnit = determineNumberEntitiesInSimpleUnit(currentUnit)
211   else:
212     return "ERROR IN DETERMINING NUMBER ENTITIES IN UNIT"
213   return totalEntitiesInCurrentUnit
214
215
216
217
218
219
220
221 # simple math function performed a few times in the HTNs
222 def flipDegreeDirection180(directionInDegrees):
223   directionInDegrees = directionInDegrees + 180
224   if directionInDegrees >= 360:
225     directionInDegrees = directionInDegrees - 360
226   return directionInDegrees
```

E.2 masterSecurityFormation.xml

This is the primary HTN that is utilized to instantiate a security formation. *masterSecurityFormation* conducts the majority of the calculations needed to determine a security formation's geometry. This file activates the *unitSecurityFormation* HTN.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="
      masterSecurityFormation" Type="DEFAULT"><Parent>null</Parent><DataMap><DataKey>recursionDescriptor,java.lang.
      String,"RECURSIVE" or "NONRECURSIVE"</DataKey><DataKey>arrayListOfUnitNames,java.util.ArrayList,names of units to
      participate in formation</DataKey><DataKey>wayPoint,cxxi.model.objects.features.CMWayPoint</DataKey><DataKey>
```

```
    arcBetweenLocations , java . lang . Double , meters</DataKey><DataKey>formationOrientation , java . lang . Double , radians</
    DataKey><DataKey>angleToCover , java . lang . Double , radians</DataKey><DataKey>formationCenterDescriptor , java . lang .
    String , "FLOT" or "CENTER"</DataKey><DataKey>convexConcaveString , java . lang . String , "CONVEX" (normal) or "CONCAVE" (
    inverted formation)</DataKey><DataKey>useIsCommanderBoolean , java . lang . String , "True" or "False"</DataKey></DataMap>
    <Code IsFile="false"/><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>
    masterSecurityFormation</Parent><Code IsFile="false">if _gt_activeNode.getVar("isInited") == None:
2   _gt_activeNode.putVar("isInited", 1)
3   _htn_precon_ret=1
4 </Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="isCommander" Type="DEFAULT"><Parent>initInfo</Parent><Code
     IsFile="false">useIsCommanderBoolean = _gt_activeNode.getParam("useIsCommanderBoolean")
5
6 if useIsCommanderBoolean == "True":
7   if state.isCommander():
8     _htn_precon_ret=1
9 elif useIsCommanderBoolean == "False":
10   _htn_precon_ret=1
11 else:
12   printMessage("ERROR WITH useIsCommanderBoolean", True)</Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="
    getNumberPersonnel" Type="DEFAULT"><Parent>isCommander</Parent><Code IsFile="false">from cxxi.model.knowledge.
    group.holders import NewUnitHolder
13 from HTN import securityFormationLibrary
14
15
16
17
18 arrayListOfUnitNames = _gt_activeNode.getParam("arrayListOfUnitNames")
19 recursionDescriptor = _gt_activeNode.getParam("recursionDescriptor")
20
21
22 numberTotalEntities = 0
23 for i in range(len(arrayListOfUnitNames)):
24   currentUnitName = arrayListOfUnitNames[i]
25   currentUnit = NewUnitHolder.retrieveUnitByName(currentUnitName) # of type OrganizationalUnit
26
27   if recursionDescriptor == "RECURSIVE":
28     numberEntitiesInCurrentUnit = securityFormationLibrary.determineNumberEntitiesInUnit(currentUnit)
29   elif recursionDescriptor == "NONRECURSIVE":
30     numberEntitiesInCurrentUnit = securityFormationLibrary.determineNumberEntitiesInSimpleUnit(currentUnit)
31   else:
32     printMessage("ERROR IN ENTERING RECURSION DESCRIPTOR", True)
33
34   numberTotalEntities = numberTotalEntities + numberEntitiesInCurrentUnit
35
36
37   ######################################################
38   ## I chose not to use these commands on purpose
39   ## I believe using the above commands presents an opportunity to build on the "DAP" concept
40   #members = currentUnit.getAllUnitMembers()
41   #alternateNumberMembers = len(members)
42   #printMessage("alternate numberTotalEntities = "+str(alternateNumberMembers), True)
43   #############################################################
44
45
46
47
48 printMessage("numberTotalEntities = "+str(numberTotalEntities), True)
49
50
51
52
53
54 _gt_activeNode.putVar("numberTotalEntities", numberTotalEntities)
55
56
57 </Code><Import/></HTNNode><HTNNode AllowMsg="true" BreakPoints="" Name="isDestroyedUnit" Type="DEFAULT"><Parent>
    isCommander</Parent><Code IsFile="false">numberTotalEntities = _gt_activeNode.getVar("numberTotalEntities")
```

```
58
59 if numberTotalEntities == 0:
60   _htn_precon_ret=1</Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="exitHTN" Type="GOAL"><Parent>
       isDestroyedUnit</Parent><Code IsFile="false">
61 printMessage("NO ENTITIES IN UNIT; MASTER SECURITY FORMATION EXITING", True)
62
63 </Code><Import/></HTNNode></HTNNode><HTNNode AllowMsg="false" BreakPoints="" Name="calculatePolars" Type="DEFAULT"><
       Parent>isCommander</Parent><Code IsFile="false">from HTN import securityFormationLibrary
64 import math
65
66 numberTotalEntities = _gt_activeNode.getVar("numberTotalEntities")
67 arcBetweenLocations = _gt_activeNode.getParam("arcBetweenLocations")
68 formationOrientation = _gt_activeNode.getParam("formationOrientation")
69 angleToCover = _gt_activeNode.getParam("angleToCover")
70
71
72 printMessage("numberTotalEntities = "+str(numberTotalEntities), True)
73
74 listOfPolarPlotsToPositions = securityFormationLibrary.calculateSecurityFormationPositions(
75   arcBetweenLocations,
76   formationOrientation,
77   angleToCover,
78   numberTotalEntities,
79   False)
80
81 _gt_activeNode.putVar("listOfPolarPlotsToPositions", listOfPolarPlotsToPositions)
82
83 printMessage("listOfPolarPlotsToPositions = "+str(listOfPolarPlotsToPositions), True)
84 </Code><Import/></HTNNode><HTNNode AllowMsg="true" BreakPoints="" Name="findPerimeterLocations" Type="DEFAULT"><Parent>
       isCommander</Parent><Code IsFile="false">from mtry.cxxi.model.HierarchicalTaskNetwork import HTNUtilities
85 from java.util import Vector
86 from cxxi.model.behavior import OrderUtilities
87 import math
88 from HTN import securityFormationLibrary
89
90
91 # get parameters and variables
92 inputtedPosition = _gt_activeNode.getParam("wayPoint")
