EXTENDING AFSIM WITH BEHAVIORAL EMERGENCE

THESIS

Jeffrey L. Choate, Captain, USAF

AFIT-ENG-MS-17-M-014

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-17-M-014

EXTENDING AFSIM WITH BEHAVIORAL EMERGENCE

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 Science

Jeffrey L. Choate, B.S.E.E.

Captain, USAF

March 2017

AFIT-ENG-MS-17-M-014

EXTENDING AFSIM WITH BEHAVIORAL EMERGENCE

THESIS

Jeffrey L. Choate, B.S.E.E.
Captain, USAF

Committee Membership:

Dr. Gilbert L. Peterson
Chairman

Dr. Douglas D. Hodson
Member

Capt Jason M. Bindewald, PhD
Member

AFIT-ENG-MS-17-M-014

Abstract

The Advanced Framework for Simulation, Integration, and Modeling (AFSIM) provides a capability to evaluate mission level scenarios described in its scripting language. The AFSIM scripting language includes multiple intelligent agent modeling techniques, none of which explicitly provide the ability to have behaviors emerge. Behavioral emergence occurs when a system composed of many simple behaviors working together exhibits a complex pattern not directly attributable to the simpler components. Without behavioral emergence an intelligent agent designer must explicitly write methods for every combination of circumstances that their agent may encounter. A priori consideration of every possible configuration of the world state is intractable. This problem can be solved by adding the Unified Behavior Framework (UBF) to AFSIM which provides a means to explicitly control behavioral emergence. This thesis creates a plug-in exposing UBF to AFSIM, extending AFSIM's scripting language, and demonstrating behavioral emergence via a case study of these new behaviors.

Acknowledgements

I would like to thank my wife for the support while working on this thesis and I would like to thank Dr. Peterson and the Air Force for allowing me the opportunity to learn about intelligent agent control structures over the past year.

Jeffrey L. Choate

Table of Contents

Page

Page

List of Figures

Figure | Page

List of Tables

I. Introduction

Modeling and simulation (M&S) tools are used to create data for real world decision making [2]. These tools allow for simulations of dangerous scenarios without the loss of life or risk of harm to real people or assets. These tools may be run faster than real time allowing for many strategies to be explored in order to find the most desirable [3]. These tools provide realism by emulating the behaviors of intelligent agents. Modeling and simulation programs are useful from their safety, realism, and numerousness of strategies they may explore.

Modeling and simulation tools use a variety of simple logic and intelligent agent control structures in order to create the decision sequences of their components [3]. Simple control structures are programs on infinite loops making decisions linearly; in a broad sense intelligent agent control structures may be explained similarly. However, intelligent agent control structures add organization and modularity to the infinite loops. These additions are tree structures, states, behaviors, predefined code blocks, transition functions, and many more [3, 4, 5]. Simple logic and intelligent agent frameworks provide for the decision sequences in modeling and simulation tools.

Simple logic and intelligent agent frameworks are included in modeling and simulation applications in a variety of ways. The most basic method is when an application forces a user to make their decisions in C++, or another programming language [6, 7]. Another method provides some structure for users by limiting and generalizing the commands they can use over a full C++ type language, called a scripting language or scripting [3]. Graphical user interfaces can also be used to force structure on a user and help them visualize the structure of the underlying language [8]. Also, many languages may be combined or used in conjunction to provide all of their benefits at the cost of adding complexity to the resulting scripting language. A developer needs to balance the size and contents of the language to combat this. Ultimately, simple logic and intelligent agent frameworks are included in modeling and simulation tools via predetermined sets of commands, or languages.

Users of an intelligent agent framework gain the advantages of the structure but are forced into the disadvantages of that structure [9]. Modeling and simulation tools attempt to overcome this by including multiple intelligent agent control frameworks [3]. However, new frameworks are continuously being created and modified. Continuous assessment is needed to determine if new frameworks should be added.

