View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AFTI Scholar (Air Force Institute of Technology)

Air Force Institute of Technology
AFIT Scholar

Theses and Dissertations

Student Graduate Works

3-26-2015

The Unified Behavior Framework for the Simulation of Autonomous Agents

Daniel M. Roberson

Follow this and additional works at: https://scholar.afit.edu/etd

Part of the Artificial Intelligence and Robotics Commons

Recommended Citation

Roberson, Daniel M., "The Unified Behavior Framework for the Simulation of Autonomous Agents" (2015). *Theses and Dissertations*. 55.
https://scholar.afit.edu/etd/55

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact richard.mansfield@afit.edu.

THE UNIFIED BEHAVIOR FRAMEWORK FOR THE SIMULATION OF AUTONOMOUS AGENTS

THESIS

Daniel M. Roberson, First Lieutenant, USAF

AFIT-ENG-MS-15-M-014

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

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

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-15-M-014

THE UNIFIED BEHAVIOR FRAMEWORK FOR THE SIMULATION OF AUTONOMOUS AGENTS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Daniel M. Roberson, B.S.

First Lieutenant, USAF

March 2015

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

AFIT-ENG-MS-15-M-014

THE UNIFIED BEHAVIOR FRAMEWORK FOR THE SIMULATION OF AUTONOMOUS AGENTS

Daniel M. Roberson, B.S.

First Lieutenant, USAF

Committee Membership:

Douglas D. Hodson, PhD
Chair

Gilbert L. Peterson, PhD
Member

Maj Brian G. Woolley, PhD
Member

AFIT-ENG-MS-15-M-014

Abstract

Since the 1980s, researchers have designed a variety of robot control architectures intending to imbue robots with some degree of autonomy. A recently developed architecture, the Unified Behavior Framework (UBF), implements a variation of the three-layer architecture with a reactive controller to rapidly make behavior decisions. Additionally, the UBF utilizes software design patterns that promote the reuse of code and free designers to dynamically switch between behavior paradigms. This paper explores the application of the UBF to the simulation domain. By employing software engineering principles to implement the UBF architecture within an open-source simulation framework, we have extended the versatility of both. The consolidation of these frameworks assists the designer in efficiently constructing simulations of one or more autonomous agents that exhibit similar behaviors. A typical air-to-air engagement scenario between six UBF agents controlling both friendly and enemy aircraft demonstrates the utility of the UBF architecture as a flexible mechanism for reusing behavior code and rapidly creating autonomous agents in simulation.

Table of Contents

Page

Abstract iv

Table of Contents v

List of Figures vii

List of Acronyms viii

I. Introduction 1

1.1 Problem Statement 1
1.2 Research Goal 2
1.3 Thesis Overview 2

II. Background 3

2.1 Early Inroads in Autonomous Robots 3
2.2 A Behavioral Perspective 5
2.3 Three Layers of Autonomy 7
2.4 The Commercial Gaming Industry's Solution 9
2.5 The Unified Behavior Framework 12
2.6 The Simulation Framework 12
2.7 The Scenario Under Study 15
2.7.1 Ingress 16
2.7.2 Beyond Visual Range (BVR) Engagement 16
2.7.3 Within Visual Range (WVR) Engagement 16
2.7.4 Egress 16

III. IEEE Conference Paper 18

IV. Conclusion 30

4.1 Controller Complexity 30
4.2 UBF versus Behavior Trees 32
4.3 Granularity of Behaviors 32
4.4 Future Work 33
4.4.1 Fusion Arbiter 33
4.4.2 Sequencer 34

Page

4.4.3 Addressing the Complexity 34

Appendix: Implementation Code 36

List of Figures

Figure Page

2.1 A graphical depiction of the sense-plan-act (SPA) architecture as described by Brooks and Gat [1, 2]. 4

2.2 A depiction of two of Braitenberg's "Vehicles" which inspired the behavioral perspective of autonomous robot control [3]. The vehicle on the left (Vehicle a) is exhibiting fear of a light source, while the one on the right (Vehicle b) exhibits love of the same. 6

2.3 A graphical depiction of reactive control architectures as described by Gat [2]. . 7

2.4 A graphical depiction of three layer architectures as described by Gat [2]. . . . 8

2.5 An example behavior tree implementing autonomous driving behavior (with the aid of a GPS). Note that the nodes in the tree will be "ticked" from top to bottom, implying that behaviors higher in the tree have higher priority. 10

2.6 A UML diagram of the Unified Behavior Framework (UBF) [4]. 13

2.7 A graphical depiction of the structure of the OpenEaagles simulation framework. 14

2.8 A birds-eye-view depiction of the sweep mission. 17

List of Acronyms

Acronym	Definition
OpenEaagles	Open Extensible Architecture for the Analysis and Generation of Linked Simulations
UBF	Unified Behavior Framework
SPA	sense-plan-act
WTA	winner takes all
BVR	beyond visual range
WVR	within visual range
IEEE	Institute of Electrical and Electronics Engineers
AI	artificial intelligence
USAF	United States Air Force
CRA	Charles River Analytics
UML	Unified Modeling Language
HOTAS	hands on throttle-and-stick
IFF	Identification Friend or Foe

THE UNIFIED BEHAVIOR FRAMEWORK FOR
THE SIMULATION OF AUTONOMOUS AGENTS

I. Introduction

The development of autonomy has been a goal, if not the primary goal, of the artificial intelligence (AI) research community since its inception. Robots are possibly the purest demonstration of true autonomous agents, as they must navigate the dynamic environment that is the real world using simple (by comparison to human capabilities) sensors and actuators. While robots are indeed a valuable metric for evaluating our progress towards the goal of complete autonomy, there is room for research by utilizing methods like simulation and modeling that consume less time and fewer monetary resources. A recently developed reactive control framework, known as the Unified Behavior Framework, has proven to be an effective method for autonomous robot agents. By implementing this framework within an open-source simulation framework, known as OpenEaagles, autonomous agents could be more easily designed, developed, tested and understood.

1.1 Problem Statement

Autonomy in and of itself is a difficult problem facing the AI research community. Instead of focusing on developing costly and complicated robotic platforms, effort could be focused on tackling these problems in simulation before investing in the hardware needed for robot development. The value of simulation for autonomy research is indispensable; however, it would be futile if every different architecture required it's own made-from-scratch code base. The effort involved in building an autonomous agent architecture from the ground up quickly expands beyond reasonable, especially when such a large number

of architectures is available for use. Software engineering principles can enable dynamic switching between robot architectures, and can ensure rapid design and development of autonomous agents with radically different characteristics and capabilities.

1.2 Research Goal

This research attempts to attain a degree of autonomy in AI agents inserted into a military context. The use of simulation to conserve resources is critical to furthering the development of robot control architectures. Finally, the application of modern software engineering principles is necessary to promote code reuse and ensure design flexibility.

1.3 Thesis Overview

This thesis presents a brief introduction in Chapter 1, outlining the problem and goal of this research. In Chapter 2, we explore the history of autonomous robot architectures, covering early research all the way to current methods for implementing autonomy. Chapter 3 includes a short paper that presents the bulk of this research in a condensed format, ready for publishing. Finally, Chapter 4 discusses the results of this research and the potential for future work in this research area.

II. Background

Autonomy is likely the biggest thrust of the AI community's research since its inception. The pursuit of an agent capable of thinking, acting, solving problems, and behaving independent of human input is a main driver in this research. Many approaches to this problem have been tried, not the least of which being the ambitious goal of constructing a robot capable of acting alone to complete a task. Robots, in some sense, represent the ultimate goal of autonomy, being more than just a computer program that can perform tasks independently, but an actual physical entity that is capable of living and acting in the real world. Seeing this goal of robotic autonomy come to fruition is likely years in the future; however, the brief history of robotics has already shown rapid and unimaginable progress.

2.1 Early Inroads in Autonomous Robots

The early 1980s began the AI research community's headfirst dive into the problem of autonomous robots. Initially, it was natural to break autonomy down into a series of chronological steps. By trying to understand how humans solve problems, and applying that to the robot control program, we could potentially imbue the robot with human-like thought patterns. Of course, it is never that simple. The chronological understanding of a human's interaction with the world is deceivingly simple. First, we receive data about the world through our sensory organs, and we interpret this data and form our understanding of our surrounding environment. Then, we can start to make decisions based on that "world model," planning and choosing how we think we should act to maneuver through our environment towards our goal. Finally, we can actually use our muscles and our motor organs to propel ourselves through the environment, proceeding towards our goal.

Clearly, this is a massively oversimplified version of how we humans actually operate. Simple sensory processes are almost beyond comprehension, when one considers the

complexity of our most used anatomical "sensors," our eyes. We know, based on optical physics, that the retina in the back of the human eye actually receives an inverted (upside-down) image of the world around us. In the extremely short amount of time it takes for that image to be interpreted and understood by us, our brains have managed to revert the image to its true orientation. Not only that, but our brain has also merged the two separate images from each of our eyes to form the visual scene that allows us to see the world around us [5].

In light of how truly complicated these tasks really are - tasks we perform without thinking - it follows that implementing even basic functionality on robots is exceedingly difficult. The initial approach, a chronological understanding of problem-solving, proved to be effectively useless in getting robots to perform autonomously. The aptly and humorously-named robot, "Shakey," demonstrates the early issues with the chronological view of autonomy [6]. Shakey implemented this approach, later to be called the sense-plan-act (SPA) architecture, and the problems with it were quickly revealed by Shakey's, for lack of a better term, shakiness when navigating the real world [2]. The SPA architecture is depicted in Figure 2.1.

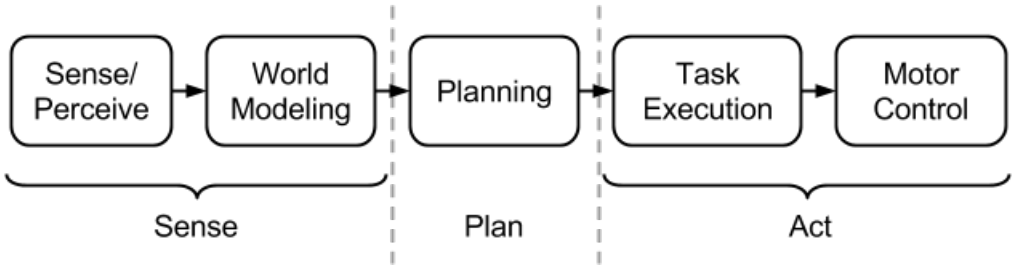

Figure 2.1: A graphical depiction of the SPA architecture as described by Brooks and Gat [1, 2].

A solution was needed, or perhaps a new way of thinking about the problem. It was quickly obvious that in the dynamic environment that is the world, a robot does not have much time to spend reevaluating sensor data and reconstructing a world model. While it builds a model, the world around it is changing faster than it can cope.

2.2 A Behavioral Perspective

In a psychology thought experiment he called "Vehicles", Braitenberg challenged the AI community's sense-plan-act understanding of autonomy [3]. While trying to demonstrate how seemingly complex behaviors could be caused by simple internal mechanisms, Braitenberg inadvertently presented a solution to the autonomy problem. Instead of breaking the problem down sequentially, it was possible that a behavioral understanding was more appropriate. Figure 2.2 illustrates two of Braitenberg's simple robots that exhibit what we might refer to as fear and love, respectively, of a light source.

The first to understand and implement a behavioral solution was Brooks, who gave us the subsumption architecture. By layering behaviors, Brooks was able to create a robot that responded instinctively to certain conditions. These low-level behaviors, or instincts, were achieved by tightly coupling the robot's sensors to the robot's actuators, as Braitenberg described. Thereby, obstacle avoidance, for instance, did not require for a complete world model to be built; instead, a simple behavior was developed that stopped the robot's forward motion when an obstacle was detected nearby. This simple behavior was decidedly quicker than robots built on SPA architectures. Brooks built more complex behaviors on top of the lower-level "instincts," so that the robot could explore by wandering throughout its environment. However, the lowest-level behaviors ran regardless of higher-level behaviors. In this way, the higher-level planning and decision-making behaviors, which required a fair amount of time to execute, could do so without fear of the robot crashing into a wall while they produced a plan.

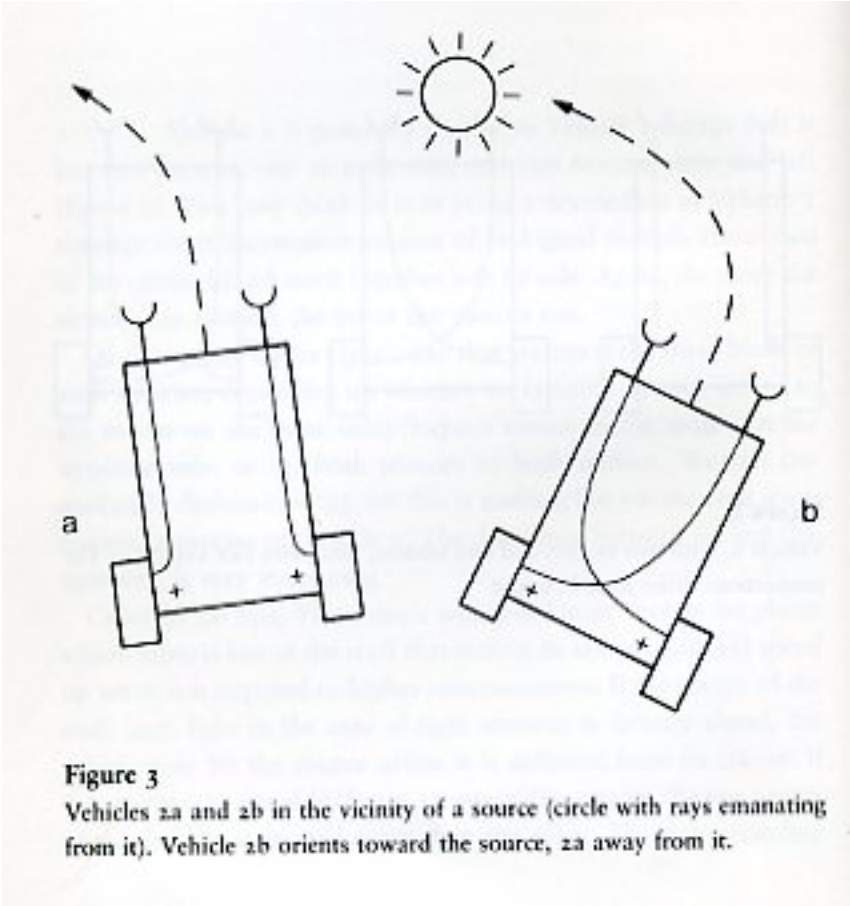

Figure 2.2: A depiction of two of Braitenberg's "Vehicles" which inspired the behavioral perspective of autonomous robot control [3]. The vehicle on the left (Vehicle a) is exhibiting fear of a light source, while the one on the right (Vehicle b) exhibits love of the same.

This concept, tightly coupling sensors to actuators, came to be known as "reactive control," or "reactive planning" [2]. This concept is illustrated in Figure 2.3.

Though subsumption is now well-recognized as the first approach that used a reactive control behavioral architecture, it was not without its own issues. Gat describes these complications with subsumption; namely, its difficulty with managing complexity due to insufficient modularity, but additionally, it was an inaccurate representation of human

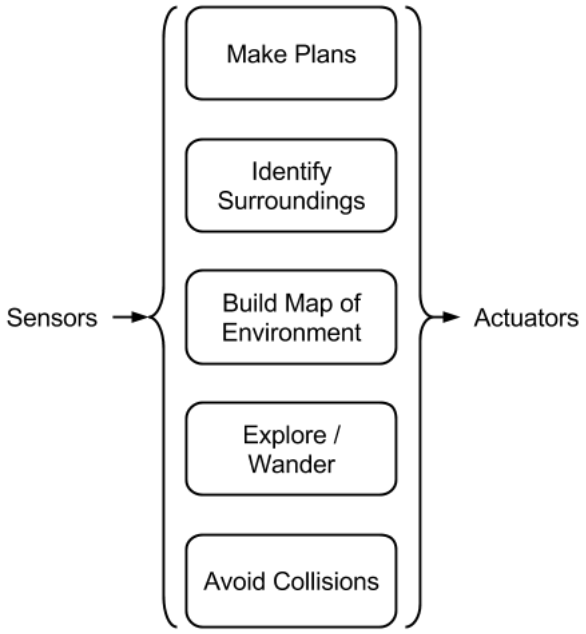

Figure 2.3: A graphical depiction of reactive control architectures as described by Gat [2].

intelligence. Multiple architectures emerged to combat these problems, and many of them followed a structure that Gat described as the three layer architecture [2].

2.3 Three Layers of Autonomy

The three layer architecture, as expected, is constructed with three layers of execution. Each layer contains different elements pertaining to the robot's behavior. Figure 2.4 depicts the arrangement of the three layer architecture.

The lowest layer, the controller, composes the reactive control element of tightly coupled sensors and actuators. Therefore, the controller usually retains a very limited amount of state, and sometimes none. Gat recommends that if internal state is used, "it should expire after some constant-bounded time" [2].

Above the controller layer is the sequencer. The sequencer is able to retain some state, and therefore addresses the inability of basic reactive control architectures to switch

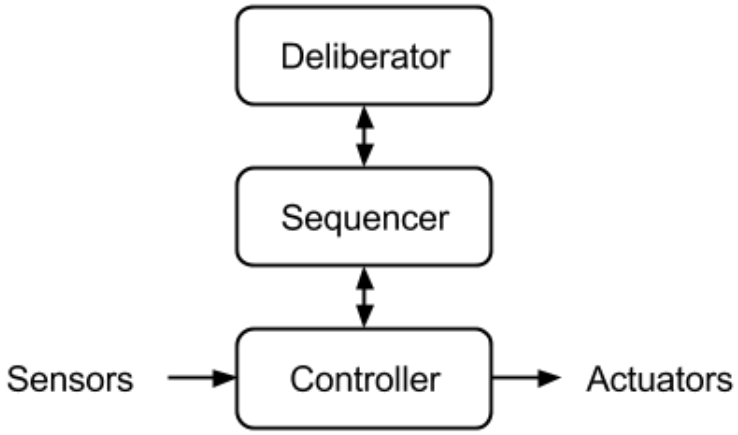

Figure 2.4: A graphical depiction of three layer architectures as described by Gat [2].

between goals or tasks. By recognizing characteristics of the state, the sequencer may be able to switch the behavior of the controller based on the goal being pursued or task being performed. However, because the sequencer is separate from the controller, it does not interfere with low level behaviors' execution until it switches the current behavior scheme. For this reason, the three layer architecture avoids the bottlenecking problems of the SPA architecture.

At the highest layer, the deliberator performs time-intensive, high-latency tasks for the robot. Usually, this involves planning activities, but could also involve expensive algorithms like vision-processing or other polynomial-time algorithms. Running on separate threads, the deliberator, again, does not interfere with the operation of the sequencer or controller. Depending on the architecture, however, Gat notes that the deliberator can either be authoritative or subject to the sequencer. In some architectures, the deliberator produces plans and sends them to the sequencer for execution. On the other hand, some deliberators receive requests from the sequencer, executing its threads only when asked [2].

2.4 The Commercial Gaming Industry's Solution

Splitting up the different execution elements of a robot's control architecture into three layers solved many of the problems of the earlier SPA and reactive control architectures. However, some of these execution issues differ from the autonomy problem in commercial games. Whereas a robot has to worry about latency in order to rapidly interpret information from the sensors and apply outputs to the actuators, commercial games' non-player characters (NPCs) have direct access to the world model, without needing to interpret sensor data. Also, NPCs don't need actuators or motors, and therefore short scripts will effect actions onto the NPCs in the game. Without this bottleneck, which the robot community solved with reactive control, the commercial gaming industry was able to innovate a different solution to the autonomy problem.

Isla, in 2005, introduced one of the gaming industry's recent methods for implementing autonomous NPCs, which he called "behavior trees." He restates the autonomy problem succinctly: "A 'common sense' AI is a long-standing goal for much of the research AI community" [7]. Though some of the intricacies of autonomy are different in commercial games, clearly, the problem is the same.

Behavior trees are a decision-making structure that allow NPCs to execute a wide variety of behaviors simply and quickly. Isla also recognizes that "we are concerned...with how easy it is for the users of the AI system - the level designers - to make use of the system to put together a dramatic and fun experience for the player" [7]. The ease with which behavior trees can be understood allows for designers to quickly build and populate complicated behavior trees that respond realistically and rapidly within the game environment.

The structure of behavior trees technically is a hierarchal finite state machine implemented as a directed acyclic graph. In this way, though nodes in the tree may implement the same behaviors, they must not create cycles in the tree's traversal. Behavior

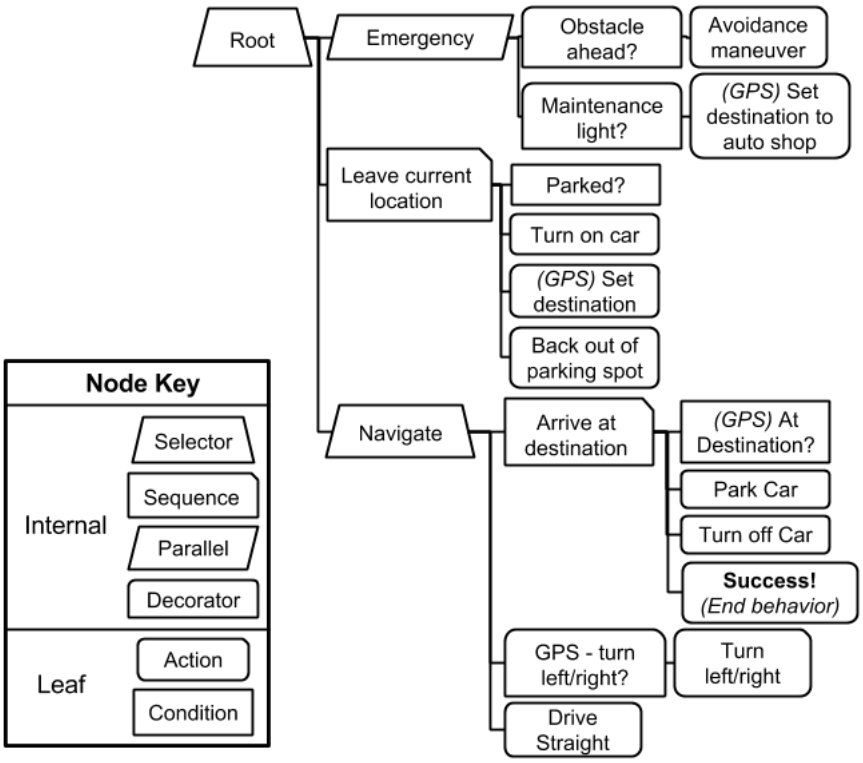

Figure 2.5: An example behavior tree implementing autonomous driving behavior (with the aid of a GPS). Note that the nodes in the tree will be "ticked" from top to bottom, implying that behaviors higher in the tree have higher priority.

trees are traversed depth-first, and each time a traversal is initiated, it is termed as "ticking" the tree [8]. As the tree is traversed, the leaf or external nodes return values indicating success or failure. Or in the case of a script being run to complete an action, they could return a running condition.

These external nodes can either be action or condition node types. The action node type is the only element of the tree structure where scripts are run that affect the NPC's activity. Therefore, action nodes return success or failure when an action has completed, but while a behavior script is in progress, the action node will return running.

Condition nodes are used within the tree structure to direct the traversal of the tree. They typically contain some sort of test condition, and depending on how that condition evaluates, they will return success or failure. Unlike action nodes, condition nodes cannot return running.

To facilitate the decision-making within the tree, internal nodes (with at least one child node) may be one of several types. Marzinotto defines them as sequence, selector, parallel, or decorator nodes. These node types help the designer implement the pattern of traversal through the tree.

Sequence nodes are used to tick a sequence of children nodes in order, sometimes looping over the same sequence repetitiously. When a child of a sequence node returns failure or running, the sequence node stops ticking and returns failure or running, respectively. As long as the children nodes return success, however, the sequence node will continue to tick its children in order.

Selector nodes tick their children in order, like sequence nodes, however they are waiting for one child to return success or running, rather than failure. In this way, they select one of the children. The order of the selector node's children can be considered the order of priority (high to low), as once a node is selected, none of the later children will be ticked.