93 formationCenterDescriptor = _gt_activeNode.getParam("formationCenterDescriptor")
94 formationOrientation = _gt_activeNode.getParam("formationOrientation")
95 listOfPolarPlotsToPositions = _gt_activeNode.getVar("listOfPolarPlotsToPositions")
96 circleRadius = listOfPolarPlotsToPositions[0][1]
97 convexConcaveString = _gt_activeNode.getParam("convexConcaveString")
98
99 # establish new variables
100 perimeterLocationsVector = Vector()
101 directionsToLocationsVector = Vector()
102
103
104
105
106
107 # convert orientation from radians to degrees
108 formationOrientationInDegrees = (formationOrientation/math.pi)*180
109
110
111
112 # assesses formation Center Descriptor value (center or forward line of troops)
113 if formationCenterDescriptor == "CENTER":
114   circleCenterLocationWithAGL = inputtedPosition.getLocationWithAGL()
115 elif formationCenterDescriptor == "FLOT":
116   if convexConcaveString == "CONVEX":
117     initialCenterLocationWithAGL = inputtedPosition.getLocationWithAGL()
118     oppositeFormationOrientationInDegrees = securityFormationLibrary.flipDegreeDirection180(
         formationOrientationInDegrees)
119     circleCenterLocationWithAGL = HTNUtilities._py_offsetLoc(
```

```
120         initialCenterLocationWithAGL,
121         oppositeFormationOrientationInDegrees,
122         circleRadius);
123     elif convexConcaveString == "CONCAVE":
124       initialCenterLocationWithAGL = inputtedPosition.getLocationWithAGL()
125       oppositeFormationOrientationInDegrees = securityFormationLibrary.flipDegreeDirection180(
          formationOrientationInDegrees)
126       circleCenterLocationWithAGL = HTNUtilities._py_offsetLoc(
127         initialCenterLocationWithAGL,
128         formationOrientationInDegrees,
129         circleRadius);
130     else:
131       printMessage("ERROR IN ENTERING convexConcaveDescriptor", True)
132 else:
133   printMessage("ERROR IN ENTERING FORMATION CENTER DESCRIPTOR", True)
134
135
136
137
138
139 # determines the locations of the security positions
140 for i in range(len(listOfPolarPlotsToPositions)):
141
142   #printMessage("polarToPosition = "+str(listOfPolarPlotsToPositions[i]), True)
143
144   direction = listOfPolarPlotsToPositions[i][0]
145   directionInDegrees = (direction/math.pi)*180
146
147
148   # activated if formation is of type "CONCAVE"
149   if convexConcaveString == "CONCAVE":
150     directionInDegrees = securityFormationLibrary.flipDegreeDirection180(directionInDegrees)
151
152
153   distance = listOfPolarPlotsToPositions[i][1]
154   currentPerimeterLocation = HTNUtilities._py_offsetLoc(circleCenterLocationWithAGL, directionInDegrees, distance);
155   perimeterLocationsVector.add(currentPerimeterLocation)
156
157
158   # this conditional is not actually needed because 'directionsInDegrees' is not used again
159   # but I kept it in because it helps remind me how these CONCAVE calculations work
160   #if convexConcaveString == "CONCAVE":
161   # directionInDegrees = dynamicManeuverWork.flipDegreeDirection180(directionInDegrees)
162
163
164   directionsToLocationsVector.add(direction)
165
166
167
168 # saves the locations of the security positions
169 _gt_activeNode.putVar("perimeterLocationsVector", perimeterLocationsVector)
170 _gt_activeNode.putVar("directionsToLocationsVector", directionsToLocationsVector)
171
172
173 </Code><Import/></HTNNode><HTNNode AllowMsg="false" BreakPoints="" Name="activateSubordinateHTNs" Type="GOAL"><Parent>
       isCommander</Parent><Code IsFile="false">from HTN import UtilityFuncsExp
174 from HTN import securityFormationLibrary
175
176 recursionDescriptor = _gt_activeNode.getParam("recursionDescriptor")
177 perimeterLocationsVector = _gt_activeNode.getVar("perimeterLocationsVector")
178 directionsToLocationsVector = _gt_activeNode.getVar("directionsToLocationsVector")
179 arrayListOfUnitNames = _gt_activeNode.getParam("arrayListOfUnitNames")
180
181
182 # establish new variables
183 unitFirstPositionNumber = 0
```

```
184 unorderedMasterDapVector = Vector() # DAP/Dap stands for "Decision Assignment Profile"
185 orderedMasterDapVector = Vector()
186
187
188 # create DAPs for the top level units and store them inside a vector
189 # DAPs for RECURSIVE and NONRECURSIVE formations should be different
190 # as DAP functionality is increased, it is important to ensure that this function continues to work properly
191 for i in range(len(arrayListOfUnitNames)):
192   currentUnitName = arrayListOfUnitNames[i]
193   currentUnit = NewUnitHolder.retrieveUnitByName(currentUnitName) # of type OrganizationalUnit
194
195   # if using recursive option, classify all units as CHILD - this way they are treated as potentially complex units
196   # if using nonrecursive option, classify all units as PARENT - this way they are treated as simple units
197   if recursionDescriptor == "RECURSIVE":
198     currentUnitDap = securityFormationLibrary.createUnitDecisionAssignmentProfile(currentUnit, "CHILD")
199   elif recursionDescriptor == "NONRECURSIVE":
200     currentUnitDap = securityFormationLibrary.createUnitDecisionAssignmentProfile(currentUnit, "PARENT")
201   else:
202     printMessage("ERROR IN ENTERING RECURSION DESCRIPTOR", True)
203
204
205
206   unorderedMasterDapVector.add(currentUnitDap)
207
208
209
210
211
212 ################################################################################
213 # THIS IS THE LOCATION WHERE THE INTELLIGENT "DECISION" CAN BE MADE CONCERNING
214 # HOW UNITS SHOULD BE PLACED WITHIN THE OVERALL FORMATION
215 # DO THIS BY ORDERING the units described in 'unorderedMasterDapVector'
216 orderedMasterDapVector = unorderedMasterDapVector
217 ################################################################################
218
219
220
221
222
223 securityFormationLibrary.assignSecurityFormationPositionsToUnits(orderedMasterDapVector, perimeterLocationsVector,
        directionsToLocationsVector, unitFirstPositionNumber, False)
224
225 printMessage("MasterHTN Move To Position Node Complete", True)</Code><Import/></HTNNode></HTNNode></HTNNode></HTNNode>
```