1.1 Problem Statement

The Advanced Framework for Simulation, Integration, and Modeling (AFSIM) includes intelligent agent frameworks and a variety of commands, none of which explicitly provide the capability of behavioral emergence. Behavioral emergence occurs when a system composed of many simple behaviors working together exhibits a complex pattern not directly attributable to the simpler components [10]. Without behavioral emergence an intelligent agent designer must explicitly design behaviors for every combination of circumstances that their agent may encounter. A priori consideration of every possible configuration of the world state is intractable. This problem can be solved by adding the Unified Behavior Framework (UBF) to AFSIM which provides a means to explicitly control behavioral emergence.

1.2 Thesis Objective

The primary objective of this thesis is to extend the AFSIM scripting language with the UBF. The resulting plug-in to the AFSIM executable allows for emergent behaviors in AFSIM.

1.3 Demonstrated Advancements

The thesis objective itself has a succinct answer, being a few pages of new terms needed that AFSIM does not currently utilize. However, extending a scripting language with a new structure requires a comparison to other existing frameworks for multiple reasons. The first reason is in order to include optimizations where compatible. The second is to map synonymous terms in order to prevent confusion for readers familiar with other frameworks. The third reason is to allow for reproduction of the thesis in a different environment by displaying the concepts that are implemented by the plug-in versus those already included in AFSIM's scripting language.

This thesis implements the UBF action objects with a slightly different technique than past implementations to provide an increase in platform independence. Accompanying this new technique are disadvantages and advantages. The advantages and disadvantages should be examined in order to allow a reader to decide if they consider this technique worthwhile for their own use or not.

In order to identify the advancements demonstrated through this thesis the following two questions will be answered:

1. How do the commands in this language cover commands from other languages or frameworks?

2. What are the key advantages or disadvantages in implementing UBF in a platform independent way appropriate to AFSIM?

1.4 Sponsor

This research is sponsored by the Aerospace Systems Directorate, Modeling and Simulation Branch of the Air Force Research Laboratories (AFRL/RQQD) at Wright-Patterson Air Force Base. AFRL/RQQD uses the Advanced Framework for Simulation, Integration, and Modeling (AFSIM) as their current modeling and simulation framework. This thesis is oriented at improving the intelligent agent control and design capabilities of AFSIM via additions to AFSIM's library of script commands.

1.5 Contributions

This thesis creates a dynamic link library that serves as a plug-in to AFSIM executables. This allows the plug-in to be small and easily transferable between individuals in the AFSIM community. It also allows for the plug-in to be maintained separately from releases of the main AFSIM software. Tying in to the AFSIM executable provides the plug-in and its background C++ class structure to the AFSIM script language.

The plug-in exposes the UBF class structure and its benefits to the AFSIM analyst. This adds a designed method of implementing emergent behaviors and tuning them [9]. This also adds an increase in flexibility by allowing an AFSIM user to choose the agent architecture [6, 9]. Ultimately this provides AFSIM analysts the capability to create behaviors in their intelligent agents, i.e. simulated aircraft, that they could not or was difficult before.

The capability to create behaviors using this plug-in that were difficult before has its own contributions. One is by reducing development time using the now built-in capabilities. Another is by increasing maintainability, modularity, and modifiability by replacing overly complicated solutions with smaller simpler solutions that have the same effect. Modularity is also enhanced through the platform independence of this implementation via the usage of all custom action recommendations over that of a pre-determined action vector. Another way modularity is increased is via UBF behavior's ability to communicate with one another enabling simple mapping of behaviors between implementations. From those contributions is the potential for AFSIM analysts to save time in creating intelligent agents and their employers to save money as a result.

These contributions are not unique to AFSIM or modeling and simulation community. Through this thesis, non-AFSIM specific class diagrams are provided to allow developers insight into the necessary objects. All commands that were implemented and exposed to AFSIM are documented. Also, required commands reused from AFSIM are identified. These items allow a developer of robots, video games, and any other intelligent agent controller the tools to implement their own version of a UBF extension to a scripting language.

1.6 Results

This thesis identifies and provides the commands necessary to expose behavioral emergence in the Unified Behavior Framework (UBF) to AFSIM analysts. Case studies are used to demonstrate this behavioral emergence. The first study is a scenario acting as a proof of concept that UBF is able to replace a BT in AFSIM. It does this by taking an AFSIM training scenario and replacing the BT with a UBF tree. The enemy aircraft destroyed and ally targets lost for both BT and UBF tree scenario are the same, simply showing a proof of concept that a UBF tree can replace the Behavior Trees (BT) used in AFSIM.