Parallel nodes tick every one of their children, regardless of return condition. However, as the children nodes return, the parallel node records each of the return values. After all children have been ticked, parallel nodes use the return value information to decide whether to return success, running, or failure, typically by evaluating certain thresholds. For instance, if a parallel node had five children, it might return success if at least 3 returned success, it might return failure if at least 3 returned failure, but it would return running otherwise.

Decorator nodes are allowed only one child node. The decorator node records internal variables, which are evaluated every time the decorator node is ticked. If the internal conditions over those variables are met, the decorator node ticks its child node and returns based on an internal function.

Figure 2.5 shows an example behavior tree, utilizing all different types of nodes mentioned above, and implements autonomous car-driving behavior.

Clearly, the ease with which behavior trees can be understood facilitates a quick and seamless design process, especially in commercial games where sensors and actuators do not create latency in complex behaviors. Therefore, behavior trees are a valuable construct to understand, especially when comparing them to the autonomous robot architectures mentioned in sections 2.1- 2.3.

2.5 The Unified Behavior Framework

Meanwhile, in the autonomous robot research domain, Woolley created a framework which centered around the reactive control concept, pairing the speed of reactive control with the modularity and extensibility of the composite and strategy design patterns. The incorporation of these software engineering principles promoted code reuse and enabled the rapid design of behaviors. The framework was coined Unified Behavior Framework (UBF), highlighting its ability to dynamically switch between behavioral architectures, thereby unifying them under one framework [4]. A Unified Modeling Language (UML) diagram of the UBF is shown in Figure 2.6.

The paper presented in chapter 3 discusses both the history and the implementation of UBF as a part of this research.

2.6 The Simulation Framework

As this research is hinged on demonstrating the UBF's potential for simulation applications, a simulation framework was necessary. A worthy framework was found in

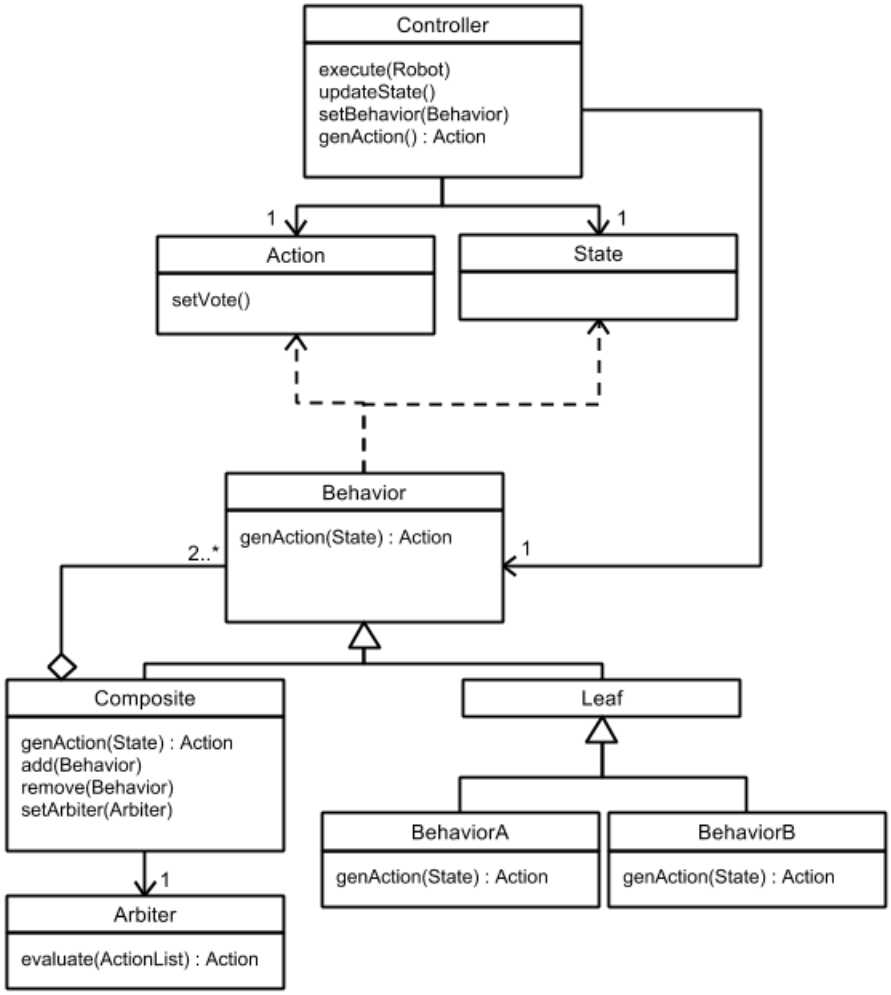

Figure 2.6: A UML diagram of the Unified Behavior Framework (UBF) [4].

the open-source domain, referred to as Open Extensible Architecture for the Analysis and Generation of Linked Simulations (OpenEaagles). This framework "is a simulation design pattern that provides a structure for constructing simulation applications." By following modern software engineering principles, OpenEaagles "aids the design of robust, scalable, virtual, constructive, stand-alone, and distributed simulation applications" [9]. Because OpenEaagles is oriented towards aerial scenario simulation (it originated in the Simulation and Analysis Facility for the United States Air Force (USAF)), it was useful to give military

context to our implementation. Figure 2.7 illustrates the architecture of a simulation application built using OpenEaagles.

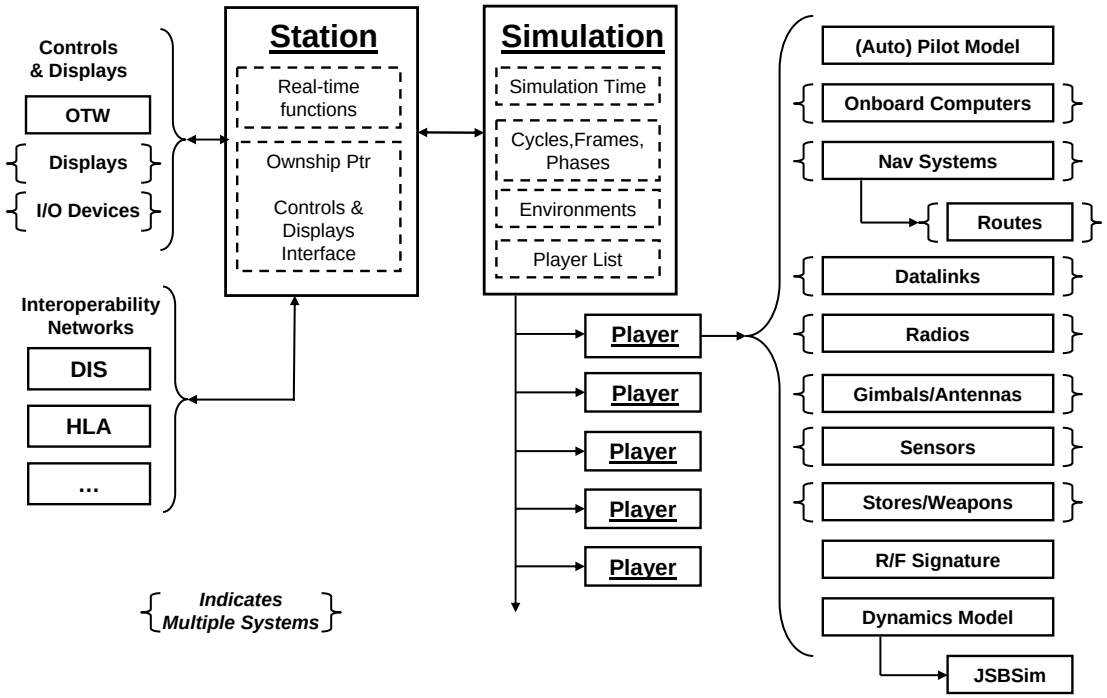

Figure 2.7: A graphical depiction of the structure of the OpenEaagles simulation framework.

The simulation framework is written in C++ due to the performance gains afforded by C, along with the object-oriented nature of C++. To take advantage of the performance of C++, the framework utilizes common memory management techniques within its own object system. In addition, OpenEaagles, is split into two threads of execution, a foreground

and background thread, which allows time-sensitive computations to be performed quickly, while less important computations can run in the background when processor space is available.

Real-time simulations are built to accurately simulate the passing of time. In order to accomplish this, OpenEaagles implements a frame-based approach, where each entity within the simulation updates based on how much real-world time has passed. With this in mind, every entity in the simulation has a method that re-calculates the state of that entity, and is called every time the simulation frame advances.

As OpenEaagles is a well-developed, stable, and open source simulation framework, it is certainly suitable for the implementation and of UBF and the development of our scenario application.

2.7 The Scenario Under Study

As was discussed, a scenario is critical to determining a correct implementation of UBF within OpenEaagles. Additionally, our scenario can provide some military context, ensuring that our implementation is appropriate for the combat domain.

A simple scenario that proved useful for evaluating autonomous agents was developed in a report completed by Charles River Analytics (CRA) in 1999. This scenario, known as the sweep mission, was simple enough to implement, yet complex enough that it adequately tested the capabilities of UBF and OpenEaagles [10].

Though the sweep mission is discussed in depth by the CRA, some adjustments were made to ensure that the UBF implementation stayed the focus of the research. The version of the sweep mission utilized for this project includes four phases: ingress, beyond visual range (BVR) engagement, within visual range (WVR) engagement, and egress. An overview of the mission is depicted in Figure 2.8.

2.7.1 Ingress.

The ingress phase of the mission consists of navigating along a set of waypoints to the designated mission area. During this phase, friendly aircraft will be scanning visually and on their radar for enemy aircraft, or "bogeys." If enemy aircraft are discovered, the friendlies will proceed to the engagement phase.

2.7.2 Beyond Visual Range (BVR) Engagement.

BVR engagement occurs if the friendly flight notices enemy aircraft outside of their visual range, typically by radar scans picking up a track of those enemy aircraft. If the friendly aircraft have the capability and a cache of long-range weapons, they may engage the enemy aircraft by releasing those weapons. If the enemy aircraft are not destroyed by the time the friendlies have closed the distance between them, then the friendly flight will proceed to the WVR portion of the engagement phase.

2.7.3 Within Visual Range (WVR) Engagement.

Once the enemy aircraft are visible to the friendly flight of aircraft, the WVR engagement phase begins. This portion of the engagement is known colloquially as "dogfighting," where the friendly and enemy aircraft perform complex maneuvering for advantage, engaging each other with short-range weapons. Upon destruction of the enemy aircraft, or if an emergency condition occurs, the friendlies will progress to the egress phase of the mission.

2.7.4 Egress.

Egress occurs when the friendly aircraft are leaving the mission area. Sometimes this is due to mission success, where all bogeys have been destroyed. Otherwise, if a friendly aircraft is low on fuel, or if too many friendly aircraft have been damaged or destroyed, the egress phase may be entered. Ultimately, the friendly aircraft return to the original waypoint, usually their home airfield, during the egress phase. This may be via a direct path to the home airfield, or it may, like ingress, occur by navigating along a set of waypoints.

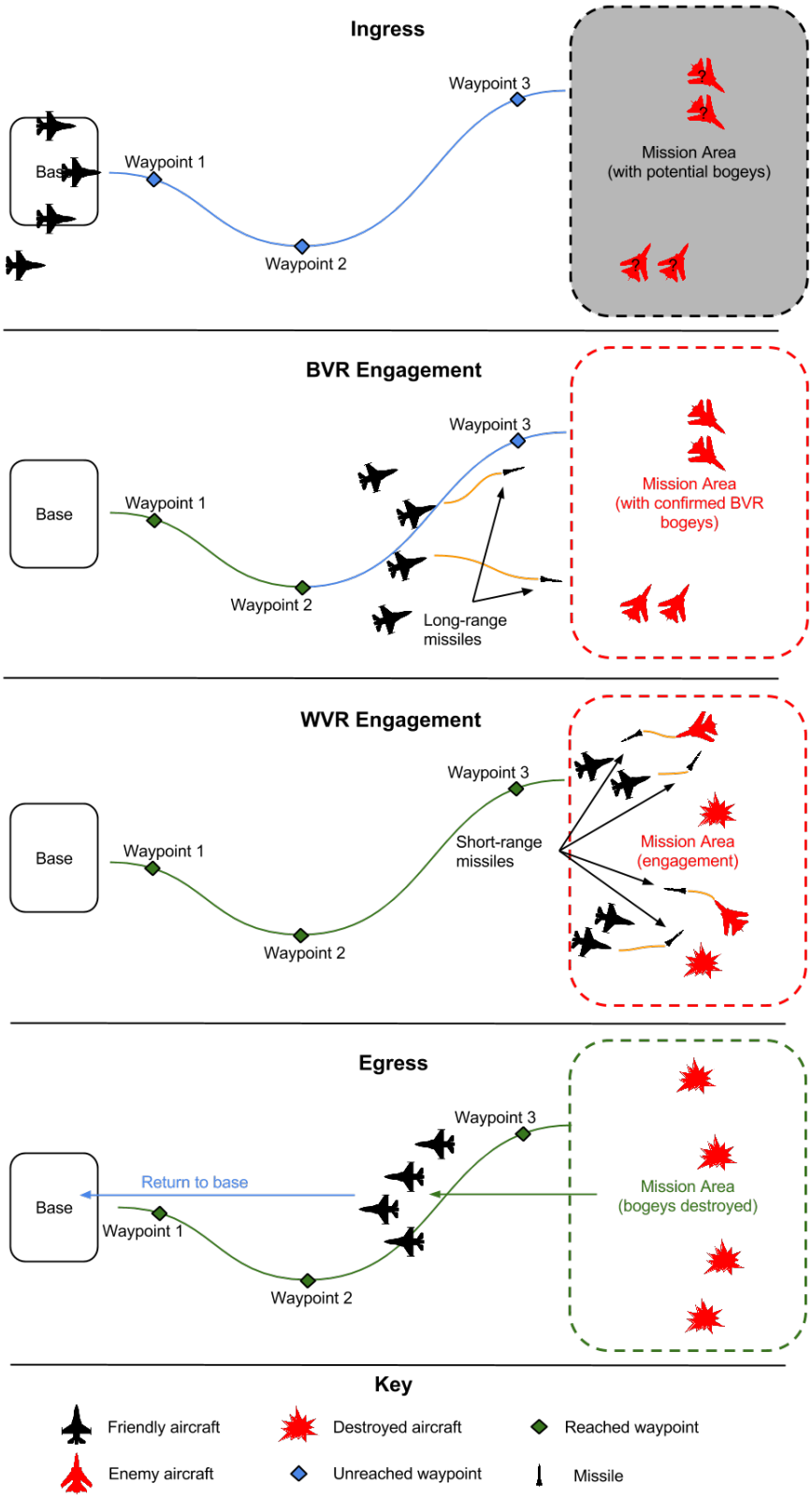

Figure 2.8: A birds-eye-view depiction of the sweep mission.

III. IEEE Conference Paper

Included as a condensed and complete overview of this thesis' research, is the following conference paper in IEEE format.

The Unified Behavior Framework for the Simulation of Autonomous Agents

Daniel Roberson*, Douglas Hodson†, Gilbert Peterson‡, and Brian Woolley§
Department of Electrical and Computer Engineering
Air Force Institute of Technology
Wright-Patterson Air Force Base, Ohio 45433
Email: *daniel.roberson@afit.edu, †douglas.hodson@afit.edu, ‡gilbert.peterson@afit.edu, §brian.woolley@afit.edu
Phone: *443-504-9177, †937-255-3636 x4719 , ‡937-255-3636 x4281 , §937-255-3636 x4618

***Abstract*—Since the 1980s, researchers have designed a variety of robot control architectures intending to imbue robots with some degree of autonomy. A recently developed architecture, the Unified Behavior Framework (UBF), implements a variation of the three-layer architecture with a reactive controller to rapidly make behavior decisions. Additionally, the UBF utilizes software design patterns that promote the reuse of code and free designers to dynamically switch between behavior paradigms. This paper explores the application of the UBF to the simulation domain. By employing software engineering principles to implement the UBF architecture within an open-source simulation framework, we have extended the versatility of both. The consolidation of these frameworks assists the designer in efficiently constructing simulations of one or more autonomous agents that exhibit similar behaviors. A typical air-to-air engagement scenario between six UBF agents controlling both friendly and enemy aircraft demonstrates the utility of the UBF architecture as a flexible mechanism for reusing behavior code and rapidly creating autonomous agents in simulation.**

I. Introduction

The pursuit of autonomous agents is one of the main thrusts of the artificial intelligence research community. This has manifested in the robotics community, where development has progressed towards the creation of robots that can autonomously pursue goals in the real world. Building robots to explore autonomy is practical, but it requires investment of time and resources beyond the design and development of the software. On the other hand, simulation is an effective and inexpensive way of exploring autonomy that does not require the hardware, integration effort, and risk of damage inherent in designing, constructing, and testing robots. Not only that, but robots can be simulated in a variety of environments that push the limits of their autonomous capability. The ability to stress and analyze a robot might otherwise be impractical in a real-world context. So, it seems that simulation is a good option for researching and testing applications of autonomous robots. However, there is a plethora of robot control architectures available, and simulating each of them individually would require a huge code base. With the application of software engineering principles, it is possible to reduce this coding requirement. Doing so grants the designer access to a wide range of autonomous architectures within a single, flexible framework.

The Unified Behavior Framework (UBF) applies such software engineering principles by implementing well-established design patterns and an extensible behavior paradigm. Because of it's flexibility, UBF can be used to explore multiple robot control architectures simultaneously. Currently, UBF has been implemented mainly on robot platforms [1]. However, due to its adaptability, it is ripe for implementation on other AI platforms. In this paper, we discuss a basic implementation and demonstration of UBF within a simulation environment. OpenEaagles (Open Extensible Architecture for the Analysis and Generation of Linked Simulations) is an open-source framework that simplifies the development of a simulation application or scenario. Again, by utilizing software engineering principles, OpenEaagles lends itself to the rapid creation of scenarios, and therefore is a copacetic simulation framework in which to implement the UBF. Additionally, a simple example of an air engagement scenario was developed in order to demonstrate the utility of the implementation. This scenario, known generically as the sweep mission, verifies the ability for rapid scenario development with UBF-based agents, and it demonstrates its application in a military context.

In this paper, we will first discuss some relevant background, highlighting some history in the robotics field, previous applications of UBF, the OpenEaagles framework, and a breakdown of the sweep mission scenario. Then we will delve into this implementation, examining the UBF structure within OpenEaagles, and the specific implementation of the sweep mission within the our UBF implementation. We will discuss the results of our implementation of the sweep mission, and end with a conclusion and a brief look at possible future work.

II. Background

A. Robot Control Architectures

1) Early Robot Architectures: Perhaps the purest demonstration of an artificial intelligence (AI) agent is an autonomous robot. As Braitenberg discusses, the external appearance of autonomy can be attained through the simplest of internal behavior mechanisms [2]. In pursuit of such an autonomous agent, the challenge of creating and implementing autonomous behavior in robots has evolved greatly since it's beginnings in the mid-1980s. "Shakey the Robot" demonstrates the early approach to robot control, which Gat describes

as the Sense-Plan-Act (SPA) architecture [3], [4]. As Figure 1 indicates, the problem of autonomy was typically decomposed into a "series...of functional units" namely the sensing, planning, and acting units (which may be further decomposed depending on the specifics of an implementation) [5]. While SPA is indeed the most chronological way of breaking down the autonomy problem, the approach was plagued by problems due to the variability inherent in the real world. Because planning and acting cannot occur prior to the construction of a world model, those latter stages depend on the correctness of the world model constructed in the sensing stage. However, the sensing stage tends to require the longest amount of computational time, and the dynamic nature of the real world can quickly render the robot's internal world model obsolete [6].

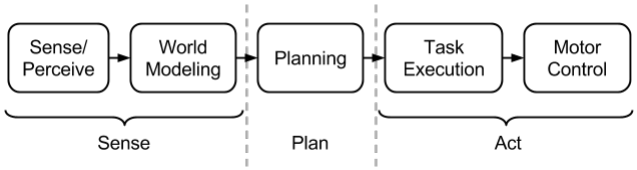

Fig. 1. A graphical depiction of the sense-plan-act (SPA) architecture as described by Brooks and Gat [5], [4].

2) The Emergence of Subsumption: The shortcomings of the SPA architecture were first addressed by Brooks, who dealt with the robot autonomy problem with the subsumption architecture [5]. Instead of breaking the problem down into the chronological components of sensing, then planning, then acting, he broke it down by tasks, or behaviors. These behaviors are "layered," so that the more complex behaviors are built on top of the existing simpler ones, and the complex behaviors have the ability to override the simple behaviors' outputs. Since the behavior layers execute simultaneously, the simpler behaviors will continue to produce motor outputs, but more complex behaviors are able to suppress those outputs when deemed unnecessary or when more complex behaviors are desired. In this way, the robot will function when only simple behaviors are implemented, but the robot can gain "levels of competence" as more complex behaviors are layered over the already-functioning lower levels [5].

3) Reactive Control: Brooks brought the subsumption architecture into the forefront of robot control architecture design, highlighting the usefulness of a tight coupling between a robot's sensors and actuators. A graphical representation of this coupling is illustrated by Figure 2. Deemed "reactive planning" or "reactive control" by Brooks' contemporaries, this approach recognizes the bottlenecks inherent in the SPA architecture, primarily in the sensing stage where the world model is constructed [4]. As reactive architectures expanded upon Brooks' architecture, some of the issues with subsumption were revealed. Gat describes some of these issues; namely, the lack of modularity, no mechanism for managing complexity, the inability of low-level behaviors to affect high-level behaviors, and the inaccuracy of subsumption's representation

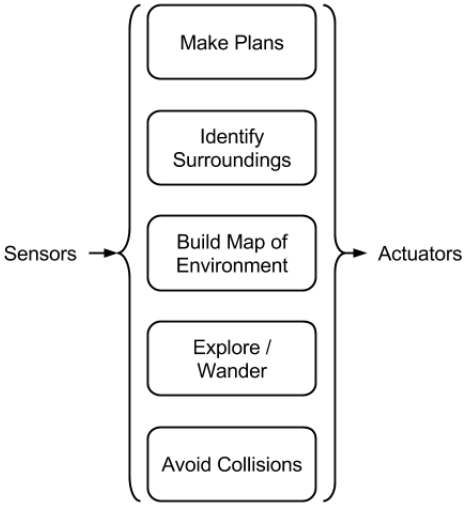

Fig. 2. A graphical depiction of reactive control architectures as described by Gat [4].

of human intelligence [4]. As reactive control architectures addressed subsumption's shortcomings in the years following Brook's initial concept, robots began to act more intelligently in dynamic environments and even began to accomplish simple tasks involving interaction with the world. Gat highlights two of these robots, Tooth and Rocky III, which were able to search for and retrieve small objects in the surrounding environment. However, these initial foray's into reactive control architectures revealed their own shortcomings. While a reactive control architecture's tight coupling between sensors and actuators leads to quick and effective action in a dynamic environment, robots were dedicated to single tasks, and needed complete reprogramming to change their main goal [4].

4) Three Layer Architectures: This main shortcoming was addressed by expanding robot architectures to include three layers of execution. The reactive control layer came to be known as the controller, and two higher layers were added, the sequencer above the controller, and the deliberator above that. Figure 3 depicts the typical structure of a three layer architecture. The controller layer, as before, maintained a tight coupling between sensors and actuators, so that it could react quickly to a changing environment. As mentioned, however, this reactive control layer did not retain state by building a world model, it only implements and executes a predefined and preprogrammed behavior, much like Braitenberg's "Vehicles" [2]. Therefore, incorrect or inaccurate sensor data might manifest in incorrect or unexpected behavior. With the addition of the sequencer, multiple tasks could be performed by the robot depending on the situation. The sequencer could detect the current situation by maintaining some internal state. With these changes, the sequencer was able to switch the current behavior/task based on the current environment. Finally, the third and highest layer, the deliberator, is where the time-consuming planning aspects of robot intelligence reside. Either by building plans and sending them to the sequencer for

execution, or by receiving requests from the sequencer, the deliberator runs as it's own thread producing high-level plans or running time-consuming operations separately from the speedier sequencer and controller threads [4], [7].