E.3 unitSecurityFormation.xml

This is the second HTN that is utilized to instantiate a security formation. *unitSecurityFormation* was designed to work with future mechanisms that will intelligently locate subunits within a single security formation. This file recursively activates itself, or it activates the *simpleUnitSecurityFormation* HTN.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="
        unitSecurityFormation" Type="DEFAULT"><Parent>null</Parent><DataMap><DataKey>perimeterLocationsVector,java.util.
        Vector</DataKey><DataKey>directionsToLocationsVector,java.util.Vector</DataKey><DataKey>unitFirstPositionNumber,
        java.lang.Integer</DataKey><DataKey>unitName,java.lang.String</DataKey></DataMap><Code IsFile="false"/><Import/><
        HTNNode AllowMsg="true" BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>unitSecurityFormation</Parent><Code
        IsFile="false">if _gt_activeNode.getVar("isInited") == None:
2   _gt_activeNode.putVar("isInited", 1)
3   _htn_precon_ret=1
4 </Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="processInformation" Type="GOAL"><Parent>initInfo</Parent>
        <Code IsFile="false">from java.util import Vector
```

```
5 from HTN import securityFormationLibrary
6 from HTN import UtilityFuncsExp
7 from cxxi.model.knowledge.group.holders import NewUnitHolder
8
9
10 printMessage("EXECUTING unitMoveOrderHTN", True)
11
12 perimeterLocationsVector = _gt_activeNode.getParam("perimeterLocationsVector")
13 directionsToLocationsVector = _gt_activeNode.getParam("directionsToLocationsVector")
14 unitFirstPositionNumber = _gt_activeNode.getParam("unitFirstPositionNumber")
15
16 # important to get the unit in this manner; the entity executing this HTN may not be in the unit executing the action
17 unitName = _gt_activeNode.getParam("unitName")
18 currentUnit = NewUnitHolder.retrieveUnitByName(unitName) # of type OrganizationalUnit
19
20 printMessage("unitName to execute = "+str(unitName), True)
21
22 currentUnitGroups = currentUnit.getGroups()
23
24 printMessage("currentUnitGroups = "+str(currentUnitGroups), True)
25
26
27 # if this is a simple unit (no sub-units), then adds simpleUnitMoveOrderHTN to its goal stack
28 if len(currentUnitGroups) == 0:
29
30   printMessage("unitMoveOrderHTN IS EXECUTING A simpleUnitMoveOrderHTN", True)
31
32   currentUnitMembers = currentUnit.getMembers()
33   firstEntityFromCurrentUnitMembers = currentUnitMembers[0]
34   entityName = firstEntityFromCurrentUnitMembers.getAssignedName()
35
36   UtilityFuncsExp.addGoal(
37        str(entityName),
38        0.1,
39        "HTN/Trees/simpleUnitSecurityFormation.xml",
40        [perimeterLocationsVector, directionsToLocationsVector, unitFirstPositionNumber],
41        None)
42
43
44
45
46
47 # activates if this is a complex unit (with sub-units); below code displays recursive properites of unitMoveOrderHTN
48 # dap stands for "decision assignment profile"; it was created to store info about units
49 # see securityFormationLibrary.py for more details
50 elif len(currentUnitGroups) &gt; 0:
51
52   # create new variables
53   unorderedMasterDapVector = Vector()
54   orderedMasterDapVector = Vector()
55
56   # creates a dap for the unit's headquarters element (which itself is a simple unit)
57   parentUnitDap = securityFormationLibrary.createUnitDecisionAssignmentProfile(currentUnit, "PARENT")
58   printMessage("parentUnitDap = "+str(parentUnitDap), True)
59   unorderedMasterDapVector.add(parentUnitDap)
60
61   # creates daps for the unit's sub-units (these units may be simple or complex)
62   for i in range(len(currentUnitGroups)):
63     currentChildUnit = currentUnitGroups[i]
64     currentChildUnitDap = securityFormationLibrary.createUnitDecisionAssignmentProfile(currentChildUnit, "CHILD")
65     unorderedMasterDapVector.add(currentChildUnitDap)
66
67   printMessage("unorderedMasterDapVector = "+str(unorderedMasterDapVector), True)
68
69
70
```