The second case study compares a BT implementation and a UBF tree in order to evaluate the effects of behavioral emergence. Each structure uses two main behaviors that are similar, with only required structural differences. Multiple UBF agents are created with different 'vote' values for their obstacle avoidance behavior and all agents' time to reach a goal point is measured for comparison. These measurements show tuning UBF behavior's output is necessary and UBF agents can achieve a goal faster while still meeting the objectives compared to a BT agent.

The third scenario acts as a proof of concept demonstrating UBF's ability to create behavioral emergence. It does this by implementing a classical behavioral emergence technique for swarming, Boids [11], by implementing the three tenet behaviors of Boids. Hence, a behavior each for 'Cohesion', 'Separation', and 'Alignment' tenets are created and utilize the existing structure from the second case study. Five aircraft are given the Boids inspired UBF tree and the UBF agent from scenario two is reused on a single aircraft. During the scenario the swarm agents group up with one another, shift the group towards the scenario two agent, and maintain the group for the remainder of the scenario, essentially allowing a swarming behavior to emerge from the built in features of UBF.

The fourth scenario examines the effort required to modify a BT in comparison to a UBF tree. Scenario two and three structures are used as the concepts that are to be combined. This scenario demonstrates the fact that maintaining and extending a BT is proportional to the number of behaviors affected. UBF effectively combats this proportionality of effort via its arbitration system, structure, and increased capability of code reuse.

While those examples explore the behavioral emergence of UBF, extending AFSIM's scripting language around UBF has two other concerns. The first concern is how other frameworks' and languages' concepts are covered. Maximizing the concepts that are implemented by this plug-in is accomplished in order to provide capabilities and optimizations that the original UBF structure may not have. This plug-in does not implement every concept identified, however Section 6.2 provides ideas for the remaining concepts. In order to show readers how the other concepts are implemented, Section 3.2 provides a discussion of each concept and the method with which it is implemented.

The second concern is how the platform independence of this implementation affects users versus other UBF implementations; the platform independence referred to here is the generic value fields of the action objects. Through the scenarios it can be seen that this extension requires additional work initially. This is because the action objects have to be mapped to outputs; establishing a standard mapping for an agent can mitigate that issue. This increase in effort can be considered an advantage because it provides the capability for a user to translate behaviors that others create to their own tree's input requirements. The platform independence of this implementation initially increases workload for users while providing an increase in flexibility.

1.7 Assumptions and Terms

Many of the terms and techniques discussed in this thesis are independent of specific programming languages. However, general Objected Oriented (OO) knowledge is assumed when discussing the implementation of the framework into the AFSIM code base and the language into the AFSIM script's grammars.

Also, some terms used are for a specific purpose even if semantically similar. These terms are 'user', 'analyst', and 'developer'. 'Developer' always refers to an individual working in the C++ code base of AFSIM; this includes working on a plug-in for the code base, which is the method used by this thesis. 'Analyst' always refers to an individual working in the script language of AFSIM. Analysts utilize the commands and tags implemented by developers to create intelligent agents and complex mission level scenarios. 'User' is a general term, referring to neither role specifically. A single individual may inherit any or all of these roles but the roles are made clear because that is a typical role distinction with AFSIM users and it distinguishes between portions of this thesis' effort.

Terms that start with 'Wsf' indicate object types provided by AFSIM. Typical object types in this thesis are WsfTrack, WsfGeoPoint, and WsfRoute. WsfTrack is a radar track object providing details about another agent in the environment. The WsfGeoPoint is an object which groups together coordinates in multiple formats as well as altitude. WsfRoute is an object consisting of a series of geographical coordinates that an agent may be instructed to follow.