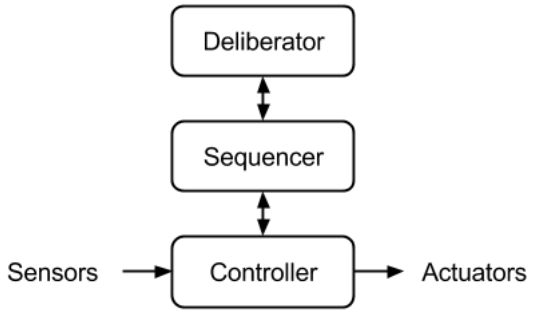

Fig. 3. A graphical depiction of three layer architectures as described by Gat [4].

B. Behavior trees

While the robotics community has progressed from SPA, through subsumption, all the way to three layer architectures for controlling their robotic agents, the commercial gaming industry has faced similar problems when trying to create realistic non-player characters (NPCs). Like robots, these NPCs are expected to be autonomous, acting with realistic, human-like intelligence within the game environment. As Isla states, "a 'common sense' AI is a long-standing goal for much of the research AI community." In pursuit of this goal, Isla introduced an AI concept, colloquially referred to as "behavior trees," which was first implemented in the popular console game Halo 2. More technically, behavior trees are hierarchical finite state machines (HFSMs) implemented as directed acyclic graphs (DAGs) [8], [9].

In the same way that recent robot architectures focus on individual tasks, or behaviors, an agent's behavior tree executes relatively short behavior scripts directly onto the NPC, so that it exhibits the specified behavior. These scripts are built into a tree structure that is traversed depth-first node-by-node. The tree is queried, or "ticked" at a certain frequency, and behaviors are executed (or not) based on the structure of the tree and the types of nodes that are being ticked. In order to facilitate decision-making, the tree contains multiple types of nodes. As Marzinotto defines them, these node types are either specified as internal or external (leaf) nodes. The internal node types are selector, sequence, parallel, and decorator, while the external/leaf nodes are either actions or conditions. In addition, after being ticked all nodes will either return a success, failure, or running condition, indicating whether the behavior was successful, or if it is still running [9]. Figure 4 provides a simple example of a behavior tree that utilizes at least one of each type of node and implements autonomous vehicle-driving behavior.

At the leaf level, action nodes are the only nodes that actually implement control steps upon the agent. When an

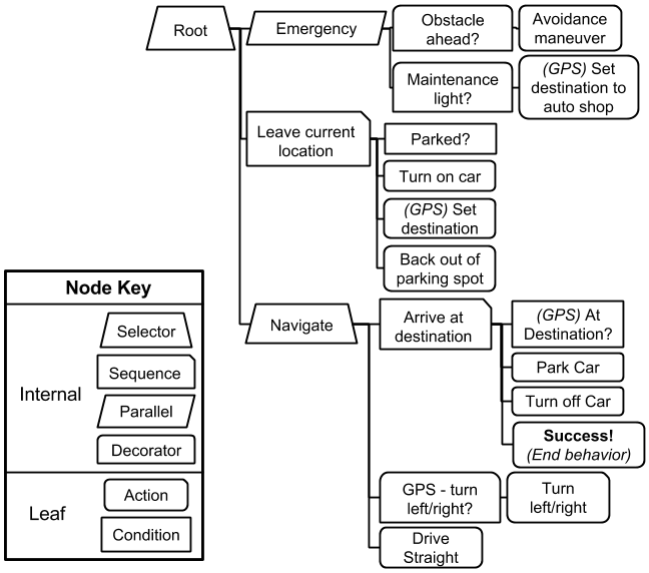

Fig. 4. An example behavior tree implementing autonomous driving behavior (with the aid of a GPS). Note that the nodes in the tree will be "ticked" from top to bottom, implying that behaviors higher in the tree have higher priority.

action node is ticked, it will execute the control step and return running until the control step is complete. Once completed, success or failure return values indicate whether the control step achieved the desired state.

Condition nodes, like action nodes, evaluate the agent's state and return either success or failure, however, they cannot exercise control over the agent, and therefore cannot return running.

Internally, selector, sequence, parallel, and decorator nodes represent different elements of the agent's decision-making process. Selector nodes select one child by ticking each child in order until one of the children returns running or success, which the selector node also returns. If all children return failure, the selector node fails.

Sequence nodes execute each child in order, by ticking each until one of the children returns running or failure. If none of the sequence node's children fails, it will return success, otherwise, it will return running or failure based on the running/failed child's return condition.

Parallel nodes tick all children regardless of return condition, ticking each child node in sequence. The parallel node maintains a count of the return values of every child. If either the success or return value counts are greater than established thresholds, the parallel node will return the respective success or failure condition. If neither threshold is met, the parallel node will return running.

Finally, decorator nodes have internal variables and conditions that are evaluated when ticked, and are only allowed one child node. If the conditions based on the internal variables of the decorator node are met, the child node is also ticked. The return value of a decorator node is based on a function as applied to the node's internal variables.

Due to their ease of understanding and the ability to quickly

construct large trees, behavior trees are extremely effective for building AI agents in commercial games. As Marzinotto demonstrates, with slight modifications, behavior trees can also be effectively applied to robot control architectures [9]. There are a few limitations when it comes to robot control, however. First is the necessity for the behavior tree action nodes to have direct control over the robot's actuators. This is less of a problem, as the running return value of nodes accounts for the time it takes for a node to complete the relevant behavior. However, in addition to requiring direct control over the actuators, the entire behavior tree also needs access to the current world state. In commercial games, these are not issues, as the NPCs can be given complete and 100% accurate information about the virtual world at any time through the code, with no sensors or world model-building required. In the robot control domain, however, the state of the robot must be gathered from the sensors and built into some sort of world model. This world model is sometimes inaccurate, and as the world changes, it can quickly become obsolete, as discussed in section II-A1. Marzinotto admits that "a large number of checks has to be performed over the state spaces of the Actions in the [behavior] tree," acknowledging this shortfall of behavior trees for robot control. In his case, Marzinotto works around this problem of behavior trees by being willing to accept a delayed state update rather than interrupt the ticking over the behavior tree [9]. Also, behavior trees lack the flexibility of behavior-switching and goal-setting provided by sequencer and deliberator (respectively) of the three layer architecture.

C. Unified Behavior Framework

In response to the issues of behavior trees for robot control, the Unified Behavior Framework (UBF) decouples the behavior tree from the state and actions. By reintroducing the controller, the UBF enforces a tight coupling between sensors and actuators, ensuring the rapid response times of reactive control architectures. UBF also utilizes the strategy and the composite design patterns to guarantee design flexibility and versatility over multiple behavior paradigms [10]. In this way, UBF reduces latency in the autonomous robots, while offering implementation flexibility by applying software engineering principles. Additionally, the modular design of UBF speeds up the development and testing phases of software design and promotes the reuse of code [1].

The UBF was initially implemented on robot platforms, as a way to accomplish real-time, reactive robot control [10], [1]. In robot control implementations, a driving factor is the speed with which the robot reacts to the changing environment. Again, the current methodology for ensuring quick response time in reactive control is to tightly couple sensors to actuators through the use of a controller. Figure 5 contains a UML diagram of the Unified Behavior Framework.

1) Behavior: The initial success of the subsumption architecture came from viewing the functional units of the robot control architecture as individual robot tasks or behaviors, instead of chronological steps in the robot's decision making

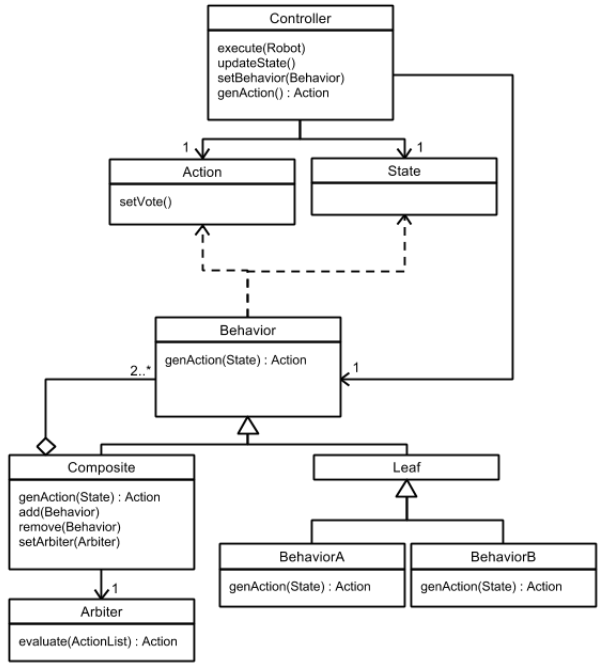

Fig. 5. A UML diagram of the Unified Behavior Framework (UBF) [10].

process. The UBF utilizes this concept, viewing the smallest units of the architecture as individual behaviors. And, taking a page from the commercial game industry, these behaviors are developed individually and added to a tree structure. However, similar to the three layer architecture, behaviors are not given access to the robot's sensors or actuators; instead, the sensing and actuation is left to the controller, discussed below. As expected, these behaviors are the central part of the agents "intelligence," and they define individual tasks that the robot intends to perform. In practice, UBF behaviors interpret the perceived state of the robot (as represented by the UBF State class). Then, based on the task being performed, the behavior may test certain conditions or otherwise evaluate the state passed to it. After interpreting the state, the behavior recommends a specific action to take. During each traversal of the UBF tree, every behavior recommends and returns an action for the robot to take.

2) Controller, State & Action: As with three layer architectures, the controller is the direct interface between the UBF and the sensors and actuators of the robot. As the layer closest to the hardware, the controller has two primary responsibilities. First, the controller develops the "world model," or the state, by interpreting the incoming sensor data. Then the controller actuates the robot's motors and controls based on the characteristics of the action output by the UBF behavior tree.

As is the case with any robot control architecture, some representation of the real world, or the world model, is present in UBF. This is referred to as the state. Through the updateState() method, the controller interprets the sensor data

for the robot. Because of the possible inaccuracies and failures of sensors in robot control applications, the state is described more accurately as the "perceived state," as the actual world state cannot be known, but can only be interpreted based on input from the sensors.

A quick philosophical aside: although we might imply that these robots are somehow inferior due to their limitations in perceiving the world state correctly, we must humble ourselves; we humans are also limited to the inputs from our "sensors" - our eyes, ears, mouth, skin, etc. So, in the same way, our understanding of the world's state may also be flawed, despite our inherent trust in our perspective.

As described in the previous section, each behavior in the UBF tree recommends an action for the robot to take. This action is a representation of what a behavior is recommending that the robot do, it does not actually control the motors on the robot, keeping in line with three layer architectures. By this method, the UBF behavior tree remains decoupled from the specifics of the robot, enhancing the flexibility of the framework for use in different applications. Actions might represent small adjustments to the robots actuators, but are typically more abstract representations, such as vectors indicating a desired direction and magnitude for the robot to go. As such, the action can be tailored to the desired effect on the robot, but the details of the actuation of controls is left to the controller, and is therefore not dealt with inside the UBF behavior tree.

Because the controller is the only direct link to the sensors and actuators, other elements of the UBF behavior tree are interchangeable between different robots by making adjustments to the controller. In the same way, differing UBF behavior trees and architectures can be swapped in and out on the same robot by retaining the controller. Due to this structuring, the behavior packages can even be swapped in and out at runtime [10].

3) Arbiter: Because each behavior recommends an action, multiple actions are being passed up the UBF behavior tree as return values from behaviors' children. Therefore, a method of choosing the "correct" action from child behaviors the UBF behavior tree is required. This is the reason for the UBF Arbiter class. The arbiter is contained within UBF behaviors that are internal nodes in the UBF behavior tree. These internal behaviors have one or more children that will be recommending actions for the robot to perform. The arbiter acts as another decision-maker, determining which of its children is the appropriate action to pass further up the tree to its parent, until the desired action is returned from the UBF behavior tree's root node (which also contains an arbiter). In this way, the root node of the tree will use its arbiter to recommend a single action, based on the returned actions of the entire tree.

Arbiters can have differing schemes for determining the most important action. Simple arbiters, such as a winner-takes-all (WTA) arbiter, might just choose the highest-voted action from that behavior's children nodes. A more complex arbiter might "fuse" multiple returned actions into one, where the components of the composite action are weighted by each individual action's vote. This is known as a "fusion" arbiter.

In a general sense, fusion arbiters combine one or more actions returned by its children in the UBF behavior tree. By "fusing," a single action will be created and returned by the fusion arbiter which has elements from multiple of the child behaviors' recommended actions. Typically, some set of the highest-voted actions returned to the fusion arbiter are selected, and those actions combinable attributes are all added to a single action which is then returned by the arbiter. There are varying ways that this can be achieved. One method would be to select the highest-voted actions, and combine their non-conflicting attributes. Or, to achieve "fairness" between the highest-voted actions, their attributes might be weighted relative to their respective votes before being "fused" into the arbiters returned action. In this way, a fusion arbiter is really a larger category of arbiters with infinite possibilities of how to combine the actions returned by the UBF tree.

The variety and customization available for arbitration implementations allows for great flexibility, whereby the entire behavior of a robot can be modified by using a different arbitration scheme, even if the rest of the UBF behavior tree remains unchanged.

D. "OpenEaagles" Simulation Framework

UBF has been implemented as a robot control architecture, but is clearly ripe for implementation in simulation. To maintain the flexibility and versatility that UBF provides, a simulation framework that was also developed using these principles is necessary. The Open Extensible Architecture for the Analysis and Generation of Linked Simulations (OpenEaagles) is such a framework. OpenEaagles is open-source, meaning that the code base is readily accessible. With the express purpose of "[aiding] the design of robust, scalable, virtual, constructive, stand-alone, and distributed simulation applications," OpenEaagles is a worthwhile tool in which to add UBF capability [11].

OpenEaagles is an open-source simulation framework that defines the design pattern shown in Figure 6 for constructing a wide variety of simulation applications. The framework itself is written in C++ and leverages modern object-oriented software design principles while incorporating fundamental real-time system design techniques to build time sensitive, low latency, fast response time applications, if needed. By providing abstract representations of many different system components (that the object-oriented design philosophy promotes), multiple levels of fidelity can be easily intermixed and selected for optimal runtime performance. Abstract representations of systems allow a developer to tune the application to run efficiently so, for example, interaction latency deadlines for human-in-the-loop simulations can be met. On the flip side, constructive-only simulation applications that do not need to meet time-critical deadlines can use models with even higher levels of fidelity.

The framework embraces the Model-View-Controller (MVC) software design pattern by partitioning functional components into packages. As shown, the Station class serves as a view-controller or central connection point that associates

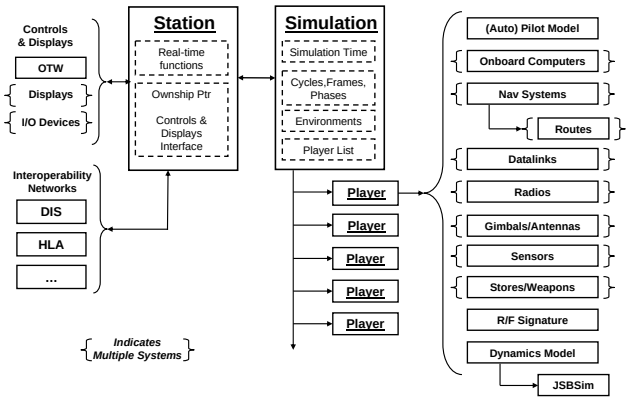

Fig. 6. A graphical depiction of the structure of the OpenEaagles simulation framework.

simulation of systems (M) with specific views (V) which include graphics, I/O and networks in the case of a distributed simulation.

As a simulation framework, OpenEaagles is not an application itself applications which are stand-alone executable software programs designed to support specific simulation experiments are built leveraging the framework.

Currently, OpenEaagles has a sophisticated autopilot system, but that is the extent of built-in mechanisms for aircraft autonomy. Other than that, no AI exists in the framework for making simulation entities autonomous. Hence, our addition of the UBF to OpenEaagles will ultimately benefit the capabilities of both frameworks.

Due to UBF's abstract design structure, it was implemented within OpenEaagles as a set of cooperating classes to define agents which can be attached to Players to provide more intelligent features than currently available. Within this structure, UBF agents have access to Player state (world model) and all Player systems which are attached as components such as antennas, sensors, weapons, etc. The Players themselves also include a sophisticated autopilot system which can be used to augment and provide low level control functionality.

E. Sweep Mission Scenario

In order to demonstrate the utility of the UBF implementation in OpenEaagles, a simple example application and scenario was created; one that would demonstrate the usefulness of being able to quickly build autonomous agents in simulation. A simple military mission known as a "sweep" was defined to test UBF-based agents. In this mission, a flight of friendly aircraft navigate towards enemy-controlled or contested airspace. The friendly aircraft search for and engage any enemy aircraft encountered, leaving the area upon destruction of the enemies or an emergency condition being met. The mission is split into four phases: Ingress, Beyond Visual Range (BVR) Engagement, Within Visual Range (WVR) Engagement, and Egress. Figure 7 and Figure 8 depict graphically the progression of the typical sweep mission.

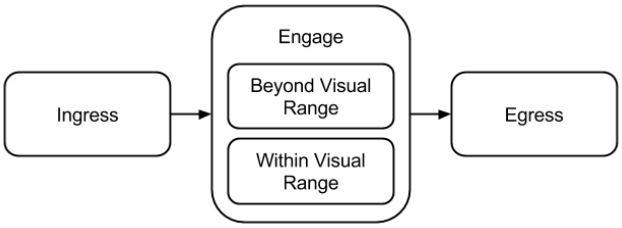

Fig. 7. A graphical depiction of the sweep mission phases.

1) Ingress Phase: The ingress phase of the sweep mission consists of navigating along a set of waypoints to the designated mission area. The flight of friendly aircraft follows the flight lead in formation towards the mission area, watching and evaluating their radar for potential enemy target aircraft. Upon acquiring a target, the friendly aircraft proceed to the engagement phase of the mission.

2) Engagement Phase: Engagement is the mission phase upon which the friendly aircraft launch missiles and fire guns against the enemy targets in attempt to shoot down those aircraft. Engagement is broken into two sub-phases based on the distance to the target.

a) Beyond Visual Range: The beyond visual range (BVR) phase of engagement consists of any combat that occurs when an enemy target is not visible to the friendly pilot through the windscreen, only on radar and signal warning systems. If targets are not detected until they are visible to the friendly pilots, it is possible to skip the BVR phase of engagement. While BVR, the friendly aircraft will confirm that the radar targets are indeed enemy aircraft, and then will engage the target(s) with long-range missiles. If the targets are not destroyed while BVR, and they become visible to the pilots, the within visual range engagement phase is entered.

b) Within Visual Range: Within Visual Range (WVR) combat occurs when enemy aircraft are close enough that the friendly pilots can see them from the cockpit, and not exclusively on radar or signal warning systems. Within visual range combat tends to involve more complicated aircraft maneuvering in order to achieve an advantageous position relative to the enemy aircraft. When an advantageous position is attained, the friendly aircraft may engage the enemy with short range missiles or guns.

3) Egress Phase: The egress phase of the mission occurs after the desired mission objective is completed; namely, if the mission area is clear of enemies. Egress may also be necessary if other emergency conditions are met. If friendly aircraft are low on fuel, or if multiple flight members have been shot down by enemy aircraft, it might be necessary to exit the mission area as quickly as possible. During egress, remaining friendly aircraft proceed to the home airfield, again, sometimes by way

of navigation waypoints exiting the mission area, or possibly by the most direct route to base.

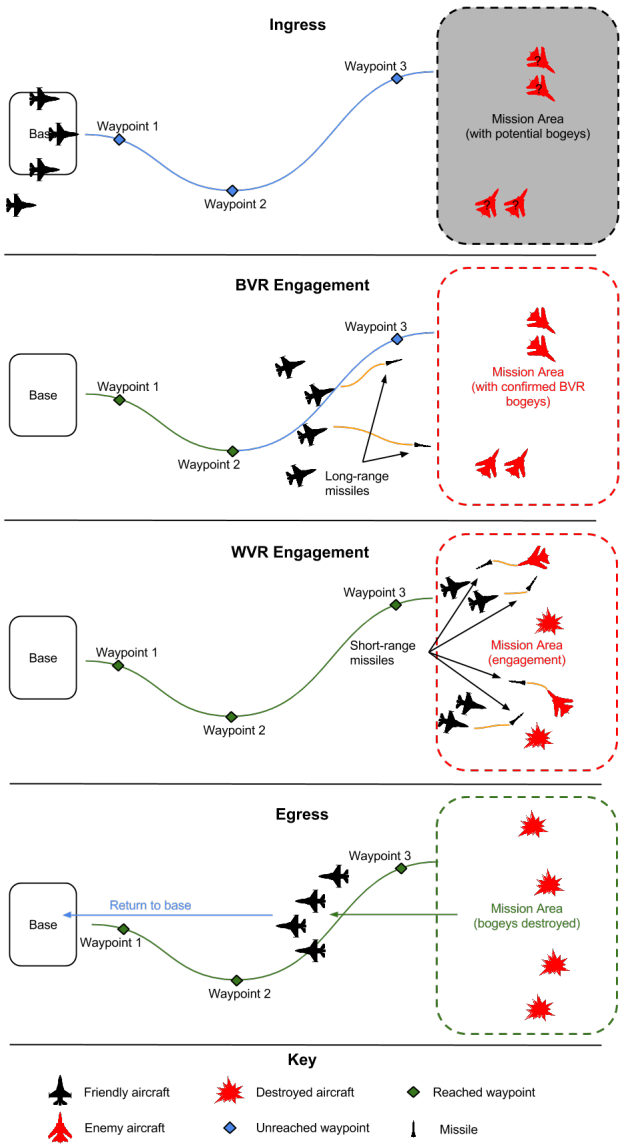

Fig. 8. A birds-eye-view depiction of the sweep mission.

III. Methods & Results

To take advantage of the flexibility of UBF and OpenEaagles, we have laid out the sweep mission scenario used to test our implementation. Having described UBF in detail, and understanding the benefits of the OpenEaagles simulation framework, we will delve into our specific implementation. Due to differences between a robot platform and a simulation environment, appropriate adjustments were made to UBF before implementing. First, we will describe UBF as implemented within OpenEaagles. The sweep mission will be highlighted again, but accounting for the details of bringing it to life in our situation. Finally, we will discuss the specific UBF behaviors built and utilized by our agents to successfully navigate the sweep mission.

A. Implementation of UBF in OpenEaagles

Using the basic structure of UBF as described in section II-C, the architecture was built into the object system provided by OpenEaagles. Then, abstract classes defined by the architecture were extended to provide specific functionality (i.e., behaviors, arbiters) relevant to the sweep mission being implemented. Some changes were made to the original UBF structure to tailor it to the OpenEaagles simulation environment, which are described in detail in the following sections. Figure 9 contains a UML diagram of the UBF including the changes that were made in the OpenEaagles implementation.

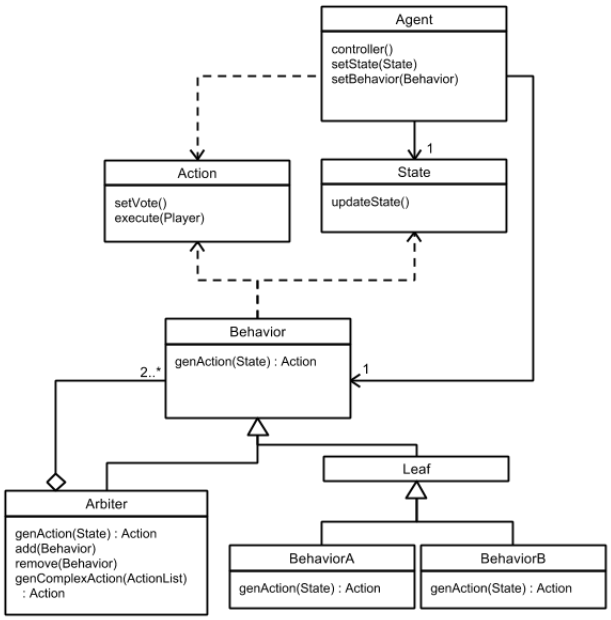

Fig. 9. A UML diagram of the OpenEaagles implementation of the Unified Behavior Framework (UBF).

1) Agent: As discussed in section II-C, UBF within a robot control application provides flexibility between platforms by utilizing different controllers that interface to hardware. Within a simulation environment, hardware is simulated, and can be accessed through the OpenEaagles object system. Therefore, a controller isn't implemented in the same way as it would be on a robot. Also, within OpenEaagles, player entities are built using the composite design pattern; each entity is a composite of many individual components, which each are composites of their own components. To maintain consistency with this design pattern, UBF needed an overarching component object that contains the whole of the UBF structure. The most effective method was to create an Agent class that contains multiple elements of the UBF, namely, the controller, the root behavior, and the state. This UBF agent can be added - as a component - to an (intended) autonomous player entity in order to add UBF functionality. Through the periodic time phase updates of the Simulation, the agent trickles down requests for updates to the state, and requests for execution of the actions on the autonomous player entity.