```
71
72
73
74   ############################################################################
75   # THIS IS THE LOCATION WHERE THE INTELLIGENT "DECISION" CAN BE MADE CONCERNING
76   # HOW UNITS SHOULD BE PLACED WITHIN THE PARENT UNIT FORMATION
77   # DO THIS BY ORDERING UNITS DESCRIBED IN 'unorderedMasterDapVector'
78   orderedMasterDapVector = unorderedMasterDapVector
79   ############################################################################
80
81
82
83
84
85
86
87   # this function adds unitMoveOrderHTN/simpleUnitMoveOrderHTN to the units described in the orderedMasterDapVector
88   securityFormationLibrary.assignSecurityFormationPositionsToUnits(
89     orderedMasterDapVector,
90     perimeterLocationsVector,
91     directionsToLocationsVector,
92     unitFirstPositionNumber,
93     False)
94
95
96
97 </Code><Import/></HTNNode></HTNNode></HTNNode>
```

E.4 simpleUnitSecurityFormation.xml

This is the third HTN that is utilized to instantiate a security formation. *simpleUnitSecurityFormation* was designed to work with future mechanisms that will intelligently locate individual entities within a security formation. This file activates the *basicIndividualMove* HTN.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="
      simpleUnitSecurityFormation" Type="DEFAULT"><Parent>null</Parent><DataMap><DataKey>perimeterLocationsVector,java.
      util.Vector</DataKey><DataKey>directionsToLocationsVector,java.util.Vector</DataKey><DataKey>
      unitFirstPositionNumber,java.lang.Integer</DataKey></DataMap><Code IsFile="false"/><Import/><HTNNode AllowMsg="
      true" BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>simpleUnitSecurityFormation</Parent><Code IsFile="
      false">if _gt_activeNode.getVar("isInited") == None:
2   _gt_activeNode.putVar("isInited", 1)
3   _htn_precon_ret=1
4 </Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="processInformation" Type="GOAL"><Parent>initInfo</Parent>
      <Code IsFile="false">printMessage("EXECUTING SIMPLE UNIT MOVE ORDER HTN", True)
5
6 perimeterLocationsVector = _gt_activeNode.getParam("perimeterLocationsVector")
7 directionsToLocationsVector = _gt_activeNode.getParam("directionsToLocationsVector")
8 unitFirstPositionNumber = _gt_activeNode.getParam("unitFirstPositionNumber")
9
10 state.changeFormation("NO_FORMATION", "overrideFillScript")
11
12
13 positionNumberForEntity = unitFirstPositionNumber
14 unitToIter = state.getCurrentUnit()
15
16 totalNumberMembers = 0
17
18
19
20 ##################################################################################################
```

```
21 # assigns individualMoveHTN to members of this unit
22 # right now position assignment is simply made in the order that the entities are listed
23 # to "intelligently" assign entities to positions that match their capabilities, this is a place to do it
24 ##########################################################################################################
25
26
27
28
29 #for member in unitToIter.getAllEntities():
30 for member in unitToIter.getMembers():
31
32   totalNumberMembers = totalNumberMembers + 1
33
34   UtilityFuncsExp.addGoal(
35        str(member.getAssignedName()),
36        0.1,
37        "HTN/Trees/basicIndividualMove.xml",
38        [perimeterLocationsVector[positionNumberForEntity],directionsToLocationsVector[positionNumberForEntity]],
39        None)
40
41   positionNumberForEntity = positionNumberForEntity + 1
42
43   printMessage("MEMBER NAME="+member.getAssignedName(), True)
44
45
46 printMessage("totalNumberMembers = "+str(totalNumberMembers), True)</Code><Import/></HTNNode></HTNNode></HTNNode>
```

E.5 basicIndividualMove.xml

This is the fourth HTN that is utilized to instantiate a security formation. *basicIndividualMove* moves entities to their assigned positions. This HTN is for demonstration purposes and is intended to be replaced by a behavior mechanism which more accurately represents the ways that entities move in combat situations.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="
       basicIndividualMoveHTN" Type="DEFAULT"><Parent>null</Parent><DataMap><DataKey>locationToGoTo,cxxi.model.physics.
       geometry.Location</DataKey><DataKey>directionToLookInRadians,java.lang.Double</DataKey></DataMap><Code IsFile="
       false"/><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>
       basicIndividualMoveHTN</Parent><Code IsFile="false">if _gt_activeNode.getVar("isInited") == None:
2   _gt_activeNode.putVar("isInited", 1)
3   _htn_precon_ret=1
4 </Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="moveToPosition" Type="GOAL"><Parent>initInfo</Parent><
       Code IsFile="false">from cxxi.model.behavior import OrderUtilities
5 from java.util import Vector
6 import math
7
8 printMessage("executing individual move HTN"+str(info.getMyAssignedName()), True)
9
10 locationToGoTo = _gt_activeNode.getParam("locationToGoTo")
11
12 directionToLookInRadians = _gt_activeNode.getParam("directionToLookInRadians")
13
14 directionToLookInDegrees = (directionToLookInRadians/math.pi)*180
15
16 compoundOrder = OrderUtilities.moveIntoPosition(locationToGoTo, 3.0, directionToLookInDegrees)
17
18 orders.addOrder(compoundOrder)
19
20 </Code><Import/></HTNNode></HTNNode></HTNNode>
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX F: Modifiers to Security Formation Behavior

The following HTNs are used to modify the security formation behavior's functionality.

F.1 removeUnitFromAffSecForm.xml

This HTN removes units from security formations which were initially established with a procedural affordance that utilized the *masterSecurityFormation* or *securityFormation_byPhaseAffordanceMod* HTNs.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="removeUnitFromAffSecForm" Type="DEFAULT"><Parent>null</Parent><DataMap><DataKey>wpNameInSits,java.lang.String</DataKey><DataKey>listOfUnitNamesToRemove,java.util.ArrayList</DataKey><DataKey>controlMeasureToMoveTo,cxxi.model.objects.features.CMWayPoint</DataKey><DataKey>newFormationParameterBoolean,java.lang.String,"True" or "False"</DataKey><DataKey>newArcBetweenLocations,java.lang.Double,in meters</DataKey></DataMap><Code IsFile="false"/><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>removeUnitFromAffSecForm</Parent><Code IsFile="false">if _gt_activeNode.getVar("isInited") == None:
2   _gt_activeNode.putVar("isInited", 1)
3   _htn_precon_ret=1
4 </Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="doWork" Type="GOAL"><Parent>initInfo</Parent><Code IsFile="false">from HTN import UtilityFuncsExp
5 from java.util import ArrayList
6 from mtry.cxxi.model.behaviorextensions import RuleMappingSupport
7 from mtry.cxxi.model.affordances import affordanceUtilities
8 import math
9 import random
10 from cxxi.model.knowledge.group.holders import NewUnitHolder
11
12
13
14
15 ########################################################################################
16 # THIS HTN WAS DESIGNED SPECIFICALLY TO WORK WITH SECURITY FORMATIONS THAT
17 # WERE INSTANTIATED BY A PROCEDURAL AFFORDANCE WHICH UTILIZED EITHER
18 # masterSecurityFormation.xml OR securityFormation_byProcessingUnit.xml
19 ########################################################################################
20
21
22
23 wpNameInSits = _gt_activeNode.getParam("wpNameInSits")
24 listOfUnitNamesToRemove = _gt_activeNode.getParam("listOfUnitNamesToRemove")
25 controlMeasureToMoveTo = _gt_activeNode.getParam("controlMeasureToMoveTo")
26 newFormationParameterBoolean = _gt_activeNode.getParam("newFormationParameterBoolean")
27 newArcBetweenLocations = _gt_activeNode.getParam("newArcBetweenLocations")
28
29
30
31 htnParameterList = affordanceUtilities.getListOfHtnParameters(wpNameInSits)
32
33
34 ##########################################################################################
35 # THIS COULD BREAK IF THE HTN PARAMETERS ARE CHANGED
36 # THE INDEX NUMBER MUST CORRESPOND TO THE unitNamesArrayList
37 unitNamesArrayList = htnParameterList[1]
38 ##########################################################################################
39
```