1.8 Thesis Structure

This thesis is structured as follows. This chapter introduces the overall concepts and goals. Chapter II presents an overview of other intelligent frameworks and summarizes the concepts and commands in them. Chapter III presents the class structures, flow of control and data through UBF behaviors, a map of the concepts to AFSIM and this implementations terms, and a manual for each of the new commands added to AFSIM. Chapter IV is the experimental implementation and the evaluation criteria used to demonstrate the new capabilities. Chapter V presents the findings which resulted from the implementation of the new UBF behaviors in AFSIM. Chapter VI covers the conclusions of this research. Appendix A contains the code used to create the UBF C++ structure and implement it in AFSIM. Appendix B contains the scripts used in the various scenarios throughout the thesis.

II. Intelligent Agent Architectures/Frameworks/Languages

In Chapter I, the main issue this thesis addresses is identified as the Advanced Framework for Simulation, Integration, and Modeling (AFSIM) lacking the capability of behavioral emergence. Adding the Unified Behavior Framework (UBF) to AFSIM can solve this issue because UBF is capable of behavioral emergence. However, AFSIM creates scenarios through its scripting language and UBF has not had a language created around it previously. To fill this capability gap in AFSIM, its scripting language must be extended to include and implement UBF via custom UBF behaviors.

When extending a language to fill a capability gap, a review of related works is necessary. The review first and foremost establishes a set of requirements and capabilities that other frameworks cover. A new custom UBF behavior object should strive to cover the capabilities of other frameworks to prevent a user from needing/using multiple controllers to gain their multiple benefits and to prevent a user from sacrificing a capability in their controller choice. To identify the commands and components necessary, a notional behavior component definition is made for each of the intelligent agent controllers that are reviewed. Finally, this chapter provides a summary of all reviewed intelligent agent controllers to centralize and allow traceability of all concepts and components.

2.1 Behavior Component Definition for Comparison

To succinctly present the components of the many intelligent agent controllers a behavior component definition is used. Presenting each framework succinctly allows readers to view the components of an intelligent agent controller at a glance. Using a common definition for each of these controllers allows a reader's glances to easily compare the components amongst them. We chose to label our component definition for each controller with the term "behavior" because that is the basic building block of the many of the controllers as well as our objective framework. To be consistent, even controllers that do not specifically utilize or call their components "behaviors" have this definition presented for them; this is to allow a means for comparison. Thus, a succinct behavior component definition, Table 1 assists with the understanding of the various intelligent agent controllers and with mapping them to the extension accomplished by this thesis.

Table 1. Behavior Definition

Let G be the behavior component of an intelligent agent.
G=$\{S, O, C, F\}$
S is the signature identifying a behavior
O is the organization of the behavior structure
C are commands which act a behavior
F are flags associating information and attributes to a behavior

2.2 Robot Architectures

Here a look at intelligent agent frameworks is provided by focusing on implementation efforts of physical robots. This examination is useful because robotic implementations represent a system implementation that is comprehensive enough for a specific platform and thus provide insight into needed functionality that looking only at the controller portion of an agent could overlook.

2.2.1 Subsumption.

Subsumption is one of the earliest intelligent agent controllers [1, 12] which started to provide a methodology to organize behaviors. Subsumption organizes behaviors into a layered web of behaviors which take inputs from and provide outputs to their intelligent agent and to one another; see Figure 1 for an example Subsumption structure. The outputs of the behaviors are combined by halting an output signal with an inhibit decision, overwriting an output signal with a suppression decision, or simply being used as input to another behavior. The layered concept provides for more important decisions, such as avoiding obstacles, to always be considered and maintain their effectiveness even when additional levels are added [1]. With this simple organization of behaviors, Subsumption was the "best known departure from the sense-plan-act" idiom [12] to a grouping of task/behavior oriented units.

The behavior component of Subsumption is defined in Table 2:

Table 2. Subsumption Definition

G=$\{S, O, C, F\}$
S=Name of behavior
O=Tree, environment actuation at the root, outputs of parents used by children as inputs or outputs are suppressed or inhibited
C=None
F=*Type* (*Inhibit*, *Suppress*, *Behavior*)

2.2.2 Colony.

Colony is a descendant of Subsumption [9] controller framework. It is slightly different from Subsumption in that it only uses the suppression operation. This causes the behaviors to use a "fixed-priority arbitration system" [13] as the decisions cascade down. Thus, introducing the concept of "priority based behavior hierarchies"[9].