2) Controller: Since direct hardware access is not necessary when using a software framework like OpenEaagles, the controller was implemented somewhat differently. Not only does the controller no longer directly update the state of UBF, it is also implemented as a method in the Agent class, rather than its own class separate from actions. This structure retains the decoupling between the UBF behavior tree and the controller, and it enables actions/controllers to be tailored to a specific (simulated) platform. This is more appropriate for simulation: unlike robot architectures, the variety of platforms available in simulation means that different platforms will not only have different control mechanisms, but the actions that can logically be performed between them might be drastically different. For instance, increasing altitude on an aircraft is a logical action for that aircraft, but trying to use that Action on a ground vehicle does not make sense. In this case, it is more appropriate to have different versions of the action class in addition to differing controllers.

By implementing the controller as an action method, the UBF agent's "perceived" state is also no longer tied to the controller, but is separated into its own state class, which will be discussed in detail in the next section.

3) State: As an abstraction, state actually contains no data other than that specific to the OpenEaagles object system. Within individual implementations, state can be populated with world model information that is important to a specific agent. The controller previously contained the updateState() method, as it alone had access to the robot hardware, specifically, the sensors needed to evaluate the state. The OpenEaagles framework, allows for much wider access to the simulation environment details that might be important to a UBF agent. Therefore, updating the state in practice might occur differently than on a robot. An update can occur by evaluating the simulated sensors' input data, emulating the operation of a robot control application. However, software-simulated entities generally have privileged access to true (though simulated) world state details. In the interest of simplifying a scenario, state was granted this privileged access to the actual simulation state. On the other hand, there is flexibility to implement a more true-to-life state update, one that emulates a robot's state update process, if desired. By separating state into it's own class, rather than relegating it to the controller, the state update can be implementation-defined, adding to the flexibility of UBF in the OpenEaagles simulation environment.

It should also be noted that in OpenEaagles (as in most real-time simulation frameworks), simulation occurs via discrete time steps. Therefore, the State class contains the updateState() method that is tied to the simulation's time-step process, received as requests from the Agent class' controller() method, by which the state is updated as the simulation progresses.

4) Action: As aforementioned, the OpenEaagles implementation of the Action class includes a execute() method that interprets the details of the action and then executes it by "actuating" the relevant controls within the simulated player entity. As is the case with the state, the action/execute() combination allows for more flexibility in the implementation of UBF to specific platforms.

5) Behavior: Behaviors are the smallest functional unit of the UBF, in accordance with the original principles of the subsumption architecture. Behaviors comprise individual tasks that a player entity might perform, which might be as simple as flying straight, or as complicated as following an enemy aircraft. As the UBF's design originally intended, UBF behaviors in OpenEaagles accept the state of the UBF agent's player entity and return an action via the genAction() method. Each internal behavior node also passes the state down the tree to its children, so that every behavior in the tree will receive an updated state every time the UBF tree is polled for an action. Based on the specific behavior involved, each behavior returns a recommended action. Associated with each action is a vote, which indicates the priority of that action as determined by the behavior. A higher value vote indicates a higher priority action. As the returned actions are passed up the tree, arbiters must decide which of the actions (or which combination) will be returned further up the UBF behavior tree.

6) Arbiter: Unlike the original UBF design, arbiters are not a component of internal behavior nodes in the OpenEaagles UBF implementation. Instead, the Arbiter class is subclassed off of the Behavior class, so that the arbiters *are* the internal behavior nodes, though a more specific version of a behavior. In a nutshell, this implementation combines "arbiter" functionality with the composite behavior. This facilitates the selection of actions as behaviors return actions up the UBF tree. Each arbiter, as described, has some decision scheme that selects or constructs the action that is returned up the UBF behavior tree. In the OpenEaagles implementation, the Arbiter class includes a genComplexAction() method, which is the method for returning an action based on the recommendations of its children.

B. Scenario Implementation

1) Reducing the complexity: Complexity is a very relevant issue when trying to build a well-software-engineered product. While any project will become more complex as it grows, the intent is generally to reduce complexity and maintain simplicity. In this case, reducing the complexity of the scenario was necessary to obtain an effective demonstration.

To reduce the complexity, some sacrifices were made with regard to the pilot mental model fidelity. Where pilots might fly specific maneuvers in order to pursue an enemy aircraft, the UBF agent essentially turns on the autopilot and sets it to follow the enemy aircraft. In the same way, the pilots defensive maneuvering is limited to a break maneuver, whereas a human pilot likely has a large repertoire of defensive maneuvers at his/her disposal to defend against an incoming missile or a pursuing enemy. These sacrifices were necessary to successfully implement the desired scenario, but with more work and study on a human pilots decision making, a much more accurate representation of the pilots mind could be obtained with the UBF tree.

In addition to the mental model fidelity, complexity was also reduced with regards to the maneuverability of the

aircraft. The OpenEaagles simulation framework includes a very detailed aerodynamics model called JSBSim. In order to create a more manageable implementation, however, this UBF agent utilizes a more simplistic aerodynamics model called the LaeroModel. While the LaeroModel prevents hands-on-stick-and-throttle (HOTAS) control of the aircraft allowing for detailed maneuvers and upside down flight, the simplicity of the model interface greatly reduces the UBF action code required. This was a necessary and acceptable sacrifice in order to implement the sweep mission scenario. As with any simulation, detail is a function of the defined experiment.

2) Scenario Arbiters, State, and Action classes:

a) Arbiters: As mentioned in section II-C3, there are a variety of arbitration schemes available to facilitate decision making in the UBF behavior tree. In our scenario, two separate Arbiters were designed. Unfortunately, due to time constraints, only one was tested and verified with the sweep mission scenario.

b) Winner-takes-all Arbiter: The winner-takes-all (WTA) arbiter simply selects the action with the highest vote. This is the simpler of the two arbiters implemented, not requiring any special manipulation of returned actions. Because of its simplicity, the WTA was the arbiter used in the scenario implementation. This allowed for straightforward construction of the UBF behavior tree and unambiguous confirmation that behaviors were responding as expected.

c) Fusion Arbiter: In addition to the WTA arbiter, a simple fusion arbiter was implemented, but again, it was not tested or verified. Our arbiter takes an extremely basic approach, simply averaging the altitude, velocity, and heading components of each action, and launching a missile if there is a highly-voted action recommending weapons-release.

d) PlaneState: For the OpenEaagles implementation, the PlaneState class was subclassed off of the generic State class. This class contains useful information to an aircraft such as heading, altitude, velocity, missiles onboard, etc. During the updateState() routine, the PlaneState class polls the simulation to ascertain and populate the PlaneState object. Each of the UBF agents has a state created when the agent is initialized, and the state is not destroyed, rather it is changed as it is updated with the simulation time steps.

e) PlaneAction: The PlaneAction class is the subclass of Action that implements actions for the aircraft agent. Some leniency had to be taken with this class in order to simplify the flying of the aircraft. While specific controls such as the throttle or control column could be actuated to direct the aircraft to the desired vector or location, this is clearly an extremely complicated way of flying the aircraft. Essentially, a UBF agent would require the flying skill of an experienced pilot in order to even perform basic flight maneuvers. As implemented, however, actions are able to use more effective, if less realistic control methods, without requiring an experienced pilot's flying ability. OpenEaagles provides a simplistic aerodynamic model that will "fly" the plane absent of any actual inputs to the simulated controls; only a basic understanding of some elements of flight (altitude, velocity, and heading) is required. In this manner, the PlaneActions controller() method stores the details of the intended action, and modifies the heading, velocity, altitude, launches missiles, engages the autopilot, etc., to allow the aircraft to act according to the agents desire.

Again, the inherent flexibility of this implementation method allows for a future implementation to utilize a more accurate emulation of the original UBF's design, if desired.

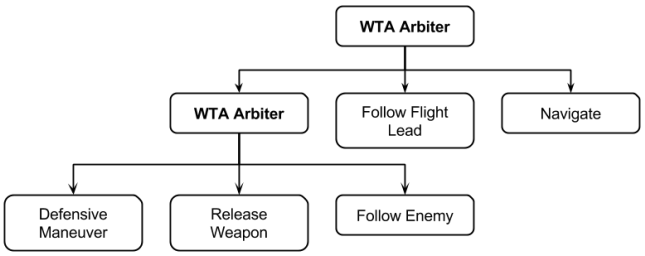

Fig. 10. The UBF behavior tree for the sweep mission scenario.

3) UBF behaviors: In designing the scenario, multiple behaviors were created so that the pilot agents could seek out their sweep mission goal of destroying the enemy aircraft. These behaviors are implemented for the agent's navigation, missile evasion, pursuit of the enemy, and weapons release. They are discussed individually in the sections below. Figure 10 shows the UBF tree structure for the sweep mission scenario.

a) Navigate: In order to successfully complete the mission, the UBF agent needs a way of navigating along a mission path towards the intended mission area. In a real sweep mission, the intended waypoints would be known and planned ahead of mission execution, and the pilot would follow those waypoints until the engagement phase. In this way, the mission waypoints were programmed into the navigation computer of the aircraft before the mission started. The UBF agent turns on the autopilot, instructing it to follow those waypoints, in order to execute the navigation required for the mission.

b) Follow the Lead: Following is a behavior that is necessary for formation flight. While having all of the friendly UBF agents navigating to the same waypoints might imitate this behavior, it does not truly replicate how a pilot would behave, keeping track of the lead aircraft and following his movements. That being said, however, all of the friendly UBF agents in our scenario were given knowledge of the waypoints within their navigation computers. This allowed for an agent to take over as the flight lead if the current one was shot down.

Because a particular formation is specified, the wall formation (shown in figure 11), the UBF agents can use their flight ranking to determine their physical position in relation to the flight lead. In this way, the UBF agent can tell its flight ranking based on the players predefined name assigned when constructing the simulation. To make things simpler, a naming convention of "*(flight name)(rank)*" was used to identify which flight the agent is a part of, and their intended rank in the flight.

As rank could change if flight leaders were shot down, there was a mechanism built into PlaneStates updateState() method that determines the actual current ranking, rather than just the original predefined ranking.

To actually follow the flight leader in proper formation, the autopilot was again utilized for the convenient functionality provided within OpenEaagles. The autopilot has a following mode built in, which allows the user to define who to follow, and the position relative to the leader. For instance, in our scenario, "eagle2" followed 6000 feet left, 1000 feet behind, and 500 feet below eagle1.

Utilizing this autopilot functionality, along with the naming convention that defines a flight and its members, the follow behavior was implemented that allows the "eagle" flight (and the enemy "bogey" flight) to fly in wall formation during any non-engagement portions of the scenario.

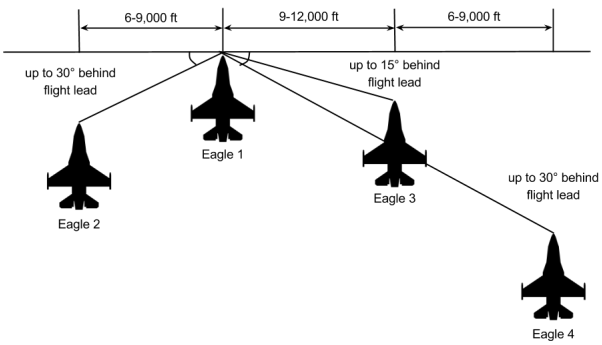

Fig. 11. A graphical depiction of the "wall" flight formation.

c) Pursue an Enemy: Pursuing the enemy is a behavior that is necessary for eventually attacking the enemy, which ultimately is the purpose of the sweep mission. To implement this behavior, again, the autopilot following functionality was utilized. This, while not an accurate representation of how a pilot might maneuver to engage an enemy, does provide a simple, convenient way to implement the pursuing behavior. This is certainly an area for future improvement, whereas a complex model that is more representative of an actual pilot could be implemented.

In this case, the UBF agent first detects the enemy using its onboard radar systems. After detecting the enemy, the agent is given special access to simulation information about the enemy player in order to provide the data necessary for the autopilot to enter following mode against that enemy.

d) Release Weapon: As it is the ultimate mission of the sweep mission, the UBF agent requires a behavior that decides when to release a weapon against an enemy target. A pilot would normally have some idea of how probable a kill is based on the location of the enemy aircraft in relation to his own aircraft. The term for the region with the highest probability of a kill is a weapons employment zone, or WEZ.

The UBF agent evaluates whether an enemy target is visible (on radar), and then whether that target is within the agents WEZ. If so, the behavior recommends the release of a weapon, which is performed through the stores management system of the UBF agents player.

e) Break (Defensive Maneuver): Finally, a maneuver that attempts to evade incoming missiles is necessary. This behavior detects a missile based on its radar track. As with the pursuit of an enemy aircraft, this behavior could be modified to be more accurate to a true pilots behavior. In the meantime, the detection of the missile is performed within the simulation, which of course has omniscience about whether the radar track is a missile or not.

Once detected, the incoming missile also needs to be determined to be coming at the UBF agent interested in it. As before, the simulation is polled to determine the missiles target. If the target is the current UBF agents player entity, the UBF agent knows that the missile is pursuing it, and can then initiate defensive maneuvering.

In order to be simple, the current defensive maneuver implementation has two phases. The first phase occurs if the missile is detected outside of a two nautical mile radius of the UBF agent. When the missile is far away, as determined by this arbitrary boundary, the UBF agent maneuvers his plane towards the incoming missile, and increases altitude. This is designed as a preparation phase for when the missile is danger close, within the two nautical mile radius. Upon the missile breaching two nautical miles, the UBF agent then performs a break maneuver, or a hard, diving turn (to either side, depending on the angle of the incoming missile).

IV. Conclusion

Through the development and implementation of the Unified Behavior Framework within the OpenEaagles simulation framework, we have demonstrated the potential for creating simulated autonomous agents in a military simulation context. Some specific issues that arose during the process were the granularity of behaviors, and the contrast between UBF and behavior trees. In this section, we will briefly discuss these issues as they relate to our implementation.

A. Granularity of behaviors

A difficult design decision presented itself when building the UBF tree of behaviors for our scenario. Behaviors can be as "simple" as performing a basic stick or throttle control change, but they can also be very complex, attempting to attain a specific heading, altitude and velocity by a long series of control input changes. When designing behaviors, it is necessary to make some decisions about how complex, or "granular," the individual UBF behaviors will be. The granularity of the behaviors will also have a direct effect on the size of the UBF behavior tree, and it can affect the arbitration scheme drastically. As the behaviors become simpler and smaller, the UBF tree will grow, and vice versa. WTA arbiters are useful for "large grain" behaviors and small trees, while a fusion arbiter becomes much more interesting as the UBF tree grows and includes "small grain" behaviors that can be fused in interesting ways.

In this case, the design decision was made to allow for very complex, "large grain" behaviors. In this way, the scenario behavior tree remained fairly small in size. This decision was due to the exponential jump in complexity of breaking some tasks down into multiple behaviors. In addition, behaviors that utilized the autopilot navigation and follow modes would have been much more complex if not using the autopilot, and instead building multiple less-complex, autopilot-lacking behaviors. Instead of having a large UBF sub-tree dedicated to navigating to the next waypoint, the UBF agent only required one behavior that turned on the autopilot when navigation was the desired behavior. Though it results in a much more complex exhibited behavior, by choosing this level of granularity, the behavior was actually much simpler to implement.

B. UBF versus Behavior Trees

A question that arose when implementing our sweep mission scenario using UBF was, would the sweep mission be easier to implement with Behavior Trees? The answer, of course, is complicated. When thinking about the sweep mission scenario, the behaviors desired from the pilot agent are well-understood and well-defined. This lends itself to behavior trees, with behaviors that are expected and scripted, rather than unexpected, or "emergent" behaviors. Clearly, the benefits of UBF are lost on such a simple and well-defined scenario. On the other hand, the design elements of UBF lend it to future experimentation within the simulation environment. With the UBF framework in place, the opportunity to simulate pilot agents that exhibit unpredictable behavior is now ripe for exploration. Instead of defining a scenario based on detailed pilot procedures, agents can be designed to behave like we would expect a pilot to in various situations, and then put those agents through their paces to understand how an agent might behave under unpredictable circumstances. While behavior trees would presumably produce consistent behavior, the UBF agents would allow for emergence of behaviors that give deeper insight into agent design.

C. Future Work

1) Fusion arbiter and emergent behaviors: One of the major benefits of the Unified Behavior Framework's arbiter scheme, is the opportunity for emergent behavior. Emergent behavior is somewhat of a misnomer; in truth, the actions are emergent. Specific behaviors in the UBF tree are deterministic when considered on their own. When utilizing an arbitration scheme that allows for actions to combine multiple behaviors returned actions, such as fusion, those deterministic responses can now become unpredictable, or emergent. While this may produce odd and possibly detrimental behavior, it also provides for complex combinations of actions that may have been unexpected. By introducing this element of unpredictability and randomness, the capability of the UBF agent grows beyond that of the scripted nature of behavior trees.

A fusion arbiter was developed as part of this effort, but it was not utilized as part of the scenario. Along with increasing the fidelity of the pilot mental model, the fusion arbiter is certainly ready for future work.

2) Sequencer: Three layer architectures demonstrate the usefulness of separating complex planning algorithms from the reactive control mechanisms needed for rapid action in dynamic environments. In our scenario, these higher-level planning activities were not necessary, as our agents were seeking a very specific goal: destroy any enemies encountered. On the other hand, a real-world pilot would likely come across situations that required a change of goal; an emergency condition or a change of waypoints. While our agents' single-mindedness did not affect the results of the simulation, it demonstrates a lack of capability that could be remedied with the addition of a sequencer to the OpenEaagles UBF agent. In later implementations, adding a sequencer would be an effective way to give planning abilities, so that agents could switch between UBF behavior trees if a goal change was necessary during the middle of a mission.

Acknowledgment

The authors would like to acknowledge the support of David P. Gehl of L-3 Communications for his support in understanding the design and organization of the OpenEaagles framework.

References

[1] B. G. Woolley, G. L. Peterson, and J. T. Kresge, "Real-time behavior-based robot control," *Autonomous Robots*, vol. 30, no. 3, pp. 233–242, 2011.

[2] V. Braitenberg, *Vehicles: Experiments in Synthetic Psychology*, ser. Bradford Books. MIT Press, 1986. [Online]. Available: http://books.google.com/books?id=7KkUAT_q_sQC

[3] N. J. Nilsson, "Shakey the robot," DTIC Document, Tech. Rep., 1984.

[4] E. Gat *et al.*, "On three-layer architectures," 1998.

[5] R. A. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14–23, 1986.

[6] R. C. Arkin, "Survivable robotic systems: Reactive and homeostatic control," in *Robotics and remote systems for hazardous environments*. Prentice-Hall, Inc., 1993, pp. 135–154.

[7] R. Peter Bonasso, R. James Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. G. Slack, "Experiences with an architecture for intelligent, reactive agents," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2-3, pp. 237–256, 1997.

[8] D. Isla, "Gdc 2005 proceeding: Handling complexity in the halo 2 ai," *Retrieved October*, vol. 21, p. 2009, 2005.

[9] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ogren, "Towards a unified behavior trees framework for robot control," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, 2014.

[10] B. G. Woolley and G. L. Peterson, "Unified behavior framework for reactive robot control," *Journal of Intelligent and Robotic Systems*, vol. 55, no. 2-3, pp. 155–176, 2009.

[11] D. D. Hodson, D. P. Gehl, and R. O. Baldwin, "Building distributed simulations utilizing the eaagles framework," in *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*, vol. 2006, no. 1. NTSA, 2006.

[12] M. Cutumisu and D. Szafron, "An architecture for game behavior ai: Behavior multi-queues." in *AIIDE*, 2009.

[13] A. September, "Ieee standard glossary of software engineering terminology," *Office*, vol. 121990, no. 1, p. 1, 1990.

[14] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

IV. Conclusion

The demonstration of the sweep mission within the UBF confirms the utility of our implementation within OpenEaagles. The usefulness in a military context was also demonstrated, by the fact that a combat scenario was successfully implemented.

Through the development and implementation of this research, it became clear that implementing the sweep mission was not a simple task.

4.1 Controller Complexity

Though a successful demonstration was achieved through the sweep mission scenario, some obstacles manifested during development. Despite the apparent simplicity of the sweep mission, its implementation is much more complex than just constructing a few UBF behaviors. The construction of the controller turned out to be quite difficult. As the OpenEaagles simulation environment attempts to maintain an extremely high level of fidelity to the real world, the task of controlling an aircraft required the knowledge and skill of a trained pilot. Besides the obvious difficulty of flying a plane, complexity in designing the controller also arose out of the environment in which the agent was operating. Moving in three-dimensional space is no easy task, especially when also considering the complicated aerodynamics involved in fixed-wing flight. To reduce this burden, a simpler aerodynamics model, the LaeroModel, was used to implement manual aircraft maneuvers. This model did not require hands on throttle-and-stick (HOTAS) control of the aircraft, but was able to "fly" the plane when provided with a desired heading, altitude, and velocity. This massively simplified the design of the UBF agent, and reduced the necessity for pilot-level skill in designing behaviors and actions.

When deciding how best to navigate and fly in formation, the simplicity offered by the OpenEaagles autopilot quickly became the favorable option over splitting individual

behaviors into a navigation and following sub-tree, respectively. Again, while it would have been possible to control the aircraft directly by using HOTAS control, the autopilot functionality required no pilot-equivalent skill or understanding of flight mechanics, and was therefore the preferred method of flying the aircraft. Additionally, this may not be such an inaccurate representation of a human pilot, who would clearly prefer the ease and efficiency of the autopilot, especially when engaging in other tasks like scanning his radar screen or communicating by radio.

In addition to the difficulty of controlling the aircraft, evaluating the state of the simulation was quite complex. As a result, some parts of the aircraft state had to be retrieved from the simulation directly, rather than from the aircraft's sensors, which would have been a more accurate representation of an autonomous robot architecture.

For example, when a human pilot sees blips on his radar screen, he cannot assume that those blips represent enemy aircraft. The blips could be missiles, friendly aircraft, civilian aircraft, flocks of birds, high terrain, etc. A robot implementation of a UBF agent is also limited in this way, only having access to the sensor data, not the true world state. To emulate an agent in the most true-to-life way, we would have to imbue it with the ability to reason over these radar blips, and the agent would need access to other sensor systems such as Identification Friend or Foe (IFF). The exponential jump in complexity of implementing such ability would have quickly consumed all available design and development effort. Instead of dealing with the extra effort, it was much more effective to grant a bit of extra knowledge to our agent, by allowing it access to simulation information that would confirm a radar track as an enemy aircraft, a missile, or otherwise.

Considering the complexity of implementing the agent's state and actions, it is clear that much of the effort of implementing an autonomous agent is wrapped up in developing the controller of that agent. For our implementation within the simulation environment, some sacrifices were made in order to achieve a proof-of-concept scenario, and to ensure

that the entire UBF structure could be exercised sufficiently. Section 4.4.3 discusses the areas of future work that could improve upon these workarounds and generate a more realistic representation of an autonomous robot in the simulation.

4.2 UBF versus Behavior Trees

In the context of robot architectures, the UBF indeed is a preferred method for encouraging code reuse and promoting design flexibility. However, within simulation, an argument could be made that behavior trees would be a quicker and more efficient way of implementing simple scenarios such as the sweep mission. Because simulation entities have software-level access to the true world state, a behavior tree autonomous agent could effectively be used in the same way that commercial games do.

While this is indeed true, in another sense, it circumvents the main objective of this research. In pursuing the value that simulation provides, we ultimately desire the real representation of an autonomous agent within that simulation. Utilizing a behavior tree to control that agent would defeat this purpose by violating the true capabilities of the robot to understand its state. As discussed in section 4.1, some workarounds were utilized to implement our scenario within a reasonable timeframe. While using behavior trees would have required the same workarounds, the addition of the UBF's capabilities to the OpenEaagles framework allows for future development of fully realistic representations of autonomous robots. For this reason, the utilization of behavior trees would have severely limited the representation of an autonomous robot in simulation.