```
40
41 for unitNameToRemove in listOfUnitNamesToRemove:
42   if unitNamesArrayList.contains(unitNameToRemove):
43     unitNamesArrayList.remove(unitNameToRemove)
44
45
46     currentUnit = NewUnitHolder.retrieveUnitByName(unitNameToRemove) # of type OrganizationalUnit
47
48
49
50     #listOfCurrentEntities = currentUnit.getAllEntities()
51     listOfCurrentEntities = currentUnit.getMembers()  # NEED TO VERIFY THIS IS CORRECT METHOD TO CALL
52
53
54
55     printMessage("current entities = "+str(listOfCurrentEntities), False)
56     for currentEntity in listOfCurrentEntities:
57       currentEntityName = currentEntity.getAssignedName()
58
59       #printMessage("currentEntityName = "+str(currentEntityName), False)
60
61       UtilityFuncsExp.addGoal(
62           currentEntityName,
63           0.1,
64           "HTN/Trees/basicIndividualMove.xml",
65           [controlMeasureToMoveTo.getLocation(), random.uniform(0,2.0*math.pi)],
66           None)
67
68
69 htnParameterList = affordanceUtilities.getListOfHtnParameters(wpNameInSits)
70
71
72
73
74 ###############################################################################################
75 # THIS COULD BREAK IF THE HTN PARAMETERS ARE CHANGED
76 # THE INDEX NUMBER MUST CORRESPOND TO THE arcBetweenLocations
77 if newFormationParameterBoolean == "True":
78   printMessage("using new formation parameters", False)
79   htnParameterList[3] = newArcBetweenLocations
80 elif newFormationParameterBoolean == "False":
81   printMessage("not using new formation parameters", False)
82 else:
83   printMessage("error has occurred with newFormationParametersBoolean", False)
84 ###############################################################################################
85
86
87
88 UtilityFuncsExp.addGoal(
89    info.getMyAssignedName(),
90    0.1,
91    "HTN/Trees/masterSecurityFormation.xml",
92    htnParameterList,
93    None)</Code><Import/></HTNNode></HTNNode></HTNNode>
```

F.2 securityFormation_byPhaseAffordanceMod.xml

This HTN is intended to be utilized in conjunction with a procedural affordance. Units that activate this procedural affordance are integrated into a single security formation.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="
      securityFormation_byPhaseAffordanceMod" Type="DEFAULT"><Parent>null</Parent><DataMap><DataKey>affordanceName,java.
      lang.String</DataKey><DataKey>recursionDescriptor,java.lang.String,"RECURSIVE" or "NONRECURSIVE"</DataKey><DataKey
```

```
>arrayListOfUnitNames,java.util.ArrayList</DataKey><DataKey>wayPoint,cxxi.model.objects.features.CMWayPoint</
DataKey><DataKey>arcBetweenLocations,java.lang.Double</DataKey><DataKey>formationOrientation,java.lang.Double,
radians</DataKey><DataKey>angleToCover,java.lang.Double,radians</DataKey><DataKey>formationCenterDescriptor,java.
lang.String,"FLOT" or "CENTER"</DataKey><DataKey>convexConcaveString,java.lang.String,"CONCAVE" if want the
formation inverted</DataKey><DataKey>useIsCommanderBoolean,java.lang.String,"True" or "False"</DataKey></DataMap><
Code IsFile="false"/><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>
securityFormation_byPhaseAffordanceMod</Parent><Code IsFile="false">if _gt_activeNode.getVar("isInited") == None:
2   _gt_activeNode.putVar("isInited", 1)
3   _htn_precon_ret=1
4 </Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="isCommander" Type="DEFAULT"><Parent>initInfo</Parent><Code
    IsFile="false">useIsCommanderBoolean = _gt_activeNode.getParam("useIsCommanderBoolean")
5
6 if useIsCommanderBoolean == "True":
7   if state.isCommander():
8     _htn_precon_ret=1
9 elif useIsCommanderBoolean == "False":
10   _htn_precon_ret=1
11 else:
12   printMessage("ERROR WITH useIsCommanderBoolean", True)</Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="
    activateMasterSecurityFormation" Type="GOAL"><Parent>isCommander</Parent><Code IsFile="false">from cxxi.model.
    behavior import OrderUtilities
13 import UtilityFuncs
14 from HTN import UtilityFuncsExp
15 from java.util import Set
16 from mtry.cxxi.model.affordances import affordanceUtilities
17
18 import copy
19
20
21 goalPath = "HTN/Trees/masterSecurityFormation.xml"
22
23 affordanceName = _gt_activeNode.getParam("affordanceName")
24
25 currentParameterList = affordanceUtilities.getListOfHtnParameters(affordanceName)
26
27 printMessage("currentParameterList = "+str(currentParameterList), True)
28
29 unitName = state.getCurrentUnitName()
30 printMessage("UNIT NAME = "+str(unitName), True)
31
32
33
34
35 ##########################################################################################
36 # THIS COULD BREAK IF THE HTN PARAMETERS ARE CHANGED
37 # THE INDEX NUMBER MUST CORRESPOND TO THE unitNamesArrayList
38 if not currentParameterList[2].contains(str(unitName)):
39   currentParameterList[2].add(str(unitName))
40   printMessage(str(unitName)+" added to unit parameter list", True)
41 else:
42   printMessage(str(unitName)+" NOT ADDED TO UNIT PARAMETER LIST (DUPLICATE ISSUE)", True)
43 ##########################################################################################
44
45
46
47
48
49 updatedParameterList = affordanceUtilities.getListOfHtnParameters(wpNameInSits)
50
51 shallowCopyUpdatedParameterList = copy.copy(currentParameterList)
52
53 printMessage("shallowCopyUpdatedParameterList = "+str(shallowCopyUpdatedParameterList), True)
54
55 # removes the first item from the shallowCopyUpdatedParameterList
56 # this is done because masterSecurityFormation.xml does not use this item as one of its parameters
57 shallowCopyUpdatedParameterList.pop(0)
```

```
UtilityFuncsExp.addGoal(
   info.getMyAssignedName(),
   0.1,
   goalPath,
   shallowCopyUpdatedParameterList,
   None)
</Code><Import/></HTNNode></HTNNode></HTNNode></HTNNode>
```