The behavior component of a Colony is defined in Table 3:

2.2.3 Motor Schema.

The Motor Schema Architecture emerged in the late 1980s and provided one of the first uses of emergent behaviors [14]. It created emergent behaviors by allowing each

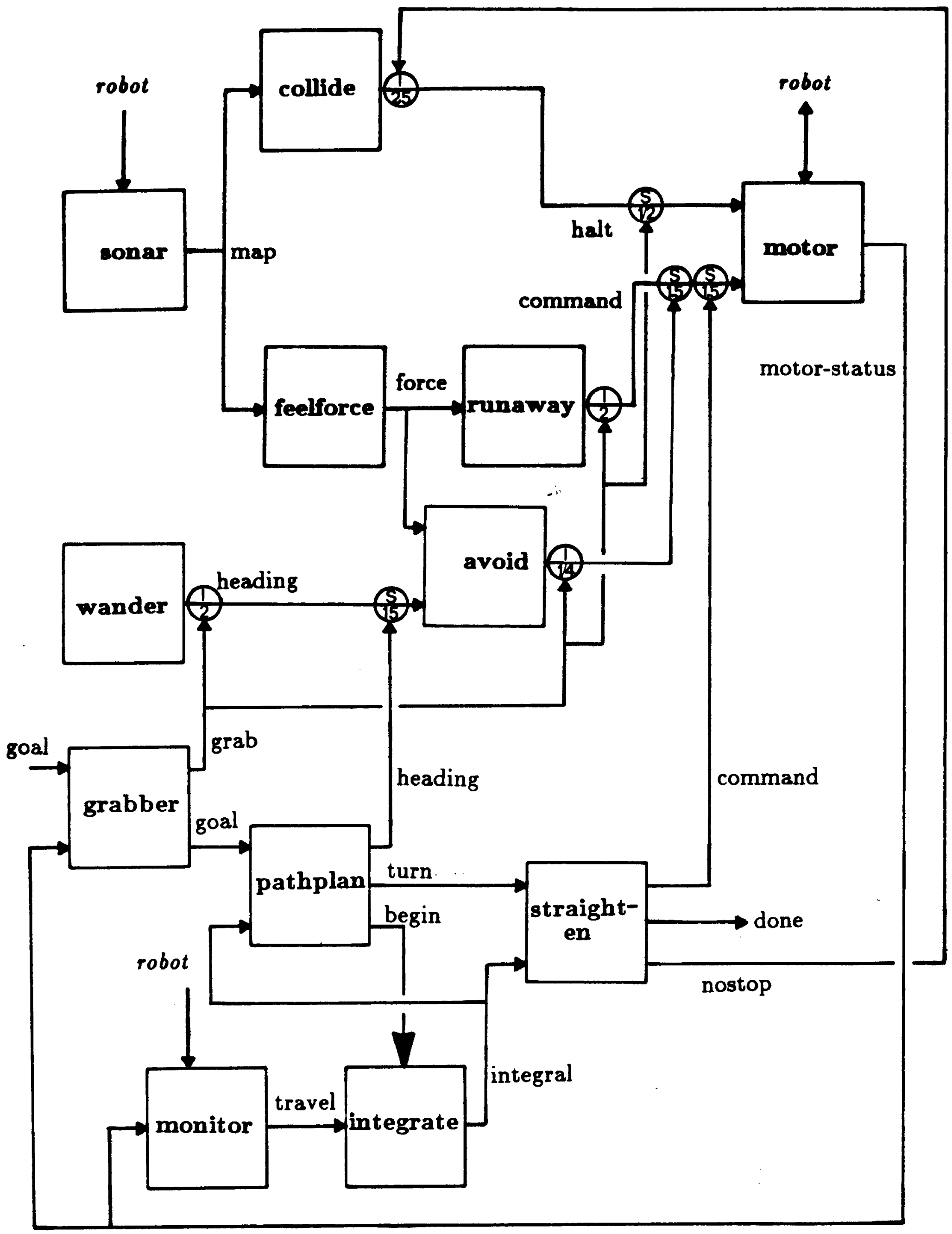


Figure 1. Example Subsumption Behavior Structure [1].