4.3 Granularity of Behaviors

When building the UBF behavior tree, the size of each behavior revealed itself to be an issue. While small behaviors that focus on limited tasks is seemingly the best way to build a tree, this proved to be difficult to implement. In addition, some of the workarounds,

such as the autopilot and the LaeroModel, required larger behaviors in order to adequately accomplish the sweep mission.

4.4 Future Work

Though this implementation of UBF certainly demonstrates its utility and flexibility in developing autonomous agents in simulation, there is clearly room for future development above and beyond the research done in this thesis. By adding variety to the arbitration schemes available for use, the UBF will attain the full potential of flexibility for implementing different robot architectures. By taking a page from three layer architectures and adding a sequencer to the UBF, it would certainly enable goal-switching in a way not available under the current framework structure. Finally, in response to the complexity issues faced during this research, a multitude of improvements could be made to the scenario implementation that increase the accuracy of the simulated autonomous agents within the OpenEaagles simulation environment.

4.4.1 Fusion Arbiter.

As mentioned in the paper in chapter 3, a fusion arbiter was designed and coded as a part of this research. However, due to time constraints, the fusion arbiter was unable to be tested with the sweep mission scenario.

Fusion arbiters, as described, are useful in exploring “emergent” behavior. Because the actions produced by a fusion arbitration scheme are ultimately combinations of multiple recommended actions, those actions are not that of any specific behavior. Therefore, these “fused” actions are sometimes unpredictable or unexpected.

Implementing the fusion arbiter adds a great deal of capability to the UBF implementation, and therefore is a natural progression beyond the current research’s winner takes all (WTA) arbitration scheme.

4.4.2 Sequencer.

Three layer architectures apply modularity to their design, separating higher-level planning and goal-determining behaviors from the low-level reactive behaviors. The UBF structure includes no mechanism for implementing goal-switching or planning outside of the UBF behavior tree. By adding a sequencer to the UBF implementation, goal-switching could be enabled within the behavioral architectures that the UBF is capable of implementing.

This lends itself to extensive military missions with multiple phases, and multiple goals, but it can also be applied to the sweep mission scenario. For instance, a navigation UBF behavior tree with a goal of navigating to a certain waypoint could be implemented, as well as an engagement UBF behavior tree devoted to the dogfighting portion of the scenario. With the addition of a sequencer, the mission phases now fall naturally into either of these UBF behavior trees. Ingress and egress are clearly just navigation from waypoint to waypoint. A sequencer can manage which waypoint is being pursued, as the agent moves along the mission path. Then, when an enemy is encountered, either on radar or by direct engagement, the sequencer would be able to switch out the navigation UBF tree with the engagement UBF tree. In this way, military missions could be broken down into minor goals or tasks, and individual UBF trees could be built per goal.

Adding a sequencer to the UBF structure within OpenEaagles, though not part of the original design of the UBF, would obviously extend the current capabilities of the framework. Not only that, but the ability to build UBF behavior trees on a by-goal or by-task basis lends itself to the military context due to the goal-based nature of most combat missions.

4.4.3 Addressing the Complexity.

As discussed in section 4.1, implementing a controller for an aircraft-based autonomous agent is quite complex. In this research, the sweep mission scenario

implementation simplified certain elements of the controller in order to work around those complexities. Because of those workarounds, the autonomous agent controlling the aircraft in the scenario is not quite a true-to-life representation of an autonomous robot.

By correcting these workarounds, and by embracing these complexities, the scenario and the relevant UBF behaviors could be more accurately implemented. With a more accurate implementation of these behaviors, autonomous agents developed in the simulation could be applied to real robotic systems after exploration, research and development within the OpenEaagles environment.

Appendix: Implementation Code

A.1 OpenEaagles UBF Implementation Code

A.1.1 Action.h.

```
//-----------------------------------------------------------------------------
// Class: Action
//-----------------------------------------------------------------------------
#ifndef __Eaagles_Basic_Ubf_Action_H__
#define __Eaagles_Basic_Ubf_Action_H__

#include "openeaagles/basic/Object.h"

namespace Eaagles {

namespace Basic {
   class Component;

namespace Ubf {

//-----------------------------------------------------------------------------
// Class: Action
//
// Description:
//   Abstract base class for all Actions. They are responsible for
//   their own execution.
//
// Factory name: UbfAction
```

```
//-----------------------------------------------------------------------------
class Action : public Basic::Object
{
   DECLARE_SUBCLASS(Action, Basic::Object)
public:
   Action();

   unsigned int getVote() const;
   void setVote(const unsigned int x);

   // Execute the behavior
   virtual bool execute(Basic::Component* actor)=0;

private:
   unsigned int vote;
};

inline void Action::setVote(const unsigned int x) { vote = x; return; }
inline unsigned int Action::getVote() const    { return vote; }

} // End Ubf namespace
} // End Basic namespace
} // End Eaagles namespace

#endif
```

A.1.2 Action.cpp.

```
//-----------------------------------------------------------------------------
// Class: Action
//-----------------------------------------------------------------------------

#include "openeaagles/basic/ubf/Action.h"

namespace Eaagles {
namespace Basic {
namespace Ubf {

IMPLEMENT_ABSTRACT_SUBCLASS(Action, "UbfAction")
EMPTY_SLOTTABLE(Action)
EMPTY_DELETEDATA(Action)
EMPTY_SERIALIZER(Action)

//-----------------------------------------------------------------------------
// Class support functions
//-----------------------------------------------------------------------------

Action::Action()
{
   STANDARD_CONSTRUCTOR()
   vote = 0;
}

void Action::copyData(const Action& org, const bool cc)
{
```

```
    BaseClass::copyData(org);
    vote = org.vote;
}


} // End Ubf namespace
} // End Basic namespace
} // End Eaagles namespace
```

A.1.3 Agent.h.

```
//-----------------------------------------------------------------------------
// Class: Agent
//-----------------------------------------------------------------------------
#ifndef __Eaagles_Basic_Ubf_Agent_H__
#define __Eaagles_Basic_Ubf_Agent_H__

#include "openeaagles/basic/Component.h"

namespace Eaagles {
namespace Basic {
namespace Ubf {

class Behavior;
class State;
class Action;

//-----------------------------------------------------------------------------
// Class: Agent
//
// Description: Generic agent class to control a component in the
   simulation - the agent's "actor"
//
// Notes:
// 1) Use 'Agent' to update the behavior framework via updateData() and use
//   'AgentTC' to update the behavior framework using updateTC().
//
```

```
// 2) The updateData() and updateTC() calls are only processed by this
    Agent
//   class and are not passed to the rest of the behavior framework.
//
//
// Factory name: UbfAgent
// Slots:
//   state     <State>   ! The agent's state object
//   behavior  <Behavior> ! behavior
//-----------------------------------------------------------------------------
class Agent : public Basic::Component
{
  DECLARE_SUBCLASS(Agent, Basic::Component)
public:
  Agent();

  // Basic::Component Interface
  virtual void updateTC(const LCreal dt = 0.0f);
  virtual void updateData(const LCreal dt = 0.0f);
  virtual void reset();

protected:
  // generic controller
  virtual void controller(const LCreal dt = 0.0f);

  Behavior* getBehavior() const     { return behavior; }
  void setBehavior(Behavior* const);
```

```
   State* getState() const          { return state; }
   void setState(State* const);

   virtual void initActor();

   Basic::Component* getActor();
   void setActor(Basic::Component* const myActor);

   // slot functions
   virtual bool setSlotBehavior(Behavior* const);
   bool setSlotState(State* const state);

private:
   Behavior* behavior;
   State* state;
   SPtr<Basic::Component> myActor;
};

inline void Agent::setActor(Basic::Component* const actor) { myActor =
    actor; return; }
inline Basic::Component* Agent::getActor()            { return myActor; }



//-----------------------------------------------------------------------------
// Class: Agent
//
// Description: Generic agent class to control a component - the agent's
    "actor"
```

```
// - a derived agent class that performs its actions in the TC thread
//
// Factory name: UbfAgentTC
//-----------------------------------------------------------------------------
class AgentTC : public Agent
{
   DECLARE_SUBCLASS(AgentTC, Agent)
public:
   AgentTC();
   // Basic::Component Interface
   virtual void updateTC(const LCreal dt = 0.0f);
   virtual void updateData(const LCreal dt = 0.0f);
};

} // End Ubf namespace
} // End Basic namespace
} // End Eaagles namespace

#endif
```

A.1.4 Agent.cpp.

```
//-----------------------------------------------------------------------------
// Agent
//-----------------------------------------------------------------------------

#include "openeaagles/basic/ubf/Agent.h"
#include "openeaagles/basic/ubf/Action.h"
#include "openeaagles/basic/ubf/Behavior.h"
#include "openeaagles/basic/ubf/State.h"

#include "openeaagles/basic/Pair.h"
#include "openeaagles/basic/String.h"

namespace Eaagles {
namespace Basic {
namespace Ubf {

//
// Class: Agent
//
// Description: An Agent that manages a component (the "actor") with a
    behavior
//            (either a player, or a player's component)
//

IMPLEMENT_SUBCLASS(Agent, "UbfAgent")
EMPTY_SERIALIZER(Agent)
EMPTY_COPYDATA(Agent)
```

```
//-----------------------------------------------------------------------------
// slot table for this class type
//-----------------------------------------------------------------------------
BEGIN_SLOTTABLE(Agent)
   "state",                  //  1) The agent's state object
   "behavior"                //  2) behavior
END_SLOTTABLE(Agent)

//  mapping of slots to handles
BEGIN_SLOT_MAP(Agent)
   ON_SLOT(1, setSlotState, State)
   ON_SLOT(2, setSlotBehavior, Behavior)
END_SLOT_MAP()



//-----------------------------------------------------------------------------
// Class support functions
//-----------------------------------------------------------------------------
Agent::Agent()
{
   STANDARD_CONSTRUCTOR()
   myActor = 0;
   behavior = 0;
   state = 0;
}

void Agent::deleteData()
```

```
{
   if ( behavior!=0 ) { behavior->unref(); behavior = 0; }
   if ( state!=0 ) { state->unref(); state = 0; }

   myActor = 0;
}

//-----------------------------------------------------------------------------
// Reset the system
//-----------------------------------------------------------------------------
void Agent::reset()
{
   // Reset our behavior and state objects
   if (behavior != 0) {
      behavior->reset();
   }
   if (state != 0) {
      state->reset();
   }

   myActor=0;
   initActor();

   // although state is not explicitly initialized as component, the set
       state
   // method sets up the component relationship since state is a
       component, it
   // will get the reset() this way (via the component i/f)
```

```
   BaseClass::reset();
}



//-----------------------------------------------------------------------------
// updateTC() -
//-----------------------------------------------------------------------------
void Agent::updateTC(const LCreal dt)
{
}



//-----------------------------------------------------------------------------
// updateData()
//-----------------------------------------------------------------------------
void Agent::updateData(const LCreal dt)
{
   controller(dt);
}



//-----------------------------------------------------------------------------
// updateData()
//-----------------------------------------------------------------------------
void Agent::controller(const LCreal dt)
{
   Basic::Component* actor = getActor();
```

```
  if ( (actor!=0) && (getState()!=0) && (getBehavior()!=0) ) {

    // update ubf state
    getState()->updateState(actor);

    // generate an action, but allow possibility of no action returned
    Action* action = getBehavior()->genAction(state, dt);
    if (action) {
      action->execute(actor);
      action->unref();
    }
  }
}



//-----------------------------------------------------------------------------
// Set our behavior model
//-----------------------------------------------------------------------------
void Agent::setBehavior(Behavior* const x)
{
  if (x==0)
    return;
  if (behavior!=0)
    behavior->unref();
  behavior = x;
  behavior->ref();
  behavior->container(this);
}
```

```
//-----------------------------------------------------------------------------
// Set our state model
//-----------------------------------------------------------------------------
void Agent::setState(State* const x)
{
   if (x==0)
      return;
   if (state!=0)
      state->unref();
   state = x;
   state->ref();
   state->container(this);
   Basic::Pair* p = new Basic::Pair("", state);
   addComponent(p);
   p->unref();
}



//-----------------------------------------------------------------------------
// finds our actor during reset() processing
//-----------------------------------------------------------------------------
void Agent::initActor()
{
   if (getActor()==0) {
      // our actor is our container
      if (container() != 0) {
```

```
        setActor(container());
      }
   }
}



//-----------------------------------------------------------------------------
// set slot functions
//-----------------------------------------------------------------------------

// Sets the state object for this agent
bool Agent::setSlotState(State* const state)
{
   bool ok = false;
   if (state != 0) {
      setState(state);
      ok = true;
   }
   return ok;
}

bool Agent::setSlotBehavior(Behavior* const x)
{
   bool ok = false;
   if ( x!=0 ) {
      setBehavior(x);
      ok = true;
   }
```

```
  return ok;
}

//-----------------------------------------------------------------------------
// getSlotByIndex()
//-----------------------------------------------------------------------------
Basic::Object* Agent::getSlotByIndex(const int si)
{
  return BaseClass::getSlotByIndex(si);
}



//=============================================================================
// Class: AgentTC
// Description: An Agent that manages a component (the "actor") with a
   behavior,
//            using TC thread to perform its activity (instead of BG
   thread)
//=============================================================================

IMPLEMENT_SUBCLASS(AgentTC, "UbfAgentTC")
EMPTY_SLOTTABLE(AgentTC)
EMPTY_CONSTRUCTOR(AgentTC)
EMPTY_SERIALIZER(AgentTC)
EMPTY_COPYDATA(AgentTC)
EMPTY_DELETEDATA(AgentTC)

//-----------------------------------------------------------------------------
```

```
// updateTC() - Calls the controller
//-----------------------------------------------------------------------------
void AgentTC::updateTC(const LCreal dt)
{
   controller(dt);
}

//-----------------------------------------------------------------------------
// updateData() -
//-----------------------------------------------------------------------------
void AgentTC::updateData(const LCreal dt)
{
}

} // End Ubf namespace
} // End Basic namespace
} // End Eaagles namespace
```

A.1.5 Arbiter.h.

```
//-----------------------------------------------------------------------------
// Class: Arbiter
//-----------------------------------------------------------------------------
#ifndef __Eaagles_Basic_Ubf_Arbiter_H__
#define __Eaagles_Basic_Ubf_Arbiter_H__

#include "Behavior.h"

namespace Eaagles {

namespace Basic {
   class List;

namespace Ubf {
   class State;
   class Action;

//-----------------------------------------------------------------------------
// Class: Arbiter
//
// Description:
//    A meta-behavior that generates a "complex action" based on the actions
//    generated our list of behaviors.
//
// Note:
//    The default is to select the Action with the highest vote value.
//
```

```
// Factory name: UbfArbiter
// Slots:
//   behaviors <PairStream>   ! List of behaviors
//-----------------------------------------------------------------------------
class Arbiter : public Behavior
{
  DECLARE_SUBCLASS(Arbiter, Behavior)

public:
  Arbiter();

  // Basic::Ubf::Behavior class functions
  virtual Action* genAction(const State* const state, const LCreal dt);

protected:
  Basic::List* getBehaviors();

  // evaluates a list of actions and return an optional "complex action"
  // (default: returns the action with the highest vote value)
  virtual Action* genComplexAction(List* const actionSet);

  // add new behavior to list
  void addBehavior(Behavior* const);

  // slot functions
  bool setSlotBehaviors(Basic::PairStream* const);

private:
```

```
  Basic::List* behaviors;
};

inline Basic::List* Arbiter::getBehaviors()        { return behaviors; }

} // End Ubf namespace
} // End Basic namespace
} // End Eaagles namespace

#endif
```

A.1.6 Arbiter.cpp.

```
//-----------------------------------------------------------------------------
// Class: Arbiter
//-----------------------------------------------------------------------------

#include "openeaagles/basic/ubf/Arbiter.h"
#include "openeaagles/basic/ubf/Action.h"

#include "openeaagles/basic/Pair.h"
#include "openeaagles/basic/PairStream.h"

namespace Eaagles {
namespace Basic {
namespace Ubf {

IMPLEMENT_SUBCLASS(Arbiter, "UbfArbiter")
EMPTY_COPYDATA(Arbiter)
EMPTY_SERIALIZER(Arbiter)

//-----------------------------------------------------------------------------
// slot table for this class type
//-----------------------------------------------------------------------------
BEGIN_SLOTTABLE(Arbiter)
   "behaviors"                // 1) behaviors
END_SLOTTABLE(Arbiter)

// mapping of slots to handles
BEGIN_SLOT_MAP(Arbiter)
```

```
   ON_SLOT(1, setSlotBehaviors, Basic::PairStream)
END_SLOT_MAP()



//-----------------------------------------------------------------------------
// Class support functions
//-----------------------------------------------------------------------------
Arbiter::Arbiter()
{
   STANDARD_CONSTRUCTOR()
   behaviors = new Basic::List();
}

void Arbiter::deleteData()
{
   // unref behaviors
   if ( behaviors!=0 ) { behaviors->unref(); behaviors = 0; }
}

//-----------------------------------------------------------------------------
// genAction() - generate an action
//-----------------------------------------------------------------------------
Action* Arbiter::genAction(const State* const state, const LCreal dt)
{
   // create list for action set
   Basic::List* actionSet = new Basic::List();

   // fill out list of recommended actions by behaviors
```

```
  Basic::List::Item* item = behaviors->getFirstItem();
  while (item != 0) {
    // get a behavior
    Behavior* behavior = static_cast<Behavior*>(item->getValue());
    // generate action, we have reference
    Action* action = behavior->genAction(state, dt);
    if (action != 0) {
      // add to action set
      actionSet->addTail(action);
      // unref our action reference
      action->unref();
    }
    // goto behavior
    item = item->getNext();
  }

  // given the set of recommended actions, the arbiter
  // decides what action to take
  Action* complexAction = genComplexAction(actionSet);

  // done with action set
  actionSet->unref();

  // return action to perform
  return complexAction;
}
```

```
//-----------------------------------------------------------------------------
// Default: select the action with the highest vote
//-----------------------------------------------------------------------------
Action* Arbiter::genComplexAction(Basic::List* const actionSet)
{
   Action* complexAction = 0;
   unsigned int maxVote = 0;

   // process entire action set
   Basic::List::Item* item = actionSet->getFirstItem();
   while (item != 0) {

      // Is this action's vote higher than the previous?
      Action* action = dynamic_cast<Action*>(item->getValue());
      if (maxVote==0 || action->getVote() > maxVote) {

         // Yes ...
         if (complexAction != 0) complexAction->unref();
         complexAction = action;
         complexAction->ref();
         maxVote = action->getVote();
      }

      // next action
      item = item->getNext();
   }

   if (maxVote > 0 && isMessageEnabled(MSG_DEBUG))
```

```
      std::cout << "Arbiter: chose action with vote= " << maxVote <<
          std::endl;

   // Use our vote value; if its been set
   if (getVote() > 0 && complexAction != 0) {
      complexAction->setVote(getVote());
   }

   // complexAction will have the vote value of whichever component action
       was selected
   return complexAction;
}




//-----------------------------------------------------------------------------
// addBehavior() - add a new behavior
//-----------------------------------------------------------------------------
void Arbiter::addBehavior(Behavior* const x)
{
   behaviors->addTail(x);
   x->container(this);
}



//-----------------------------------------------------------------------------
// Slot functions
//-----------------------------------------------------------------------------
```

```
bool Arbiter::setSlotBehaviors(Basic::PairStream* const x)
{
  bool ok = true;

  // First, make sure they are all behaviors
  {
    Basic::List::Item* item = x->getFirstItem();
    while (item != 0 && ok) {
      Basic::Pair* pair = static_cast<Basic::Pair*>(item->getValue());
      item = item->getNext();
      Behavior* b = dynamic_cast<Behavior*>( pair->object() );
      if (b == 0) {
        // Item is NOT a behavior
        std::cerr << "setSlotBehaviors: slot: " << *pair->slot() << " is
           NOT of a Behavior type!" << std::endl;
        ok = false;
      }
    }
  }

  // next, add behaviors to our list
  if (ok) {
    Basic::List::Item* item = x->getFirstItem();
    while (item != 0) {
      Basic::Pair* pair = static_cast<Basic::Pair*>(item->getValue());
      item = item->getNext();
      Behavior* b = static_cast<Behavior*>(pair->object());
```

```
        addBehavior(b);
      }
   }

   return ok;
}

//-----------------------------------------------------------------------------
// getSlotByIndex()
//-----------------------------------------------------------------------------
Basic::Object* Arbiter::getSlotByIndex(const int si)
{
   return BaseClass::getSlotByIndex(si);
}

} // End Ubf namespace
} // End Basic namespace
} // End Eaagles namespace
```

A.1.7 Behavior.h.

```
//-----------------------------------------------------------------------------
// Class: Behavior
//-----------------------------------------------------------------------------
#ifndef __Eaagles_Basic_Ubf_Behavior_H__
#define __Eaagles_Basic_Ubf_Behavior_H__

#include "openeaagles/basic/Component.h"

namespace Eaagles {
namespace Basic {
namespace Ubf {

class State;
class Action;

//-----------------------------------------------------------------------------
// Class: Behavior
// Description: Abstract base class for all behaviors. Generates an
   optional
//            action based on our current state.
//
// Factory name: UbfBehavior
// Slots:
//   vote    <Number>   ! default vote/weight value for actions generated
//                      ! by this behavior
//-----------------------------------------------------------------------------
class Behavior : public Basic::Component
```

```
{
   DECLARE_SUBCLASS(Behavior, Basic::Component)

public:
   Behavior();

   // Returns a pre-ref'd Action (or zero if no action is generated)
   virtual Action* genAction(const State* const state, const LCreal dt) =
       0;

protected:
   unsigned int getVote() const;
   virtual void setVote(const unsigned int x);

   bool setSlotVote(const Basic::Number* const num);

private:
   unsigned int vote;
};

inline void Behavior::setVote(const unsigned int x) { vote = x; }
inline unsigned int Behavior::getVote() const  { return vote; }


} // End Ubf namespace
} // End Basic namespace
} // End Eaagles namespace
```

```
#endif
```

A.1.8 Behavior.cpp.

```
//-----------------------------------------------------------------------------
// Class: Behavior
//-----------------------------------------------------------------------------

#include "openeaagles/basic/ubf/Behavior.h"

#include "openeaagles/basic/Number.h"

namespace Eaagles {
namespace Basic {
namespace Ubf {

IMPLEMENT_ABSTRACT_SUBCLASS(Behavior, "UbfBehavior")
EMPTY_DELETEDATA(Behavior)
EMPTY_COPYDATA(Behavior)
EMPTY_SERIALIZER(Behavior)

//-----------------------------------------------------------------------------
// slot table for this class type
//-----------------------------------------------------------------------------
BEGIN_SLOTTABLE(Behavior)
   "vote"          //  1) default vote/weight value for actions generated by
      this behavior
END_SLOTTABLE(Behavior)

//  mapping of slots to handles
BEGIN_SLOT_MAP(Behavior)
```

```
   ON_SLOT(1, setSlotVote, Basic::Number)
END_SLOT_MAP()



//-----------------------------------------------------------------------------
// Class support functions
//-----------------------------------------------------------------------------
Behavior::Behavior()
{
   STANDARD_CONSTRUCTOR()
   vote = 0;
}



//-----------------------------------------------------------------------------
// Slot functions
//-----------------------------------------------------------------------------

// [ 1 .. 65535 ]
bool Behavior::setSlotVote(const Basic::Number* const num)
{
   bool ok = false;
   int vote = num->getInt();
   if (vote > 0 && vote <= 65535) {
      setVote(static_cast<unsigned int>(vote));
      ok = true;
   }
   return ok;
```

```
}



//-----------------------------------------------------------------------------
// getSlotByIndex()
//-----------------------------------------------------------------------------
Basic::Object* Behavior::getSlotByIndex(const int si)
{
   return BaseClass::getSlotByIndex(si);
}


} // End Ubf namespace
} // End Basic namespace
} // End Eaagles namespace
```

A.1.9 State.h.

```
//-----------------------------------------------------------------------------
// Class: State
//-----------------------------------------------------------------------------
#ifndef __Eaagles_Basic_Ubf_State_H__
#define __Eaagles_Basic_Ubf_State_H__

#include "openeaagles/basic/Component.h"

namespace Eaagles {
namespace Basic {
namespace Ubf {

//-----------------------------------------------------------------------------
// Class: State
//
// Description: The actor's state vector, as seen by the Behaviors.
//
// Factory name: UbfState
//-----------------------------------------------------------------------------
class State : public Basic::Component
{
   DECLARE_SUBCLASS(State, Basic::Component)
public:
   State();

   virtual void updateGlobalState(void);
   virtual void updateState(Basic::Component* actor);
```

```
  virtual const State* getUbfStateByType(const std::type_info& type)
      const;
};

} // End Ubf namespace
} // End Basic namespace
} // End Eaagles namespace

#endif
```