F.3 securityFormation_byProcessingUnit.xml

The unit that processes this HTN establishes a security formation.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="
      securityFormation_byProcessingUnit" Type="DEFAULT"><Parent>null</Parent><DataMap><DataKey>recursionDescriptor,java
      .lang.String,"RECURSIVE" or "NONRECURSIVE"</DataKey><DataKey>arrayListOfUnitNames,java.util.ArrayList</DataKey><
      DataKey>wayPoint,cxxi.model.objects.features.CMWayPoint</DataKey><DataKey>arcBetweenLocations,java.lang.Double</
      DataKey><DataKey>formationOrientation,java.lang.Double,radians</DataKey><DataKey>angleToCover,java.lang.Double,
      radians</DataKey><DataKey>formationCenterDescriptor,java.lang.String,"FLOT" or "CENTER"</DataKey><DataKey>
      convexConcaveString,java.lang.String,"CONCAVE" if want the formation inverted</DataKey><DataKey>
      useIsCommanderBoolean,java.lang.String,"True" or "False"</DataKey></DataMap><Code IsFile="false"/><Import/><
      HTNNode AllowMsg="true" BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>securityFormation_byProcessingUnit</
      Parent><Code IsFile="false">if _gt_activeNode.getVar("isInited") == None:
   _gt_activeNode.putVar("isInited", 1)
   _htn_precon_ret=1
</Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="isCommander" Type="DEFAULT"><Parent>initInfo</Parent><Code
       IsFile="false">useIsCommanderBoolean = _gt_activeNode.getParam("useIsCommanderBoolean")

if useIsCommanderBoolean == "True":
   if state.isCommander():
      _htn_precon_ret=1
elif useIsCommanderBoolean == "False":
   _htn_precon_ret=1
else:
   printMessage("ERROR WITH useIsCommanderBoolean", True)</Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="
      activateMasterSecurityFormation" Type="GOAL"><Parent>isCommander</Parent><Code IsFile="false">from cxxi.model.
      behavior import OrderUtilities
import UtilityFuncs
from HTN import UtilityFuncsExp
from java.util import Set
from mtry.cxxi.model.affordances import affordanceUtilities
import copy


goalPath = "HTN/Trees/masterSecurityFormation.xml"

unitName = state.getCurrentUnitName()
printMessage("UNIT NAME = "+str(unitName), True)
unitNamesArrayList = ArrayList()
unitNamesArrayList.add(str(unitName))

recursionDescriptor = _gt_activeNode.getParam("recursionDescriptor")
wayPoint = _gt_activeNode.getParam("wayPoint")
arcBetweenLocations = _gt_activeNode.getParam("arcBetweenLocations")
formationOrientation = _gt_activeNode.getParam("formationOrientation")
angleToCover = _gt_activeNode.getParam("angleToCover")
formationCenterDescriptor = _gt_activeNode.getParam("formationCenterDescriptor")
convexConcaveString = _gt_activeNode.getParam("convexConcaveString")
useIsCommanderBoolean = _gt_activeNode.getParam("useIsCommanderBoolean")


parameterList = [recursionDescriptor, unitNamesArrayList, wayPoint, arcBetweenLocations, formationOrientation,
      angleToCover, formationCenterDescriptor, convexConcaveString, useIsCommanderBoolean]
printMessage("new security formation param list = "+str(parameterList), True)
```

```
39
40
41 UtilityFuncsExp.addGoalToUnit(
42   unitName,
43   0.1,
44   goalPath,
45   parameterList,
46   None)</Code><Import/></HTNNode></HTNNode></HTNNode></HTNNode>
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX G:
Modifiers to SquadMove Behavior

The following HTNs are used to modify the functionality of the *SquadMove* HTN.

G.1 SquadMove_MoveDesignatedUnits.xml

When this HTN is activated, units that are included in a list all conduct a SquadMove behavior.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="SquadMove_MoveDesignatedUnits" Type="DEFAULT"><Parent>null</Parent><DataMap><DataKey>unitNameToExecuteList,java.util.ArrayList</DataKey><DataKey>start_loc,cxxi.model.objects.features.CMWayPoint</DataKey><DataKey>end_loc,cxxi.model.objects.features.CMWayPoint</DataKey></DataMap><Code IsFile="false"/><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>SquadMove_MoveDesignatedUnits</Parent><Code IsFile="false">if _gt_activeNode.getVar("isInited") == None:
  _gt_activeNode.putVar("isInited", 1)
  _htn_precon_ret=1
</Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="isCommander" Type="DEFAULT"><Parent>initInfo</Parent><Code IsFile="false">if state.isCommander():
  _htn_precon_ret=1
</Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="moveInFormation" Type="GOAL"><Parent>isCommander</Parent><Code IsFile="false">from HTN import UtilityFuncsExp
from java.util import ArrayList


unitNameToExecuteList = _gt_activeNode.getParam("unitNameToExecuteList")
start_loc = _gt_activeNode.getParam("start_loc")
end_loc = _gt_activeNode.getParam("end_loc")