Table 3. Colony Definition

G=$\{S, O, C, F\}$
S=Name of behavior
O=Hierarchical Tree of behaviors with parents suppressing children
C=None
F=*Type* (*Behavior* or *Suppress*)

behavior to fuse their output into a potential vector field. This concept applies well to navigation, however is difficult to extend to motor tasks such as aiming a weapon, firing a weapon, or other mutually exclusive concepts. This is because a potential field does not lend itself well to discrete objectives; adding two aiming vectors could result in shooting between two targets versus choosing one to be shot.

The behavior component of Motor Schema is defined in Table 4:

Table 4. Motor Schema Definition

G=$\{S, O, C, F\}$
S=Name of behavior
O=List of instantiated behaviors which have their potential fields fused
C=instantiate, de-instantiate
F=None

2.2.4 Saphira/ARIA Architecture.

The focus of Saphira [9, 6, 15] is not specific to behaviors; it is an architecture, it utilizes the planning domain, it has memory, and it has many other built in processes like mappers and speech input modules. Saphira implements ARIA [15] (ActiveMedia Robotics Interface for Applications) as its controller portion. ARIA defines a Local Perceptual Space (LPS), uses the Procedural Reasoning System Lite (PRS-Lite) [15, 16], and allows for various other processes to integrate and improve it. The LPS holds the agent's world representation, inputs from other processes at the controller level, and directly receives some inputs from sensors. These other processes extract information from the LPS and add information back to the LPS as inputs to other behaviors. Even though it is not specific to behaviors, Saphira provides insight into how behaviors benefit from interfacing with systems like memory and sequencing systems of an intelligent agent.

As the controller portion of Saphira, ARIA considers concepts for behaviors such as (de)activation, custom combination of outputs, and limiting execution time. The structure of behaviors in ARIA is simply a list of behaviors that execute with access controlled by activating or deactivating behaviors to add or remove them to that list. This list structure itself does not provide for complex behavior structures similar to behavior trees. However, ARIA mitigates the lack of complexity of its behavior structure by allowing for custom resolution methods that decide how to combine behaviors that affect the same motors based on 'priority' and 'strength' assigned to each. Finally, ARIA attempts to be reactive by imposing a time limit on the execution of the active behavior list [15].

The behavior component of a Saphira is defined in Table 5:

Table 5. Saphira Definition

G=$\{S, O, C, F\}$
S=Name
O=List, outputs combined by Resolvers based on Strength and Priority if executed within time limit
C=Add/Remove from active list
F=*Priority*, *Strength*, *Resolver*, *LPS* (persistent memory)

2.3 Agent Controllers

This section provides a specific look at intelligent agent controllers implementations of intelligent agents. This examination is useful because if one looks at the system of systems view of an intelligent framework then they could overlook the underlying advantages or disadvantages of those structures.

2.3.1 Finite State Machines.

A Finite State Machine (FSM) based framework [4, 3] is another way to provide a controller for an agent. This is done via a series of discrete behaviors, or states, which transition to one another based on perceptions of the world, also called a directed graph [3]. This results in custom transfer functions for each behavior. Similar to behavior trees, a FSM's computational power is limited by the number of states it contains [3]. Adding states to a FSM becomes increasingly difficult if a user wishes to transfer to/from any other state, thus, the size of the FSM influences the implementation effort and the modifiability.

Many people intuitively code FSMs from scratch, even if they don't realize they are creating FSMs [17], because they provide an easy way to conceptualize situations for an intelligent agent. FSMs are compatible with other frameworks. FSMs can be ad-hoc and simple via a few variables which change based on some criteria. Providing tools and commands for FSMs provide ways for individuals to refactor their code into modular blocks with defined structures to them which isn't apparent in ad-hoc FSM implementations. FSMs are another tool available to intelligent agent designers and can be used in conjunction with other frameworks.