A.1.10 State.cpp.

```
#include "openeaagles/basic/ubf/State.h"

#include "openeaagles/basic/Pair.h"
#include "openeaagles/basic/PairStream.h"

namespace Eaagles {
namespace Basic {
namespace Ubf {

IMPLEMENT_ABSTRACT_SUBCLASS(State, "UbfState")
EMPTY_SLOTTABLE(State)
EMPTY_CONSTRUCTOR(State)
EMPTY_DELETEDATA(State)
EMPTY_COPYDATA(State)
EMPTY_SERIALIZER(State)

void State::updateGlobalState(void)
{
   // Update all my children
   Basic::PairStream* subcomponents = getComponents();
   if (subcomponents != 0) {
      if (isComponentSelected() != 0) {
         // When we've selected only one
         if (getSelectedComponent() != 0) {
            State* state = dynamic_cast<State*>(getSelectedComponent());
            if (state != 0)
```

```
            state->updateGlobalState();
        }
    }
    else {
        // When we should update them all
        Basic::List::Item* item = subcomponents->getFirstItem();
        while (item != 0) {
            Basic::Pair* pair = static_cast<Basic::Pair*>(item->getValue());
            Basic::Component* obj =
                static_cast<Basic::Component*>(pair->object());
            State* state = dynamic_cast<State*>(obj);
            if (state != 0)
                state->updateGlobalState();
            item = item->getNext();
        }
    }
    subcomponents->unref();
    subcomponents = 0;
  }
}

void State::updateState(Basic::Component* actor)
{
  // Update all my children
  Basic::PairStream* subcomponents = getComponents();
  if (subcomponents != 0) {
    if (isComponentSelected() != 0) {
        // When we've selected only one
```

```
        if (getSelectedComponent() != 0) {
          State* state = dynamic_cast<State*>(getSelectedComponent());
          if (state != 0)
            state->updateState(actor);
        }
      }
      else {
        // When we should update them all
        Basic::List::Item* item = subcomponents->getFirstItem();
        while (item != 0) {
          Basic::Pair* pair = static_cast<Basic::Pair*>(item->getValue());
          Basic::Component* obj =
             static_cast<Basic::Component*>(pair->object());
          State* state = dynamic_cast<State*>(obj);
          if (state != 0)
            state->updateState(actor);
          item = item->getNext();
        }
      }
      subcomponents->unref();
      subcomponents = 0;
    }
  }


const State* State::getUbfStateByType(const std::type_info& type) const
{
  const State* p = this;
```

```
  if ( !p->isClassType(type) ) {
    const Basic::Pair* pair = findByType(type);
    if (pair != 0) {
      p = dynamic_cast<const State*>( pair->object() );
    }
  }
  return p;
}



} // End Ubf namespace
} // End Basic namespace
} // End Eaagles namespace
```

A.2 Sweep Mission Scenario Implementation Code

A.2.1 FusionArbiter.h.

```
//-----------------------------------------------------------------------------
// Class: FusionArbiter
//-----------------------------------------------------------------------------
#ifndef __Eaagles_xBehaviors_FusionArbiter_H__
#define __Eaagles_xBehaviors_FusionArbiter_H__

#include "openeaagles/basic/ubf/Arbiter.h"

namespace Eaagles {

namespace Basic { class List; class Action; }

namespace xBehaviors {

//-----------------------------------------------------------------------------
// Class: FusionArbiter
//
// Description: Fusion arbiter for a plane
//-----------------------------------------------------------------------------
class FusionArbiter : public Basic::Ubf::Arbiter
{
  DECLARE_SUBCLASS(FusionArbiter, Basic::Ubf::Arbiter)

public:
  FusionArbiter();
```

```
    // generates an action based upon the recommended actions in the
        actionSet
    virtual Basic::Ubf::Action* genComplexAction(Basic::List* const
        actionSet);

private:

};

}
}

#endif
```

A.2.2 FusionArbiter.cpp.

```
//-----------------------------------------------------------------------------
// Class: FusionArbiter
//-----------------------------------------------------------------------------
#include "FusionArbiter.h"

#include "openeaagles/basic/List.h"

#include "PlaneAction.h"

namespace Eaagles {
namespace xBehaviors {

IMPLEMENT_SUBCLASS(FusionArbiter, "FusionArbiter")
EMPTY_SLOTTABLE(FusionArbiter)
EMPTY_CONSTRUCTOR(FusionArbiter)
EMPTY_COPYDATA(FusionArbiter)
EMPTY_SERIALIZER(FusionArbiter)
EMPTY_DELETEDATA(FusionArbiter)

Basic::Ubf::Action* FusionArbiter::genComplexAction(Basic::List* const
    actionSet)
{
   PlaneAction* complexAction = new PlaneAction;

   unsigned int maxVote = 0;

   // Here's our data used for the fusion part of this complex action
```

```
// which is to average/"fuse" the altitude, heading, and velocity,
// and we'll launch a missile if that behavior recommends a missile
// launch within a certain threshold of the maximum-voted action
double headingSum = 0;
double altitudeSum = 0;
double velocitySum = 0;

double headingCt = 0;
double altitudeCt = 0;
double velocityCt = 0;

bool launchMsl = false;
unsigned int mslVote = 0;

// process entire action set
const Basic::List::Item* item = actionSet->getFirstItem();
while (item != 0) {
  const PlaneAction* action = dynamic_cast<const
      PlaneAction*>(item->getValue());
  if (action!=0) {

    if (action->getVote() > maxVote) {

      complexAction->setAPNavigate(action->getAPNavigate());
      complexAction->setFollow(action->getFollow());
      complexAction->setFollowTarget(action->getFollowTarget());
      complexAction->setReleaseWeapon(action->getReleaseWeapon());
      complexAction->setWeaponTarget(action->getWeaponTarget());
```

```
   complexAction->setFltMember(action->getFltMember());
   complexAction->setVote(action->getVote());

      maxVote = action->getVote();

   setVote(action->getVote());
  }

// Here's our fusion - this happens for every action
//  independent of the vote

// Is any behavior recommending to launch a missile?
if (action->getReleaseWeapon()){
  launchMsl = true;
  mslVote = action->getVote();
}

// Let's get a sum for the heading altitude, and velocity
//  so we can average...
headingSum += complexAction->getHeading();
velocitySum += complexAction->getVelocity();
altitudeSum += complexAction->getAltitude();

// ...plus a count of the number of values we have
//  going into our final average
headingCt++;
altitudeCt++;
velocityCt++;
```

```
    }
    else {
      std::cout << "Action NOT a PlaneAction\n";
    }
    // next action
    item = item->getNext();
  } // end while (iterating through list of acitons)

  // Now we'll set the actions commands averages of the velocity,
     altitude, and heading
  // Note: these will be useless if we're engaging the autopilot at all...
  if (headingCt == 0 || velocityCt == 0 || altitudeCt == 0){
    // we don't want a divide by zero error, so in this case,
    //  we'll just let the maximum voted action decide our
    //  heading
  }
  else {
    complexAction->setHeading( (headingSum / headingCt) );
    complexAction->setVelocity( (velocitySum / velocityCt) );
    complexAction->setAltitude( (altitudeSum / altitudeCt) );
  }

  // Let's say that we should launch a missile if it's recommended
  // We'll define "recommended" as within a certain threshold (defined
  //  above) of the maximum vote
  unsigned int mslVoteThreshold = 20;
```

```
    if ( launchMsl && (mslVote > (maxVote - mslVoteThreshold)) ){
        complexAction->setReleaseWeapon(true);
    }

    std::cout << "FollowTarget: " << (complexAction->getFollowTarget() ?
        "Yes" : "No") << "\n";
    std::cout << "APNavigate: " << (complexAction->getAPNavigate() ? "Yes"
        : "No") << "\n";

    return complexAction;
}

}
}
```

A.2.3 PlaneAction.h.

```
//-----------------------------------------------------------------------------
// Class: PlaneAction
//-----------------------------------------------------------------------------

#ifndef __Eaagles_xBehaviors_PlaneAction_H__
#define __Eaagles_xBehaviors_PlaneAction_H__

#include "openeaagles/basic/ubf/Action.h"

namespace Eaagles {

namespace Simulation { class Player; }

namespace xBehaviors {

//-----------------------------------------------------------------------------
// Class: PlaneAction
//-----------------------------------------------------------------------------
class PlaneAction : public Basic::Ubf::Action
{
   DECLARE_SUBCLASS(PlaneAction, Basic::Ubf::Action)

public:
   PlaneAction();

   virtual bool execute(Basic::Component* actor);
```

```
// get/set methods
void setHeading(const double);
double getHeading() const      { return heading;        }

void setVelocity(const double);
double getVelocity() const       { return velocity;         }

void setAltitude(const double);
double getAltitude() const       { return altitude;         }

void setAPNavigate(const bool);
bool getAPNavigate() const       { return apNavigate;       }

void setFollow(const bool);
bool getFollow() const        { return follow;        }

void setFollowTarget(const bool);
bool getFollowTarget() const       { return followTarget;       }

void setReleaseWeapon(const bool);
bool getReleaseWeapon() const       { return releaseWeapon;       }

void setWeaponTarget(const int);
int getWeaponTarget() const       { return weaponTarget;       }

void setFltMember(const unsigned int);
unsigned int getFltMember() const      { return fltMember;        }
```

```
private:

   double heading;

   double velocity;

   double altitude;

   bool apNavigate;

   bool follow;
   bool followTarget;

   bool releaseWeapon;
   int weaponTarget;

   unsigned int fltMember;

};

}
}

#endif
```

A.2.4 PlaneAction.cpp.

```
//-----------------------------------------------------------------------------
// Class: PlaneAction
//-----------------------------------------------------------------------------

#include "PlaneAction.h"
#include "PlaneState.h"

#include "openeaagles/basic/PairStream.h"
#include "openeaagles/simulation/AirVehicle.h"
#include "openeaagles/simulation/DynamicsModels.h"
#include "openeaagles/simulation/Autopilot.h"
#include "openeaagles/simulation/OnboardComputer.h"
#include "openeaagles/simulation/TrackManager.h"
#include "openeaagles/simulation/Track.h"
#include "openeaagles/simulation/StoresMgr.h"
#include "openeaagles/simulation/Missile.h"
#include "openeaagles/simulation/Player.h"
#include "openeaagles/simulation/Simulation.h"

namespace Eaagles {
namespace xBehaviors {

IMPLEMENT_SUBCLASS(PlaneAction, "PlaneAction")
EMPTY_SLOTTABLE(PlaneAction)
EMPTY_DELETEDATA(PlaneAction)
EMPTY_SERIALIZER(PlaneAction)
```

```
PlaneAction::PlaneAction()
{
   STANDARD_CONSTRUCTOR()

   heading = 0;
   velocity = 0;
   altitude = 0;

   apNavigate = false;

   follow = false;

   releaseWeapon = false;

   weaponTarget = PlaneState::MAX_TRACKS;

   followTarget = false;

   fltMember = 0;
}

void PlaneAction::copyData(const PlaneAction& org, const bool cc)
{
   BaseClass::copyData(org);
   heading = org.heading;

   velocity = org.velocity;
```

```
  altitude = org.altitude;

  apNavigate = org.apNavigate;

  follow = org.follow;

  releaseWeapon = org.releaseWeapon;

  weaponTarget = org.weaponTarget;

  followTarget = org.followTarget;

  fltMember = org.fltMember;
}

void PlaneAction::setHeading(const double x)
{
  heading = x;
}

void PlaneAction::setVelocity(const double x)
{
  velocity = x;
}

void PlaneAction::setAltitude(const double x)
{
  altitude = x;
```

```
}

void PlaneAction::setAPNavigate(const bool x)
{
   apNavigate = x;
}

void PlaneAction::setFollow(const bool x)
{
   follow = x;
}

void PlaneAction::setFollowTarget(const bool x)
{
   followTarget = x;
}

void PlaneAction::setReleaseWeapon(const bool x)
{
   releaseWeapon = x;
}

void PlaneAction::setWeaponTarget(const int x)
{
   weaponTarget = x;
}

void PlaneAction::setFltMember(const unsigned int x)
```

```
{
   fltMember = x;
}

bool PlaneAction::execute(Basic::Component* actor)
{
   Simulation::Player* player = dynamic_cast<Simulation::Player*>( actor );

   if (player != 0) {
      // We want to release our weapon regardless of our navigation...
      if (getReleaseWeapon()){
         // Get our stores management, assuming we have one...
         Simulation::StoresMgr* sm = player->getStoresManagement();
         if (sm != 0){
            if (!(sm->isWeaponReleased())){ // Did we just release a weapon?
               If so, prob a bad idea fo fire again...

               // We need some more stuff here - find the track of the
                  target...
               const Simulation::OnboardComputer* oc =
                  player->getOnboardComputer();
               if (oc != 0) {

               // Get our track manager
               const Simulation::TrackManager* trackManager =
                  oc->getTrackManagerByType(typeid(Simulation::TrackManager));
               if (trackManager != 0) {
                  // Now let's get our track list (all of the radar tracks)
```

```
            SPtr<Simulation::Track> trackList[PlaneState::MAX_TRACKS];
                // MAX_TRACKS is 50
            unsigned int nTracks =
                trackManager->getTrackList(trackList,
                PlaneState::MAX_TRACKS);

            // Find our player
            Simulation::Player* ourTarget =
                dynamic_cast<Simulation::Player*>(trackList[getWeaponTarget()]->getTarge

            // Release our missile
            Simulation::Missile* missile =
                dynamic_cast<Simulation::Missile*>(sm->releaseOneMissile());

            // Make sure our target and our missile are real...
            if (missile != 0 && ourTarget != 0){
              // ... then reset our missile target (for some reason
                  this doesn't occur
              //      when the missile is released) with guidance
                  (second param)
              missile->setTargetPlayer( ourTarget , true );
            }
          }
          } // End if onboard computer
        } // End if weapon released
      } // end if stores manager
    } // end if release weapon
```

```
if (getAPNavigate()) {
   // Get our autopilot and turn on navigation
   Simulation::Autopilot* ap = dynamic_cast<Simulation::Autopilot*>(
       player->getPilot() );
   if ( (ap != 0) ){
      ap->setFollowTheLeadMode(false);
      ap->setNavMode(true);;
   }
}
else if (getFollow()){
   // Get our autopilot and turn on navigation
   Simulation::Autopilot* ap = dynamic_cast<Simulation::Autopilot*>(
       player->getPilot() );
   if ( (ap != 0) ){
   // We're following a flight lead (or we are flight lead, but
       hopefully we
         // won't reach this point if that's the case)

      // So we want our wall formation:
      //
      //                      1  (flight lead)
      //   2 |-- 6 to 9 kft --| |---- 9 to 12 kft ----| 3 (element
         lead)
      //                                              |-- 6 to 9 kft
         --| 4
      //
```

```
//   *Angles should be no more than 15 degrees "behind"
    respective element lead
//   *Vertical spacing is environment-dependent

// Ok, here's where we figure out who we are, with the
    assumption that
//  everyone in our flight:
//   (a) has a player name of the form: flightname#
//     where flightname is the flight's name (e.g., "Eagle")
//     and # is the number of the flight member IAW the diagram
//     above.
//   (b) the flight member number # is no more than one digit.

Basic::String pname( (player->getName())->getString() ); // Our
    player's name

// Get the flight name, one less char than full player name
Basic::String flightNm;
pname.getSubString(flightNm, 0, (pname.len() - 1) );


if (getFltMember() == 2){ // Follow flight lead (1) trailing left
   Basic::String leader(flightNm, "1");

   // All function params are in meters, so we need to convert
   ap->setLeadFollowingDistanceTrail( (500 * Basic::Feet::FT2M)
       ); // Follow 500 feet behind
```

```
    ap->setLeadFollowingDeltaAltitude( (-1000 *
        Basic::Feet::FT2M) ); // 1000 feet below altitude
    ap->setLeadFollowingDistanceRight( (-6000 *
        Basic::Feet::FT2M) ); // Follow 6000 feet to the left

    ap->setLeadPlayerName(leader);
}
else if (getFltMember() == 3) { // Follow flight lead (1)
     trailing right
    Basic::String leader(flightNm, "1");

    // All function params are in meters, so we need to convert
    ap->setLeadFollowingDistanceTrail( (500 * Basic::Feet::FT2M)
        ); // Follow 500 ft behind
    ap->setLeadFollowingDeltaAltitude( (-1000 *
        Basic::Feet::FT2M) ); // 1000 feet below altitude
    ap->setLeadFollowingDistanceRight( (9000 * Basic::Feet::FT2M)
        ); // Follow 9000 feet to the right

    ap->setLeadPlayerName(leader);
}
else if (getFltMember() == 4){ // Follow element lead (3)
     trailing right
    Basic::String leader(flightNm, "3");

    // All function params are in meters, so we need to convert
    ap->setLeadFollowingDistanceTrail( (500 * Basic::Feet::FT2M)
        ); // Follow 500 ft behind
```

```
        ap->setLeadFollowingDeltaAltitude( (-1000 *
            Basic::Feet::FT2M) ); // 1000 feet below altitude
        ap->setLeadFollowingDistanceRight( (6000 * Basic::Feet::FT2M)
            ); // Follow 6000 feet to the right

        ap->setLeadPlayerName(leader);
      }
      // Otherwise we'll assume the defaults are good enough...
    }

    // Now we can turn on autopilot
    ap->setNavMode(true);

    ap->setFollowTheLeadMode(true);

  }
  else if (getFollowTarget()){ // We're following our target
    // We need some more stuff here - find the track of the target...
    Simulation::Autopilot* ap = dynamic_cast<Simulation::Autopilot*>(
        player->getPilot() );
    if ( ap != 0 ){
      const Simulation::OnboardComputer* oc =
          player->getOnboardComputer();
      if (oc != 0) {
        // Get our track manager
        const Simulation::TrackManager* trackManager =
            oc->getTrackManagerByType(typeid(Simulation::TrackManager));
        if (trackManager != 0) {
```

```
        // Now let's get our track list (all of the radar tracks)
        SPtr<Simulation::Track> trackList[PlaneState::MAX_TRACKS];
            // MAX_TRACKS is 50
        unsigned int nTracks =
            trackManager->getTrackList(trackList,
            PlaneState::MAX_TRACKS);

        if (trackList[getWeaponTarget()] != 0){
           Simulation::Player* ourTarget =
               dynamic_cast<Simulation::Player*>(trackList[getWeaponTarget()]->getTar
           if (ourTarget != 0) {
              ap->setLeadPlayer(ourTarget); // Set our target as the
           }
        }
        else {
           // why did we get here?
           std::cout << "no target\n\n";
        }

        ap->setLeadFollowingDistanceTrail( (2 *
            Basic::Distance::NM2M ) ); // Follow behind
        ap->setLeadFollowingDeltaAltitude(0); // Aim for the same
            altitude (o meters difference)
        ap->setLeadFollowingDistanceRight(0); // Follow directly
            behind (0 meters difference)
     }
  } // End if onboard computer
```

```
        // Now we can turn on autopilot
        ap->setNavMode(true);

        std::cout << "\n\nPlayer: " << player->getName()->getString() <<
            ", damage: " << player->getDamage() << "\n\n";

        ap->setFollowTheLeadMode(true);

    }
}
else { // Don't need autopilot, we're controlling "manually" (not
    following or using waypoints)
    Simulation::Autopilot* ap = dynamic_cast<Simulation::Autopilot*>(
        player->getPilot() );
    if ( (ap != 0) ){ // Gotta turn everything off...we're doing
        manual control
        // This will be on if we were navigating...
        ap->setNavMode(false);

        // ...and so will these, because nav mode turns them on
        ap->setAltitudeHoldMode(false);
        ap->setVelocityHoldMode(false);
        ap->setHeadingHoldMode(false);

        // This will be on if we were following...
        ap->setFollowTheLeadMode(false);
```

```
            // We don't use this yet, but we'll turn it off just in case
            ap->setLoiterMode(false);
        }
        Simulation::DynamicsModel* model =
            dynamic_cast<Simulation::DynamicsModel*>
            (player->getDynamicsModel());

        if (model != 0) {
            // Command our "vector" of heading, velocity, and altitude
              model->setCommandedHeadingD(heading, 25, 90); // 25 degrees
                  per sec is about halfway between the capabilities of the
                  F-15 and F-22
            model->setCommandedVelocityKts(velocity);
            model->setCommandedAltitude((altitude * Basic::Distance::FT2M),
                (500 * Basic::Distance::FT2M), 90);
        }
      } // end if-else navigate, follow, follow target, or not

      return true;
    } // end if player != 0

    return false;
  }

  }
  }
```

A.2.5 PlaneBehaviors.h.

```
//-----------------------------------------------------------------------------
// Classes: PlaneBehaviorBase
//          PlaneBehaviorBase -> PlaneFlyStraight
//          PlaneBehaviorBase -> PlaneTurnRight
//-----------------------------------------------------------------------------
#ifndef __Eaagles_xBehaviors_PlaneBehaviors_H__
#define __Eaagles_xBehaviors_PlaneBehaviors_H__

#include "openeaagles/basic/ubf/Behavior.h"

namespace Eaagles {

namespace Basic { class Distance; class State; }

namespace xBehaviors {

//-----------------------------------------------------------------------------
// test code for a base class for PlaneBehaviors, implements some common
   slots
//-----------------------------------------------------------------------------
class PlaneBehaviorBase : public Basic::Ubf::Behavior
{
  DECLARE_SUBCLASS(PlaneBehaviorBase, Basic::Ubf::Behavior)
public:
   PlaneBehaviorBase();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
      const LCreal dt)=0;
```

```
protected:
   bool setSlotCriticalAltitude(const Basic::Distance* const msg);
   bool setSlotVoteOnCriticalAltitude(const Basic::Number* const num);
   bool setSlotVoteOnIncomingMissile(const Basic::Number* const num);

   unsigned int voteOnIncomingMissile;
   unsigned int voteOnCriticalAltitude;
   LCreal criticalAltitude;
};



class PlaneFlyStraight : public PlaneBehaviorBase
{
   DECLARE_SUBCLASS(PlaneFlyStraight, PlaneBehaviorBase)
public:
   PlaneFlyStraight();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
       const LCreal dt);
};

class PlaneTurnRight : public PlaneBehaviorBase
{
   DECLARE_SUBCLASS(PlaneTurnRight, PlaneBehaviorBase)
public:
   PlaneTurnRight();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
       const LCreal dt);
};
```

```
class PlaneTurnLeft : public PlaneBehaviorBase
{
   DECLARE_SUBCLASS(PlaneTurnLeft, PlaneBehaviorBase)
public:
   PlaneTurnLeft();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
       const LCreal dt);
};

class PlaneClimb : public PlaneBehaviorBase
{
   DECLARE_SUBCLASS(PlaneClimb, PlaneBehaviorBase)
public:
   PlaneClimb();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
       const LCreal dt);
};

class PlaneDive : public PlaneBehaviorBase
{
   DECLARE_SUBCLASS(PlaneDive, PlaneBehaviorBase)
public:
   PlaneDive();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
       const LCreal dt);
};
```

```
class PlaneNavigate : public PlaneBehaviorBase
{
   DECLARE_SUBCLASS(PlaneNavigate, PlaneBehaviorBase)
public:
   PlaneNavigate();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
       const LCreal dt);
};

class PlaneFollow : public PlaneBehaviorBase
{
   DECLARE_SUBCLASS(PlaneFollow, PlaneBehaviorBase)
public:
   PlaneFollow();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
       const LCreal dt);
};


class PlaneReleaseWeapon : public PlaneBehaviorBase
{
   DECLARE_SUBCLASS(PlaneReleaseWeapon, PlaneBehaviorBase)
public:
   PlaneReleaseWeapon();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
       const LCreal dt);
};
```

```
class PlaneFollowEnemy : public PlaneBehaviorBase
{
   DECLARE_SUBCLASS(PlaneFollowEnemy, PlaneBehaviorBase)
public:
   PlaneFollowEnemy();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
       const LCreal dt);
};



class PlaneBreakDefend : public PlaneBehaviorBase
{
   DECLARE_SUBCLASS(PlaneBreakDefend, PlaneBehaviorBase)
public:
   PlaneBreakDefend();
   Basic::Ubf::Action* genAction(const Basic::Ubf::State* const state,
       const LCreal dt);
private:
   double startHeading;
};

}
}

#endif
```