# set the path to the goal
goalPath = "HTN/Trees/SquadMove.xml"

for unitNameToExecute in unitNameToExecuteList:

  # add the goal to a unit
  UtilityFuncsExp.addGoalToUnit(
     unitNameToExecute,
      0.1,
      goalPath,
      [start_loc, end_loc],
      None)


</Code><Import/></HTNNode></HTNNode></HTNNode></HTNNode>
```

G.2 SquadMove_OtherUnitsInArrayList.xml

This HTN contains a list of units that are intended to conduct a SquadMove behavior. If the unit that processes this HTN is included in that list, then all units except the processing unit conduct a SquadMove behavior.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?><HTNNode AllowMsg="true" BreakPoints="" Name="
      SquadMove_OtherUnitsInArrayList" Type="DEFAULT"><Parent>null</Parent><DataMap><DataKey>unitNameToExecuteList,java.
      util.ArrayList</DataKey><DataKey>start_loc,cxxi.model.objects.features.CMWayPoint</DataKey><DataKey>end_loc,cxxi.
      model.objects.features.CMWayPoint</DataKey></DataMap><Code IsFile="false"/><Import/><HTNNode AllowMsg="true"
      BreakPoints="" Name="initInfo" Type="DEFAULT"><Parent>SquadMove_OtherUnitsInArrayList</Parent><Code IsFile="false"
      >if _gt_activeNode.getVar("isInited") == None:
2   _gt_activeNode.putVar("isInited", 1)
3   _htn_precon_ret=1
4 </Code><Import/><HTNNode AllowMsg="true" BreakPoints="" Name="isCommander" Type="DEFAULT"><Parent>initInfo</Parent><Code
       IsFile="false">if state.isCommander():
5   _htn_precon_ret=1
6 </Code><Import/><HTNNode AllowMsg="false" BreakPoints="" Name="moveInFormation" Type="GOAL"><Parent>isCommander</Parent>
      <Code IsFile="false">from HTN import UtilityFuncsExp
7 from java.util import ArrayList
8
9
10 # the purpose of this HTN is to move the units in the unitNameToExecuteList that DO NOT activate this HTN
11 # this is one tool that enables units in the urbanPatrolDemonstration to bound past one another
12
13 unitNameToExecuteList = _gt_activeNode.getParam("unitNameToExecuteList")
14 start_loc = _gt_activeNode.getParam("start_loc")
15 end_loc = _gt_activeNode.getParam("end_loc")
16
17 printMessage("originalUnitNameToExecuteList = "+str(unitNameToExecuteList), True)
18 myUnitName = state.getCurrentUnitName()
19 if unitNameToExecuteList.contains(myUnitName):
20   unitNameToExecuteList.remove(myUnitName)
21 printMessage("updatedUnitNameToExecuteList = "+str(unitNameToExecuteList), True)
22
23 unitNameToExecute = unitNameToExecuteList[0]
24 printMessage("unitNameToExecute = "+str(unitNameToExecute), True)
25
26 # set the path to the goal
27 goalPath = "HTN/Trees/SquadMove.xml"
28
29
30 # add the goal to a unit
31 UtilityFuncsExp.addGoalToUnit(
32     unitNameToExecute,
33     0.1,
34     goalPath,
35     [start_loc, end_loc],
36     None)
37 </Code><Import/></HTNNode></HTNNode></HTNNode></HTNNode>
```

References

[1] A. Ilachinski, *Artificial War: Multiagent-Based Simulation of Combat*. Singapore: World Scientific, 2004, pp. vii – 69.

[2] United States Army. (2014). *COMBATXXI Users Guide*. [Online]. Available: https://cxxi.wsmr.army.mil/confluence/display/CXXIDOC/Home

[3] S. Price, "Low-resolution screening of early stage acquisition simulation scenario development decisions," M.S. thesis, Naval Postgraduate School, Monterey, CA, 2012.

[4] United States Marine Corps, *Systems Approach to Training Manual*. Washington, DC: United States Marine Corps, 2004.

[5] J. Lien *et al.* (2013). Algorithms & applications group: Composable group behaviors. Parasol, Texas A&M University. [Online]. Available: {https://parasol.tamu.edu/dsmft/research/flock/composable}

[6] M. D. Petty and E. W. Weisel, "A composability lexicon," in *Proceedings of the Spring 2003 Simulation Interoperability Workshop*, Kissimmee, FL, Mar./Apr. 2003, pp. 181–187.

[7] J. Summers *et al.*, "Towards applying case-based reasoning to composable behavior modeling," in *Proceedings of Behavior Representation in Modeling and Simulation*, Arlington, VA, May. 2004.

[8] H. Munoz-Avila and T. Fisher, "Strategic planning for unreal tournament bots," in *AAAI Workshop on Challenges in Game AI*, San Jose, CA, Jul. 2004.

[9] R. Pillosu, "Coordinating agents with behaviour trees," presented at the *Paris Game AI Conference*, Paris, France, Jun. 2009.

[10] S. Vassos. (2012). Introduction to AI STRIPS planning and applications to video-games!: Lecture 1, part 2. Sapienza University of Rome. Rome, Italy. [Online]. Available: {http://www.dis.uniroma1.it/~degiacom/didattica/dottorato-stavros-vassos/}

[11] J. Orkin, "Three states and a plan: The AI of F.E.A.R." in *Proceedings of the Game Developer's Conference (GDC)*, San Jose, CA, Mar. 2006.

[12] J. Gibson, *The Ecological Approach to Visual Perception*. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc., 1986, ch. 8, pp. 127–142.

[13] K. Bærentsen and J. Trettvik, "An activity theory approach to affordance," in *Proceedings of the Second Nordic Conference on Human-computer Interaction*, 2002, pp. 51–60.

[14] A. Chemero and M. Turvey, "Gibsonian affordances for roboticists," *Adaptive Behavior*, vol. 15, no. 4, pp. 473–480, 2007.

[15] G. Lintern, "An affordance-based perspective on human-machine interface design," *Ecological Psychology*, vol. 12, no. 1, pp. 65–69, 2000.

[16] E. Rome *et al.*, "The MACS project: an approach to affordance-inspired robot control," in *Towards affordance-based robot control*. Springer, 2008, pp. 173–210.

[17] K. Dill *et al.* (2008, Sep.). *Terrain Analysis: An AiGameDev.com Special Report*. AiGameDev. [Online]. Available: http://aigamedev.com/premium/report/terrain-analysis-reasoning/

[18] L. Liden, "Strategic and tactical reasoning with waypoints," in *AI Game Programming Wisdom*, S. Rabin, Ed. Hingham, MA: Charles River Media, 2002, pp. 211–220.

[19] R. Straatman and A. Beij, "Killzone's AI: dynamic procedural combat tactics," presented at the *Game Developers Conference*, Palo Alto, CA, Mar. 2005.

[20] A. Beij and W. van der Sterren, "Killzone's AI: Dynamic procedural tactics," presented at the *Game Developers Conference*, Palo Alto, CA, Mar. 2005.

[21] T. Verweij, "A hierarchically-layered multiplayer bot system for a first-person shooter," M.S. thesis, Vrije Universiteit of Amsterdam, Amsterdam, Netherlands, 2007.

[22] J. Orkin, "Applying goal-oriented action planning to games," in *AI Game Programming Wisdom 2*, S. Rabin, Ed. Hingham, MA: Charles River Media, 2004, pp. 217–228.

[23] S. Vassos. (2012). Introduction to AI STRIPS planning and applications to video-games!: Lecture 2, part 1. Sapienza University of Rome. Rome, Italy. [Online]. Available: {http://www.dis.uniroma1.it/~degiacom/didattica/dottorato-stavros-vassos/}

[24] N. Nejati *et al.*, "A goal and dependency-directed algorithm for learning hierarchical task networks," in *Proceedings of the Fifth International Conference on Knowledge Capture*, Redondo Beach, CA, Sep. 2009, pp. 113–120.

[25] I. Balogh *et al.*, “Using hierarchical task networks to create dynamic behaviors in combat models,” unpublished.

[26] D. Nau *et al.*, “SHOP2: An HTN planning system,” *Journal of Artificial Intelligence Research (JAIR)*, vol. 20, pp. 379–404, Dec. 2003.

[27] D. Nau and A. Champandard. (2012, Sep.). Inside hierarchical task network (HTN) planners. AiGameDev. [Online]. Available: http://aigamedev.com/premium/interview/htn-planners/

[28] D. Nau, “Game applications of HTN planning with state variables,” presented at the *ICAPS Workshop on Planning in Games*, Rome, Italy, Jun. 2013.

[29] A. Champandard *et al.*, “Killzone 2 multiplayer bots,” presented at the *Game AI Conference*, Paris, France, Jun. 2009.

[30] A. Champandard *et al.*, “On the AI strategy for KILLZONE 2’s bots,” in *Proceedings of the Game Developer’s Conference (GDC)*, San Francisco, CA, Mar. 2010.

[31] W. van der Sterren, “Hierarchical plan-space planning for multi-unit maneuvers,” in *Proceedings of the 3rd International Planning in Games Workshop*, Rome, Italy, Jun. 2013.

[32] W. van der Sterren. (2014, Jul.). PlannedAssault. [Online]. Available: http://www.plannedassault.com/faqs

[33] D. O’Connor. (2011, May.). Strategic War Game AI in Battles from the Bulge. AiGameDev. [Online]. Available: http://aigamedev.com/premium/interview/command-ops/

[34] D. O’Connor. (2007, May.). Creative AI. Panther Games Pty Ltd. [Online]. Available: http://aigamedev.com/open/articles/2008-week-17/

[35] P. Scobell. (2010). Panther Games: Home. Panther Games Pty Ltd. [Online]. Available: http://panthergames.com

[36] W. van der Sterren, “Multi-unit planning with HTN and A*,” presented at the *Paris Game AI Conference*, Paris, France, Jun. 2009.

[37] S. Harmon *et al.*, “Validation of human behavior representations,” in *Foundations for V&V in the 21st Century Workshop*, Laurel, MD, Oct. 2002, pp. 22–24.

[38] D. Pace, “Modeling and simulation verification and validation challenges,” *Johns Hopkins APL Technical Digest*, vol. 25, no. 2, pp. 163–172, 2004.

[39] S. Borkman *et al.*, “Algorithms for ground soldier based simulations and decision support applications. phase 1,” Dignitas Technologies LLC, Orlando, FL, DTIC Tech. Rep. ADA560713, 2012.

[40] E. Clark and J. Smith, “Validation of tactical behavior in command agents using the Tesla composition language,” White Paper, Intelligent Automation, Inc., Rockville, MD, 2012.

[41] United States Marine Corps, *Marine Corps Operations (MCDP 1-0)*. Washington, DC: United States Marine Corps, 2011.

[42] United States Joint Chiefs of Staff, *Department of Defense Dictionary of Military and Associated Terms (Joint Publication 1-02)*. Washington, DC: Director for Joint Force Development (J-7), 2011.

[43] United States Marine Corps, *Warfighting (MCDP 1)*. Washington, DC: United States Marine Corps, 1997.

[44] United States Marine Corps, *Marine Rifle Squad (MCWP 3-11.2)*. Washington, DC: United States Marine Corps, 2011.

[45] United States Marine Corps, *Marine Corps Planning Process (MCWP 5-1)*. Washington, DC: United States Marine Corps, 2010.

[46] United States Marine Corps, *Infantry Training and Readiness Manual*. Washington, DC: United States Marine Corps, 2012.

[47] United States Marine Corps, *R400 Handbook*. Twentynine Palms, CA: Tactical Training and Exercise Control Group, 2013.

[48] R. Noel, “Dynamic decision language,” presented at the *COMBATXXI Advanced Users Workshop*, White Sands Missile Range, NM, Apr. 2008.

[49] M. Mainz, “Operations Order R-400,” 2007, unpublished.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California