A.2.6 PlaneBehaviors.cpp.

```
//-----------------------------------------------------------------------------
// Classes: PlaneBehaviorBase
//          PlaneBehaviorBase -> PlaneFire
//          PlaneBehaviorBase -> PlaneFlyStraight
//          PlaneBehaviorBase -> PlaneFollowEnemy
//          PlaneBehaviorBase -> PlaneTurn
//          PlaneBehaviorBase -> PlaneSlowTurn
//          PlaneBehaviorBase -> PlaneClimb
//          PlaneBehaviorBase -> PlaneDive
//          PlaneBehaviorBase -> PlaneTrim
//          PlaneBehaviorBase -> PlaneRoll
//          PlaneBehaviorBase -> PlaneBarrelRoll
//          PlaneBehaviorBase -> PlaneLoop
//-----------------------------------------------------------------------------
#include "PlaneBehaviors.h"
#include "PlaneAction.h"
#include "PlaneState.h"

#include "openeaagles/basic/units/Distances.h"
#include "openeaagles/basic/units/Angles.h"
#include "openeaagles/basic/ubf/State.h"

namespace Eaagles {
namespace xBehaviors {

IMPLEMENT_ABSTRACT_SUBCLASS(PlaneBehaviorBase, "PlaneBehaviorBase")
EMPTY_COPYDATA(PlaneBehaviorBase)
```

```
EMPTY_SERIALIZER(PlaneBehaviorBase)
EMPTY_DELETEDATA(PlaneBehaviorBase)

// slot table for this class type
BEGIN_SLOTTABLE(PlaneBehaviorBase)
    "criticalAltitude",
    "voteOnCriticalAltitude",
    "voteOnIncomingMissile"
END_SLOTTABLE(PlaneBehaviorBase)

// map slot table to handles
BEGIN_SLOT_MAP(PlaneBehaviorBase)
   ON_SLOT( 1, setSlotCriticalAltitude, Basic::Distance )
   ON_SLOT( 2, setSlotVoteOnCriticalAltitude, Basic::Number)
   ON_SLOT( 3, setSlotVoteOnIncomingMissile, Basic::Number)
END_SLOT_MAP()


PlaneBehaviorBase::PlaneBehaviorBase()
{
   STANDARD_CONSTRUCTOR()
   criticalAltitude = 3500.0f;
   voteOnCriticalAltitude = 0;
   voteOnIncomingMissile = 0;
}

bool PlaneBehaviorBase::setSlotCriticalAltitude(const Basic::Distance*
    const msg)
```

```
{
    bool ok = false;
    if (msg != 0) {
       double value = Basic::Meters::convertStatic( *msg );
       criticalAltitude = value;
       ok = true;
    }
    return ok;
}

// [ 1 .. 65535 ]
bool PlaneBehaviorBase::setSlotVoteOnCriticalAltitude(const Basic::Number*
    const num)
{
   bool ok = false;
   int vote = num->getInt();
   if (vote > 0 && vote <= 65535) {
      voteOnCriticalAltitude = static_cast<unsigned int>(vote);
      ok = true;
   }
   return ok;
}

// [ 1 .. 65535 ]
bool PlaneBehaviorBase::setSlotVoteOnIncomingMissile(const Basic::Number*
    const num)
{
   bool ok = false;
```

```
   int vote = num->getInt();
   if (vote > 0 && vote <= 65535) {
      voteOnIncomingMissile = static_cast<unsigned int>(vote);
      ok = true;
   }
   return ok;
}

//-----------------------------------------------------------------------------
// getSlotByIndex() for Graphic
//-----------------------------------------------------------------------------
Basic::Object* PlaneBehaviorBase::getSlotByIndex(const int si)
{
    return BaseClass::getSlotByIndex(si);
}

IMPLEMENT_SUBCLASS(PlaneFlyStraight, "PlaneFlyStraight")
EMPTY_SLOTTABLE(PlaneFlyStraight)
EMPTY_COPYDATA(PlaneFlyStraight)
EMPTY_SERIALIZER(PlaneFlyStraight)
EMPTY_DELETEDATA(PlaneFlyStraight)

PlaneFlyStraight::PlaneFlyStraight()
{
   STANDARD_CONSTRUCTOR()
}
```

```
Basic::Ubf::Action* PlaneFlyStraight::genAction(const Basic::Ubf::State*
    const state, const LCreal dt)
{
   PlaneAction* action = 0;
   const PlaneState* pState = dynamic_cast<const
       PlaneState*>(state->getUbfStateByType(typeid(PlaneState)));

   if (pState!=0 && pState->isAlive()) {
      action = new PlaneAction();

     if ( pState->isAPNavigating() ){ action->setAPNavigate(false); }

      action->setHeading(pState->getHeading());

     action->setVelocity(pState->getVelocity());

     action->setAltitude(pState->getAltitude());

     action->setVote(getVote());
   }
   return action;
}


IMPLEMENT_SUBCLASS(PlaneTurnRight, "PlaneTurnRight")
EMPTY_SLOTTABLE(PlaneTurnRight)
EMPTY_COPYDATA(PlaneTurnRight)
EMPTY_SERIALIZER(PlaneTurnRight)
```

```
EMPTY_DELETEDATA(PlaneTurnRight)

PlaneTurnRight::PlaneTurnRight()
{
   STANDARD_CONSTRUCTOR()
}

Basic::Ubf::Action* PlaneTurnRight::genAction(const Basic::Ubf::State*
    const state, const LCreal dt)
{
   PlaneAction* action = 0;
   const PlaneState* pState = dynamic_cast<const
       PlaneState*>(state->getUbfStateByType(typeid(PlaneState)));

   if (pState!=0 && pState->isAlive()) {
      action = new PlaneAction();

     if ( pState->isAPNavigating() ){ action->setAPNavigate(false); }

      action->setHeading(pState->getHeading() + 25); // Increase degrees
          for right turn

     action->setVelocity(pState->getVelocity());

     action->setAltitude(pState->getAltitude());

     action->setVote(getVote());
   }
```

```
   return action;
}



IMPLEMENT_SUBCLASS(PlaneTurnLeft, "PlaneTurnLeft")
EMPTY_SLOTTABLE(PlaneTurnLeft)
EMPTY_COPYDATA(PlaneTurnLeft)
EMPTY_SERIALIZER(PlaneTurnLeft)
EMPTY_DELETEDATA(PlaneTurnLeft)

PlaneTurnLeft::PlaneTurnLeft()
{
   STANDARD_CONSTRUCTOR()
}

Basic::Ubf::Action* PlaneTurnLeft::genAction(const Basic::Ubf::State*
    const state, const LCreal dt)
{
   PlaneAction* action = 0;
   const PlaneState* pState = dynamic_cast<const
       PlaneState*>(state->getUbfStateByType(typeid(PlaneState)));

   if (pState!=0 && pState->isAlive()) {
      action = new PlaneAction();

     if ( pState->isAPNavigating() ){ action->setAPNavigate(false); }
```

```
     action->setHeading(pState->getHeading() - 25); // Decrease degrees
         for left turn

    action->setVelocity(pState->getVelocity());

    action->setAltitude(pState->getAltitude());

    action->setVote(getVote());
  }
  return action;
}



IMPLEMENT_SUBCLASS(PlaneClimb, "PlaneClimb")
EMPTY_SLOTTABLE(PlaneClimb)
EMPTY_COPYDATA(PlaneClimb)
EMPTY_SERIALIZER(PlaneClimb)
EMPTY_DELETEDATA(PlaneClimb)

PlaneClimb::PlaneClimb()
{
  STANDARD_CONSTRUCTOR()
}

Basic::Ubf::Action* PlaneClimb::genAction(const Basic::Ubf::State* const
    state, const LCreal dt)
{
  PlaneAction* action = 0;
```

```
  const PlaneState* pState = dynamic_cast<const
      PlaneState*>(state->getUbfStateByType(typeid(PlaneState)));

  if (pState!=0 && pState->isAlive()) {
     action = new PlaneAction();

    if ( pState->isAPNavigating() ){ action->setAPNavigate(false); }

     action->setHeading(pState->getHeading());

    action->setVelocity(pState->getVelocity());

    action->setAltitude(pState->getAltitude() + 500); // Increase altitude

    action->setVote(getVote());
  }
  return action;
}



IMPLEMENT_SUBCLASS(PlaneDive, "PlaneDive")
EMPTY_SLOTTABLE(PlaneDive)
EMPTY_COPYDATA(PlaneDive)
EMPTY_SERIALIZER(PlaneDive)
EMPTY_DELETEDATA(PlaneDive)

PlaneDive::PlaneDive()
{
```

```
   STANDARD_CONSTRUCTOR()
}

Basic::Ubf::Action* PlaneDive::genAction(const Basic::Ubf::State* const
    state, const LCreal dt)
{
   PlaneAction* action = 0;
   const PlaneState* pState = dynamic_cast<const
       PlaneState*>(state->getUbfStateByType(typeid(PlaneState)));

   if (pState!=0 && pState->isAlive()) {
      action = new PlaneAction();

     action->setAPNavigate(false);
     action->setFollow(false);
     action->setFollowTarget(false);

      action->setHeading(pState->getHeading());

     action->setVelocity(pState->getVelocity());

     action->setAltitude(pState->getAltitude() - 500); // Decrease altitude

     action->setVote(getVote());
   }
   return action;
}
```

```
IMPLEMENT_SUBCLASS(PlaneFollow, "PlaneFollow")
EMPTY_SLOTTABLE(PlaneFollow)
EMPTY_COPYDATA(PlaneFollow)
EMPTY_SERIALIZER(PlaneFollow)
EMPTY_DELETEDATA(PlaneFollow)

PlaneFollow::PlaneFollow()
{
   STANDARD_CONSTRUCTOR()
}

Basic::Ubf::Action* PlaneFollow::genAction(const Basic::Ubf::State* const
    state, const LCreal dt)
{
   PlaneAction* action = 0;
   const PlaneState* pState = dynamic_cast<const
       PlaneState*>(state->getUbfStateByType(typeid(PlaneState)));

   if (pState!=0 && pState->isAlive()) {
      action = new PlaneAction();



     if ( (pState->isLead()) ){
      action->setAPNavigate(false);
      action->setFollow(false);
      action->setFollowTarget(false);
```

```
      action->setVote( 1 );
     }
     else {
      action->setAPNavigate(false);
      action->setFollow(true);
      action->setFollowTarget(false);

      action->setVote( getVote() );
     }

     action->setFltMember( pState->getFlightMember() );

     std::cout << "\nPlaneFollow, vote: " << action->getVote() << "\n";
   }

   return action;
}



IMPLEMENT_SUBCLASS(PlaneNavigate, "PlaneNavigate")
EMPTY_SLOTTABLE(PlaneNavigate)
EMPTY_COPYDATA(PlaneNavigate)
EMPTY_SERIALIZER(PlaneNavigate)
EMPTY_DELETEDATA(PlaneNavigate)

PlaneNavigate::PlaneNavigate()
{
   STANDARD_CONSTRUCTOR()
```

```
}

Basic::Ubf::Action* PlaneNavigate::genAction(const Basic::Ubf::State*
    const state, const LCreal dt)
{
   PlaneAction* action = 0;
   const PlaneState* pState = dynamic_cast<const
       PlaneState*>(state->getUbfStateByType(typeid(PlaneState)));

   if (pState!=0 && pState->isAlive()) {
      action = new PlaneAction();

     if ( (pState->isLead()) ) {
        action->setAPNavigate(true);
        action->setFollow(false);
        action->setFollowTarget(false);

        action->setVote( getVote() );
     }
     else {
        action->setAPNavigate(false);
        action->setFollow(false);
        action->setFollowTarget(false);

        // Just go straight
        action->setAltitude(pState->getAltitude());
        action->setHeading(pState->getHeading());
        action->setVelocity(pState->getVelocity());
```

```
        action->setVote( 1 );
      }

      action->setFltMember( pState->getFlightMember() );

      std::cout << "PlaneNavigate, vote: " << action->getVote() << "\n";
    }
    return action;
}



IMPLEMENT_SUBCLASS(PlaneReleaseWeapon, "PlaneReleaseWeapon")
EMPTY_SLOTTABLE(PlaneReleaseWeapon)
EMPTY_COPYDATA(PlaneReleaseWeapon)
EMPTY_SERIALIZER(PlaneReleaseWeapon)
EMPTY_DELETEDATA(PlaneReleaseWeapon)

PlaneReleaseWeapon::PlaneReleaseWeapon()
{
    STANDARD_CONSTRUCTOR()
}

Basic::Ubf::Action* PlaneReleaseWeapon::genAction(const Basic::Ubf::State*
     const state, const LCreal dt)
{
    PlaneAction* action = 0;
```

```
const PlaneState* pState = dynamic_cast<const
    PlaneState*>(state->getUbfStateByType(typeid(PlaneState)));

if (pState!=0 && pState->isAlive()) {
   action = new PlaneAction();

  // Maintain heading/velocity/altitude...
  action->setAPNavigate(false);
  action->setFollow(false);
  action->setFollowTarget(false);



  action->setHeading(pState->getHeading());
   action->setVelocity(pState->getVelocity());
  action->setAltitude(pState->getAltitude());

  action->setReleaseWeapon(false); // We'll set it false upfront, then
      do some tests to see if it should really be true...

  // Now for launching the missile...
  if (pState->getTargetTrack() != PlaneState::MAX_TRACKS){ // We have a
      target that we're tracking
     // Let's set our target...
     action->setWeaponTarget(pState->getTargetTrack());

     // Now let's determine if target is in our WEZ (weapons employment
        zone):
```

```
// What is our WEZ? It would be aircraft (and potentially
    adversary)-dependent
// A real WEZ is teardrop shaped off the nose of our aircraft, but
    for simplicity,
//  we're going to use a 3 dimensional "windshield"ish-shaped WEZ:
//    For our case, we will be using missiles with a maximum burst
    range of 50 nm,
//     so let's make our max WEZ range 45 nm, our min WEZ range 1 nm.
//     We want our heading and pitch to the target no more than 20
    degrees off our own nose
if (  pState->getDistanceToTracked(pState->getTargetTrack()) < (45
    * Basic::NauticalMiles::NM2M)
   && pState->getDistanceToTracked(pState->getTargetTrack()) > (0.5
       * Basic::NauticalMiles::NM2M) // Distance test

   && pState->getHeadingToTracked(pState->getTargetTrack()) <= (10)
   && pState->getHeadingToTracked(pState->getTargetTrack()) >= (-10)
       // Heading test

   && pState->getPitchToTracked(pState->getTargetTrack()) <= (10)
   && pState->getPitchToTracked(pState->getTargetTrack()) >= (-10)
       // Pitch test

   && pState->getNumMissiles() > 0){ // Make sure we actually have
       some missiles...

        // This means our target is in the WEZ and we have available
            missiles!
```

```
        // Now let's do some math to utilize our timer and incorporate
            our distance from the target
        double distNM = (
            (pState->getDistanceToTracked(pState->getTargetTrack())) *
            Basic::NauticalMiles::M2NM );

        std::cout << "distNM = " << distNM << "\n";

        // If we're closer, we should fire more often...
        // So, let's say we're 50 NM away - this means our timer needs
            to be 50 * 10 = 500 to fire again
        // But, if we're 4 NM away, our timer needs to be 4 * 50 = 200
            to fire again...
        if ( (pState->getMissileTimer() / distNM ) > 15 ) {
           // Our timer is triggered... FIRE!
           action->setReleaseWeapon(true);
        }

        // Here is where we can calculate a probability of kill p_k and
            then possibly use it for our vote
    }
} // End if get target track

// Also should we incorporate the probability of a kill into our
    vote...? Maybe...
//  it would be a good opportunity to demonstrate the usefulness of a
    function-based
```

```
    //  vote
    if ( (action->getReleaseWeapon()) ){
     action->setVote( getVote() );
    }
    else {
     action->setVote( 1 );
    }

    action->setFltMember( pState->getFlightMember() );

    std::cout << "PlaneReleaseWeapon, vote: " << action->getVote() << "\n";
  }
  return action;
}


IMPLEMENT_SUBCLASS(PlaneFollowEnemy, "PlaneFollowEnemy")
EMPTY_SLOTTABLE(PlaneFollowEnemy)
EMPTY_COPYDATA(PlaneFollowEnemy)
EMPTY_SERIALIZER(PlaneFollowEnemy)
EMPTY_DELETEDATA(PlaneFollowEnemy)

PlaneFollowEnemy::PlaneFollowEnemy()
{
  STANDARD_CONSTRUCTOR()
}
```

```
Basic::Ubf::Action* PlaneFollowEnemy::genAction(const Basic::Ubf::State*
    const state, const LCreal dt)
{
   PlaneAction* action = 0;
   const PlaneState* pState = dynamic_cast<const
       PlaneState*>(state->getUbfStateByType(typeid(PlaneState)));

   if (pState!=0 && pState->isAlive()) {
      action = new PlaneAction();

     if (pState->getTargetTrack() != PlaneState::MAX_TRACKS){ // We have a
         target

       // Test to see if we're "BVR" or "WVR"
       if ( (pState->getDistanceToTracked(pState->getTargetTrack()) >=
           (1.5 * Basic::Distance::NM2M)) ) { // BVR (farther away)

           action->setAPNavigate(false);
           action->setFollow(false); // This will turn on the autopilot for
               following
           action->setFollowTarget(true); // This will be our trigger to
               switch who we follow to the enemy

           action->setWeaponTarget(pState->getTargetTrack());

           action->setVote( getVote() ); // More important to follow if we
               have a target...
       }
```

```
else { // We're closer than some set distance, we'll call this
    "within visual range" (WVR) and do a different manuever

    action->setAPNavigate(false);
    action->setFollow(false);
    action->setFollowTarget(false);

    action->setWeaponTarget(pState->getTargetTrack()); // We should
        still set our target, right?



    // Where is our target?
    bool tgtAbove =
        (((pState->getPitchToTracked(pState->getTargetTrack()))) >=
        0 );

    bool tgtRight = ( ((
        (pState->getHeadingToTracked(pState->getTargetTrack())) ) >
        0 )
                && ((
                    (pState->getHeadingToTracked(pState->getTargetTrack()))
                    ) < 180 ) ) ;

    // If he's higher than us, let's dive and speed up
    if (tgtAbove){
       action->setAltitude( (pState->getAltitude() - 500) );
       action->setVelocity( (pState->getVelocity() + 100) );
    }
```

```
        else {
          action->setAltitude( (pState->getAltitude() + 500) );
          action->setVelocity( (pState->getVelocity() - 100) );
        } // otherwise, climb and slow

        // If he's to our right, turn left
        if (tgtRight){ action->setHeading( (pState->getHeading() - 25)
           ); }
        else {         action->setHeading( (pState->getHeading() + 25)
           ); } // otherwise, turn right



        // Turn right and climb for the advantage...
        //action->setHeading( (pState->getHeading() + 15) );
        //action->setAltitude( (pState->getAltitude() + 200) );
        //action->setVelocity( (pState->getVelocity() + 100) );

        action->setVote( getVote() );
    }
}
else { // We don't have a target, follow no one, just keep going
    straight

  action->setAPNavigate(false);
  action->setFollow(false);
  action->setFollowTarget(false);

 action->setAltitude(pState->getAltitude());
```

```
      action->setHeading(pState->getHeading());
      action->setVelocity(pState->getVelocity());

      action->setVote( 1 );
     }

     action->setFltMember( pState->getFlightMember() );

     std::cout << "PlaneFollowEnemy, vote: " << action->getVote() << "\n";
   }
   return action;
}



IMPLEMENT_SUBCLASS(PlaneBreakDefend, "PlaneBreakDefend")
EMPTY_SLOTTABLE(PlaneBreakDefend)
EMPTY_COPYDATA(PlaneBreakDefend)
EMPTY_SERIALIZER(PlaneBreakDefend)
EMPTY_DELETEDATA(PlaneBreakDefend)

PlaneBreakDefend::PlaneBreakDefend()
{
   STANDARD_CONSTRUCTOR()
}

Basic::Ubf::Action* PlaneBreakDefend::genAction(const Basic::Ubf::State*
    const state, const LCreal dt)
{
```

```
PlaneAction* action = 0;
const PlaneState* pState = dynamic_cast<const
    PlaneState*>(state->getUbfStateByType(typeid(PlaneState)));

if (pState!=0 && pState->isAlive()) {
   action = new PlaneAction();

  if (pState->isIncomingMissile()){ // We've got a missile inbound,
      let's break

   // Make sure we're not navigating or following - we need to break
   action->setAPNavigate(false);
   action->setFollow(false);
   action->setFollowTarget(false);



   // How far is the missile away? Maybe we shouldn't break until it's a
       bit closer...
   // Maybe we could use the farther away missile to report to our
       flight lead...
   if (pState->isProximityWarning()){

      // Where is the missile? Right, left, or behind us?
      bool mslRight = ( (((
          (pState->getHeadingToTracked(pState->getMissileTrack())) ) >= 0
          )
```

```
            && ((
                (pState->getHeadingToTracked(pState->getMissileTrack()))
                ) <= 135 ) )
          || (((
              (pState->getHeadingToTracked(pState->getMissileTrack()))
              ) <= -225 )
            && ((
                (pState->getHeadingToTracked(pState->getMissileTrack()))
                ) >= -360 )) );

bool mslLeft = ( (((
    (pState->getHeadingToTracked(pState->getMissileTrack())) ) < 0 )
            && ((
                (pState->getHeadingToTracked(pState->getMissileTrack()))
                ) >= -135 ) )
          || (((
              (pState->getHeadingToTracked(pState->getMissileTrack()))
              ) >= 225 )
            && ((
                (pState->getHeadingToTracked(pState->getMissileTrack()))
                ) <= 360 )) ) ;


// Is the missile coming from the front right, front left, or from
    behind (or straight ahead)
if     (mslRight) {
   action->setHeading( pState->getHeading() - 30 );
```

```
      action->setAltitude( pState->getAltitude() -
          (pState->getAltitude() / 2) ); // Drop half our altitude
          (dive)
      action->setVelocity( pState->getVelocity() + 100 ); // Speed up!
   }   // Hard left turn
   else if (mslLeft) {
      action->setHeading( pState->getHeading() + 30 );
      action->setAltitude( pState->getAltitude() -
          (pState->getAltitude() / 2) ); // Drop half our altitude
          (dive)
      action->setVelocity( pState->getVelocity() + 100 ); // Speed up!
   }   // Hard right turn
   else {
      action->setHeading( pState->getHeading() + 30); // Right turn
      action->setAltitude( pState->getAltitude() - (3 *
          (pState->getAltitude() / 4)) ); // Drop our altitude (dive)
      action->setVelocity( pState->getVelocity() - 200 ); // Slow down!
   }   // behind, let's dive

   // We can drop countermeasures here too, unless we want to do that
       in a different behavior...

   action->setVote( getVote() ); // It's quite important
}
else { // The missile isn't "close" yet, so let's maneuver early to
    help us avoid it...
   action->setAltitude( pState->getAltitude() + 1000 );         //
       Gain potential energy for our dive
```

```
    action->setVelocity( pState->getVelocity() );               // I
        guess we'll maintain speed for now...?

    // Let's do manuevers that help us get ready to evade the missile
        when it gets close...
    double headingToMissileD =
        (pState->getHeadingToTracked(pState->getMissileTrack()) );

    if ( (headingToMissileD >= 0) && (headingToMissileD <= 180) ) { //
        Turn towards the missile, slower as we get close to the heading
        we want
       action->setHeading( pState->getHeading() - (0.25 *
           headingToMissileD) );
    }
    else if ( (headingToMissileD < 0) && (headingToMissileD >= -180) )
        {
       action->setHeading( pState->getHeading() + (0.25 *
           headingToMissileD) );
    }
    else {
       action->setHeading( pState->getHeading() + (0.25 *
           headingToMissileD) );
    }

    action->setVote( (getVote() - (0.3 * getVote())) ); // Not as
        important if we're far away
}
```

```
    }
    else {
     // Make sure we're not navigating or following - just go straight
     action->setAPNavigate(false);
     action->setFollow(false);
     action->setFollowTarget(false);

     action->setAltitude(pState->getAltitude());
     action->setHeading(pState->getHeading());
     action->setVelocity(pState->getVelocity());

      action->setVote( 1 ); // Not a big deal otherwise
    }

    action->setFltMember( pState->getFlightMember() );

    std::cout << "PlaneBreakDefend, vote: " << action->getVote() << "\n";
  }
  return action;
}

} // namespace xBehaviors
} // namespace Eaagles
```

A.2.7 PlaneState.h.

```
//-----------------------------------------------------------------------------
// Class: PlaneState
//-----------------------------------------------------------------------------
#ifndef __Eaagles_xBehaviors_PlaneState_H__
#define __Eaagles_xBehaviors_PlaneState_H__

#include "openeaagles/basic/ubf/State.h"

namespace Eaagles {

namespace Simulation { class Player; }

namespace xBehaviors {

//-----------------------------------------------------------------------------
// Class: PlaneState
//
// Description: this implementation of PlaneState assumes that player
   using this
//            state has only one missile (or is ok with firing all
   missiles at
//            first target)
//-----------------------------------------------------------------------------
class PlaneState : public Basic::Ubf::State
{
  DECLARE_SUBCLASS(PlaneState, Basic::Ubf::State)
public:
```

```
PlaneState();

// Basic::Component Interface
virtual void reset();

// NewUbf::UbfState interface
virtual void updateState(Basic::Component* actor);

// set/get
virtual void setAlive(const bool x)          { alive = x; return; }
virtual bool isAlive() const                 { return alive; }

virtual void setHeading(const double x)      { heading = x; return; }
virtual double getHeading() const            { return heading; }

virtual void setAltitude(const double x)     { altitude = x; return; }
virtual double getAltitude() const           { return altitude; }

virtual void setVelocity(const double x)       { velocity = x; return; }
virtual double getVelocity() const             { return velocity; }

virtual void setAPNavigating(const bool x)       { apNavigating = x;
   return; }
virtual bool isAPNavigating() const              { return apNavigating; }

virtual void setFollowing(const bool x)       { following = x; return; }
virtual bool isFollowing() const              { return following; }
```

```
virtual void setIncomingMissile(const bool x) { incomingMissile = x;
    return; }
virtual bool isIncomingMissile() const      { return incomingMissile; }

virtual void setProximityWarning(const bool x) { proximityWarning = x;
    return; }
virtual bool isProximityWarning() const      { return proximityWarning;
    }

//sets the pitch to current object being tracked
virtual void setPitchToTracked(const unsigned int track, const double
    angle);
virtual double getPitchToTracked(const unsigned int track) const;

virtual void setHeadingToTracked(const unsigned int track, const double
    angle);
virtual double getHeadingToTracked(const unsigned int track) const;

virtual void setDistanceToTracked(const unsigned int track, const
    double distance);
virtual double getDistanceToTracked(const unsigned int track) const;

virtual void setNumTracks(const unsigned int x) { numTracks = x;
    return; }
virtual unsigned int getNumTracks() const    { return numTracks; }

//tracking setter
virtual void setTracking(const bool x)      { tracking = x; return; }
```

```
//returns true if plane is currently tracking
virtual bool isTracking() const             { return tracking; }

virtual void setLead(const unsigned int x) { lead = x; return; }
virtual unsigned int getLead()           { return lead; }
virtual bool isLead() const              { return (flightMember ==
    lead); }

virtual void setFlightMember(const unsigned int x) { flightMember = x;
    return; }
virtual unsigned int getFlightMember() const      { return
    flightMember; }

virtual void setNumMissiles(const unsigned int x)  { numMissiles = x;
    return; }
virtual unsigned int getNumMissiles() const        { return
    numMissiles; }

virtual void setMissileTimer(const double x)   { missileTimer = x;
    return; }
virtual unsigned int getMissileTimer() const        { return
    static_cast<unsigned int>(missileTimer); }
virtual void incMissileTimer(const double x)                    {
    missileTimer += x; return; }

virtual void setTargetTrack(const unsigned int x) { targetTrack = x;
    return; }
virtual unsigned int getTargetTrack() const  { return targetTrack; }
```

```
   virtual void setMissileTrack(const unsigned int x) { missileTrack = x;
      return; }
   virtual unsigned int getMissileTrack() const  { return missileTrack; }
public:
   static const unsigned int MAX_TRACKS = 50;

private:
   void initData();

   bool alive;
   double heading;
   double altitude;
   double velocity;
   bool apNavigating;
   bool following;

   unsigned int lead;
   unsigned int flightMember;

   // From old UBF, for missile and enemy tracking
   double pitchToTracked[MAX_TRACKS];
   double headingToTracked[MAX_TRACKS];
   double distanceToTracked[MAX_TRACKS];

   unsigned int targetTrack;

   unsigned int missileTrack;
```

```
    unsigned int numTracks;
    bool tracking;
    unsigned int numMissiles;
    double missileTimer;

    bool incomingMissile;
    bool proximityWarning;
};

}
}

#endif
```

A.2.8 PlaneState.cpp.

```
//-----------------------------------------------------------------------------
// Class: PlaneState
//-----------------------------------------------------------------------------

#include <string.h>

#include "PlaneState.h"

#include "openeaagles/basic/List.h"
#include "openeaagles/basic/PairStream.h"

#include "openeaagles/simulation/Radar.h"
#include "openeaagles/simulation/Rwr.h"
#include "openeaagles/simulation/TrackManager.h"
#include "openeaagles/simulation/Track.h"
#include "openeaagles/simulation/OnboardComputer.h"
#include "openeaagles/simulation/StoresMgr.h"
#include "openeaagles/simulation/Missile.h"
#include "openeaagles/simulation/AirVehicle.h"
#include "openeaagles/simulation/Simulation.h"
#include "openeaagles/simulation/Autopilot.h"

namespace Eaagles {
namespace xBehaviors {

IMPLEMENT_SUBCLASS(PlaneState, "PlaneState")
EMPTY_SLOTTABLE(PlaneState)
```

```
EMPTY_DELETEDATA(PlaneState)
EMPTY_COPYDATA(PlaneState)
EMPTY_SERIALIZER(PlaneState)

PlaneState::PlaneState()
{
   STANDARD_CONSTRUCTOR()
   initData();
}

void PlaneState::initData()
{
   alive         = false;
   heading       = 0;
   altitude      = 0;
   velocity      = 0;
   apNavigating  = false;
   following     = false;

   // From old UBF
   for (unsigned int i=0; i<MAX_TRACKS;i++) {
      pitchToTracked[i] = 0.0;
      headingToTracked[i] = 0.0;
      distanceToTracked[i] = 0.0;
   }

   targetTrack   = MAX_TRACKS; // 0 is a valid target track, use MAX_TRACKS
       to signal
```

```
                                // "no tgt track"

   numTracks     = 0;
   numMissiles  = 0;
   tracking      = false;
   incomingMissile = false;

   missileTimer = 500;
   lead = 0;
   flightMember = 0;
}

void PlaneState::reset()
{
   initData();
   BaseClass::reset();
}

void PlaneState::updateState(Basic::Component* actor)
{
    setAlive(false); // Set up

   // Get our player
   Simulation::Player* player = dynamic_cast<Simulation::Player*>( actor );
      if (player != 0) {
      // Get some simple info about the player and set our UBF state
          accordingly
      setAltitude(player->getAltitudeFt());
```

```
setAlive(player->isActive());
setHeading(player->getHeadingD());
setVelocity(player->getGroundSpeedKts());

// Is autopilot navigating?
Simulation::Autopilot* ap = dynamic_cast<Simulation::Autopilot*>(
    player->getPilot() );
if (ap != 0) { // We have an autopilot
   setAPNavigating(ap->isNavModeOn());
   setFollowing(ap->isFollowTheLeadModeOn());
}
else { // No autopilot
   setAPNavigating(false);
   setFollowing(false);
}

// Set our flight lead and flight member based on wall formation:

// Let's find our flight and flight member number...
Basic::String pname( (player->getName())->getString() ); // Our
    player's name

std::cout << "\nName: " << pname << "\n";

// Get the last character, which is the flight member #
Basic::String fltMember;
Basic::String flightNm;
```

```
pname.getSubString(flightNm, 0, (pname.len() - 1));
pname.getSubString(fltMember, (pname.len() - 1), pname.len());

// Who are we?
unsigned int me = fltMember.getInteger();

// Now let's see if anyone who outranks us is dead...
for (unsigned int i = 1; i <= 4; i++){
   char fltNum[2];
   std::sprintf(fltNum, "%d", i);
   Basic::String fltMbrName( flightNm, fltNum ); // Which flight
      member are we interested in?

   // If the flight member in question is higher ranking than me...
   if (i < me){
      Simulation::Player* fltMbr =
         dynamic_cast<Simulation::Player*>(player->getSimulation()->findPlayerByName(fl
      if (fltMbr != 0){ // Does this flight member exist?
         if ( !(fltMbr->isActive()) ){ // Are they dead?
            me--; // If so, we're higher ranking than we thought...
         }
      }
   }
}
setFlightMember(me); // Now we can set our place in the flight...

// Who is our lead? The flight lead or the element lead?
if (getFlightMember() == 4){
```

```
    setLead(3);
  }
  else {
    setLead(1);
  }

  // Get our air vehicle
  Simulation::AirVehicle* airVehicle =
     dynamic_cast<Simulation::AirVehicle*>(actor);

  // Let's calculate how many missiles we have
  //  and see if we just launched a missile
  const Simulation::StoresMgr* stores =
     airVehicle->getStoresManagement();
  if (stores != 0) {
    // For now we're assuming every weapon is a missile

    if (getNumMissiles() > stores->available(){ // Did we just launch
        a missile? (1 less in our stores)
      setMissileTimer(0); // Restart our missile timer for our
          behavior...
    }
    else {
      incMissileTimer(1); // Increment our missile timer otherwise
    }

    // Now we update our number of weapons
    setNumMissiles(stores->available());
```

```
}
else { // We don't have any stores
   setNumMissiles(0);
} // End if-else stores management

// Get our onboard computer
const Simulation::OnboardComputer* oc =
    airVehicle->getOnboardComputer();
if (oc != 0) {
   // Get our track manager
   const Simulation::TrackManager* trackManager =
       oc->getTrackManagerByType(typeid(Simulation::TrackManager));
   if (trackManager != 0) {
      // Now let's get our track list (all of the radar tracks)
      SPtr<Simulation::Track> trackList[MAX_TRACKS]; // MAX_TRACKS is
          50
      unsigned int nTracks = trackManager->getTrackList(trackList,
          MAX_TRACKS);
      setNumTracks( ((nTracks < MAX_TRACKS) ? nTracks : MAX_TRACKS) );
          // Set the number of tracks that we see...or MAX_TRACKS
                                                                  // if
                                                                     there
                                                                     are
                                                                     more
                                                                     than
                                                                     MAX_TRACKS
```

```
// Some tracking stuff to see if we're still tracking enemies or
    missiles
unsigned int numEnemies = 0;
unsigned int numMissiles = 0;

// If we don't see any tracks
if (nTracks == 0) {
   setTracking(false);
}
else {
   setTracking(true); // Not indicitave of a target, just that
      we're tracking (i.e.,
                  //  we see one or more tracks on our radar)...
   // Initialize our target and missile tracks in case we don't
      have a target...
   setTargetTrack(PlaneState::MAX_TRACKS);
   setMissileTrack(PlaneState::MAX_TRACKS);

   // Initialize our missile warnings
   setIncomingMissile(false);
   setProximityWarning(false);

   // So we can find the closest target / missile
   double minAngleTarget = 180;
   double minDistMissile = -1.0;

   // Iterate through each of the tracks
```

```
for (int trackIndex = nTracks -1; trackIndex >= 0;
    trackIndex--) {

  // Set the heading, pitch, and distance to each of the
     tracks
  setHeadingToTracked(trackIndex,
     trackList[trackIndex]->getRelAzimuthD());
  setPitchToTracked(trackIndex,
     trackList[trackIndex]->getElevationD());
  setDistanceToTracked(trackIndex,
     trackList[trackIndex]->getRange());

  // What is the target track? An aircraft or a missile?
     (Later we could look for other stuff,
  // like ground vehicles or buildings, etc.)
  if
     (trackList[trackIndex]->getTarget()->isMajorType(Simulation::Player::AIR
    // Now we should check if this is a worthy track to be
       our "target"
    if (trackList[trackIndex]->getTarget()->isActive()
                                  // Is it alive?
      &&
          trackList[trackIndex]->getTarget()->isNotSide(player->getSide()))
          { // Is it an enemy?

      // We have at least one active enemy aircraft (so we
          have at least one possible target)
      numEnemies++;
```

```
unsigned int missilesInbound = 0;

// Let's look to see if this target is already
    targeted by a missile... if so, we want to go on to
//  our next potential target, and ignore this one...
for (int trackIndex2 = nTracks -1; trackIndex2 >= 0;
    trackIndex2--) {
   // Let's only look at missiles, looking at other
       friendly aircraft would be too complicated...
   Simulation::Weapon* msl =
       dynamic_cast<Simulation::Weapon*>(trackList[trackIndex2]->getTarg
   if (msl != 0){
      if ( msl->getTargetPlayer() != 0 ){ // Make sure
          the target player still exists...
         // Check to see if our current potential
            target (trackIndex) is being pursued by a
            missile (trackIndex2 and msl)

         unsigned short missileTargetID =
            (msl->getTargetPlayer()->getID()); // the
            missile's target
         unsigned short playerID =
            (trackList[trackIndex]->getTarget()->getID());
            // the player we want to target
         if ( (missileTargetID) == (playerID) ){
            missilesInbound++;
         }
```

```
      } // end if target player exists
    } // end if it's a missile
  } // End for loop through looking for missiles

  // Make sure it's not already targeted by missiles...
  if ( missilesInbound == 0 ) {
    // These things mean that we can set this as our
        target, but first
    //  lets check that it's the "best" track for us
        (smallest angle to target)
    double angleTgt =
        trackList[trackIndex]->getRelAzimuthD();

    // Let's get our angle in the -180 to 180 degree
        window
    if (angleTgt > 180) { angleTgt = angleTgt - 360; }
    else if (angleTgt < -180) { angleTgt = angleTgt +
        360; }

    // Absolute value for easiest comparison
    angleTgt = std::abs(angleTgt);

    if (( angleTgt < minAngleTarget )){ // it's the
        best target, or...
          setTargetTrack(trackIndex); // ...if it isn't,
              we'll set this as our target
          minAngleTarget = ( angleTgt ); // and set the
              min angle
```

```
        } // End if the enemy is best
      } // end if our intended target is already targeted
    } // End if it's alive and an enemy
  }
  // If the target track is a weapon, we want to look for
     incoming missiles
  else { //if
     (trackList[trackIndex]->getTarget()->isMajorType(Simulation::Player::WEA

    Simulation::Weapon* msl =
        dynamic_cast<Simulation::Weapon*>(trackList[trackIndex]->getTarget());
    if (msl != 0){
      if ( msl->getTargetPlayer() != 0 ){ // Make sure the
         target player exists...
        if ( (msl->getTargetPlayer()->getID()) ==
            (player->getID()) ){ // Is it us?? ID's are
            unique by player

            setIncomingMissile(true); // We have an inbound
               missile whether it's closest or not -
               AAAAHHHH!
            numMissiles++;

            // Let's make sure it's the closest missile (if
               there are multiple...but we certainly hope
               not!)
            if ((trackList[trackIndex]->getRange() <
               minDistMissile) || (minDistMissile == -1.0)){
```

```
                    setMissileTrack(trackIndex);
                    minDistMissile =
                        trackList[trackIndex]->getRange();
                }

                // Set our proximity warning if the missile is
                    within a certain distance threshold
                if ( ((minDistMissile <= (2 *
                    Basic::Distance::NM2M)) && (minDistMissile !=
                    -1.0)) ){
                    setProximityWarning(true);
                }
                else { setProximityWarning(false); }

            } // end if target player is us
        } // end if the target player is real
    } // end if the missile is real
  } // end if-else what type is it?
 } // End for (loop through tracks)
} // end if-else we're not tracking targets

// If we no longer have a valid target, we need to remove that
    from our state...
if ( numEnemies == 0 ) { setTargetTrack(PlaneState::MAX_TRACKS);
    }

// If we no longer are tracking missiles, we should remove any
    incoming missile warnings...
```

```
          if ( numMissiles == 0 ) {
             setIncomingMissile(false);
             setProximityWarning(false);
             setMissileTrack(PlaneState::MAX_TRACKS);
          }

        } // End if track manager
      } // End if onboard computer
   } // end if our player exists

   BaseClass::updateState(actor);
}



void PlaneState::setPitchToTracked(const unsigned int trackNumber, const
    double angle)
{
   if ( trackNumber<numTracks ) {
      pitchToTracked[trackNumber] = angle;
   }
}

double PlaneState::getPitchToTracked(const unsigned int trackNumber) const
{
   if ( trackNumber<numTracks ) {
      return pitchToTracked[trackNumber];
   }
```

```
  std::cout << trackNumber << " is out of bounds of the tracking array of
      PlaneState! Error!\n";
  return trackNumber;
}

void PlaneState::setHeadingToTracked(const unsigned int trackNumber, const
    double angle)
{
  if (trackNumber < numTracks) {
    headingToTracked[trackNumber] = angle;
  }
}

double PlaneState::getHeadingToTracked(const unsigned int trackNumber)
    const
{
  if (trackNumber < numTracks) {
    return headingToTracked[trackNumber];
  }
  std::cout << trackNumber << " is out of bounds of the tracking array of
      PlaneState! Error!\n";
  return trackNumber;
}

void PlaneState::setDistanceToTracked(const unsigned int trackNumber,
    const double distance)
{
  if (trackNumber < numTracks) {
```

```
      distanceToTracked[trackNumber] = distance;
   }
}

double PlaneState::getDistanceToTracked(const unsigned int trackNumber)
    const
{
   if (trackNumber < numTracks) {
      return distanceToTracked[trackNumber];
   }
   std::cout << trackNumber << " is out of bounds of the tracking array of
       PlaneState! Error!\n";
   return trackNumber;
}

}
}
```

A.2.9 WTAArbiter.h.

```
//-----------------------------------------------------------------------------
// Class: WTAArbiter
//-----------------------------------------------------------------------------
#ifndef __Eaagles_xBehaviors_WTAArbiter_H__
#define __Eaagles_xBehaviors_WTAArbiter_H__

#include "openeaagles/basic/ubf/Arbiter.h"

namespace Eaagles {

namespace Basic { class List; class Action; }

namespace xBehaviors {

//-----------------------------------------------------------------------------
// Class: WTAArbiter
//
// Description: Winner takes all arbiter for a plane
//-----------------------------------------------------------------------------
class WTAArbiter : public Basic::Ubf::Arbiter
{
  DECLARE_SUBCLASS(WTAArbiter, Basic::Ubf::Arbiter)

public:
  WTAArbiter();
```

```
    // generates an action based upon the recommended actions in the
        actionSet
    virtual Basic::Ubf::Action* genComplexAction(Basic::List* const
        actionSet);

private:

};

}
}

#endif
```

A.2.10 WTAArbiter.cpp.

```
//-----------------------------------------------------------------------------
// Class: WTAArbiter
//-----------------------------------------------------------------------------
#include "WTAArbiter.h"

#include "openeaagles/basic/List.h"

#include "PlaneAction.h"

namespace Eaagles {
namespace xBehaviors {

IMPLEMENT_SUBCLASS(WTAArbiter, "WTAArbiter")
EMPTY_SLOTTABLE(WTAArbiter)
EMPTY_CONSTRUCTOR(WTAArbiter)
EMPTY_COPYDATA(WTAArbiter)
EMPTY_SERIALIZER(WTAArbiter)
EMPTY_DELETEDATA(WTAArbiter)

Basic::Ubf::Action* WTAArbiter::genComplexAction(Basic::List* const
    actionSet)
{
   PlaneAction* complexAction = new PlaneAction;

   unsigned int maxVote = 0;

   // process entire action set
```

```
const Basic::List::Item* item = actionSet->getFirstItem();
while (item != 0) {
   const PlaneAction* action = dynamic_cast<const
      PlaneAction*>(item->getValue());
   if (action!=0) {
      if (action->getVote() > maxVote) {
       complexAction->setHeading(action->getHeading());
       complexAction->setAltitude(action->getAltitude());
       complexAction->setVelocity(action->getVelocity());
       complexAction->setAPNavigate(action->getAPNavigate());
       complexAction->setFollow(action->getFollow());
       complexAction->setFollowTarget(action->getFollowTarget());
       complexAction->setReleaseWeapon(action->getReleaseWeapon());
       complexAction->setWeaponTarget(action->getWeaponTarget());
       complexAction->setFltMember(action->getFltMember());
       complexAction->setVote(action->getVote());

         maxVote = action->getVote();

       setVote(action->getVote());
      }

   }
   else {
      std::cout << "Action NOT a PlaneAction\n";
   }

   // next action
```

```
    item = item->getNext();
  }

  std::cout << "FollowTarget: " << (complexAction->getFollowTarget() ?
      "Yes" : "No") << "\n";
  std::cout << "APNavigate: " << (complexAction->getAPNavigate() ? "Yes"
      : "No") << "\n";

  return complexAction;
}

}
}
```

A.2.11 Test.

```
if (test){
}
```

A.2.12 Test 2.

```
if (test2){
}
```

A.3 Test 3

```
if (test3){
}
```

Bibliography

[1] R. A. Brooks, "A robust layered control system for a mobile robot," *Robotics and Automation, IEEE Journal of*, vol. 2, no. 1, pp. 14–23, 1986.

[2] E. Gat *et al.*, "On three-layer architectures," 1998.

[3] V. Braitenberg, *Vehicles: Experiments in Synthetic Psychology*. Bradford Books, MIT Press, 1986.

[4] B. G. Woolley and G. L. Peterson, "Unified behavior framework for reactive robot control," *Journal of Intelligent and Robotic Systems*, vol. 55, no. 2-3, pp. 155–176, 2009.

[5] "human eye," *Encyclopedia Britannica. Encyclopedia Britannica Online. Encyclopedia Britannica Inc., 2015. Web*, vol. 16, Feb. 2015.

[6] N. J. Nilsson, "Shakey the robot," tech. rep., DTIC Document, 1984.

[7] D. Isla, "Gdc 2005 proceeding: Handling complexity in the halo 2 ai," *Retrieved October*, vol. 21, p. 2009, 2005.

[8] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ogren, "Towards a unified behavior trees framework for robot control," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, 2014.

[9] D. D. Hodson, D. P. Gehl, and R. O. Baldwin, "Building distributed simulations utilizing the eaagles framework," in *The Interservice/Industry Training, Simulation & Education Conference (I/ITSEC)*, vol. 2006, NTSA, 2006.

[10] S. L. G. G. L. Z. Sandeep S. Mulgund, Karen A. Harper, "Situation awareness for pilot-in-the-loop evaluation," Tech. Rep. R96011, Charles River Analytics, 725 Concord Ave. Cambridge, MA 02138, mar 1999.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, 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 Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
26–03–2015	Master's Thesis	Oct 2013–Mar 2015

4. TITLE AND SUBTITLE

The Unified Behavior Framework for the Simulation of Autonomous Agents

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Roberson, Daniel M., First Lieutenant, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

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

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-15-M-014

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

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Since the 1980s, researchers have designed a variety of robot control architectures intending to imbue robots with some degree of autonomy. A recently developed architecture, the UBF, implements a variation of the three-layer architecture with a reactive controller to rapidly make behavior decisions. Additionally, the UBF utilizes software design patterns that promote the reuse of code and free designers to dynamically switch between behavior paradigms. This paper explores the application of the UBF to the simulation domain. By employing software engineering principles to implement the UBF architecture within an open-source simulation framework, we have extended the versatility of both. The consolidation of these frameworks assists the designer in efficiently constructing simulations of one or more autonomous agents that exhibit similar behaviors. A typical air-to-air engagement scenario between six UBF agents controlling both friendly and enemy aircraft demonstrates the utility of the UBF architecture as a flexible mechanism for reusing behavior code and rapidly creating autonomous agents in simulation.

15. SUBJECT TERMS

Autonomy, UBF, Robotics, Simulation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Douglas Hodson (ENG)
U	U	U	UU	168	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4719 douglas.hodson@afit.edu

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18