The behavior component of a finite state machine is defined in Table 6:

Table 6. Finite State Machine Definition

G=$\{S, O, C, F\}$
S=Name
O=Directed graph of behaviors with transition functions as edges
C=None
F=$Following\_States$, $Transition\_Function$

2.3.2 Behavior Trees.

The Behavior Tree (BT) controller structure is a simple and powerful control structure. Its tree structure provides the advantage of accomplishing very complex behaviors via composition of simple behaviors [5]. The second core concept of BTs is use of nodes to control the execution of child behaviors [3]. This type concept provides useful tools to users in constructing BTs allowing many useful structures to be created. The tree and type concepts are tied together with precondition code blocks to conduct the check of a behavior being successful or not and an execution code block to provide inputs to an agent's motors. With those simple concepts BTs have shown themselves to be a powerful tool in modeling behaviors of intelligent agents [5].

While the type concept is a powerful tool for BTs it is also a limiting factor for them. Through time, various BT implementations have increased the number of node types available. Classically, BTs utilize 'sequential' and 'selector' type nodes [3, 5] which operate by 'selecting' the first node to report success among a set of children or by executing children in sequence until a failure. The node types have been extended to include types 'weighted random', 'parallel', 'priority selector', and 'decorator' to provide additional flexibility and ease of use over that of the two original types. The 'decorator' type changes the success or failure of a child node's pre-condition check in some way [18]. With the original two behavior types or the extended types, the type concept is a strength to BTs but in all implementations users are limited to the types defined by said implementation.

The behavior component of a BT is defined in Table 7:

Table 7. Behavior Tree Definition

G=$\{S, O, C, F\}$
S=Name, Parameters
O=Tree, type nodes determine child execution
C=None
F=*Type* (*Selector*, *Sequence*, *Weighted_Random*, *Parallel*, *Priority_Selector*, *Decorator*)

2.3.3 AFSIM's Intelligent Agent Systems.

In the Advanced Framework for Simulation, Integration, and Modeling (AFSIM) there are multiple sub-frameworks that provide analysts the capability to develop intelligence into their agents; these all may be used independently or in concert with each another [3]. These are the reactive integrated planning architecture (RIPR), messaging systems, commander subordinate structures, generic programming logic, and conceptual ways to interact with the agents. Working together these components of AFSIM each add to the abilities and ease of which an analyst can simulate intelligent agents.

The main component in AFSIM for creating intelligent agents is the RIPR system; this provides the use of behavior trees (BTs), finite state machines (FSMs), a cognitive model, a cluster manager, and a tasking system [19]. The cognitive model simulates the limited mental abilities of a human. The cluster manager provides methods to organize enemies into groups. The tasking system works with the command structure in AFSIM to provide a means to specify a list of tasks or goals to agents. The FSM and BT systems are similar to what was explained in sections 2.3.1 and 2.3.2.

The behavior tree component in AFSIM adds some additional tags over that of generic BTs. These extra tags are mostly to make certain tasks easier and provide explicit means for code to be executed in a given situation. For the most part, these are a convenience; the *update_interval* tag is the only necessary one. Without that command the behavior tree would never be executed. These additional tags in the AFSIM BT implementation provide a small means to relieve coding time for an analyst and allow the frequency of a behavior's execution to be controlled.

The other tools in AFSIM work with and independently of BTs and RIPR. The messaging system provides a means for agent to communicate. The commander subordinate structures allow for decisions and responsibilities to be divided logically. The generic programming logic allows functional programming to be used and executed at desired times or frequencies without the need of using an entire BT or FSM. Generic programming logic is also used within BTs "precondition" and "execute" code blocks; i.e. if-else, for loop, or while loop statements. Finally the AFSIM allows conceptual control of the agents; this allows an analyst to focus on mission level concepts in the BT such as "GoToLocation(xx)" instead of the exact engine settings and exact flap settings to accomplish this. Thus, AFSIM provides many tools above that of just behaviors to assist in the creation of intelligent agents and they can all be used in concert.

The behavior component of an AFSIM behavior is defined in Table 8:

Table 8. AFSIM Behavior Definition

G=$\{S, O, C, F\}$
S=Name
O=Tree, optionally with nested FSM or nested inside an FSM or another BT
C=none
F= $Type(Sequence,\ Selector, Parallel,\ Weighted\_Random,\ Priority\_Selector,$ $node)$, $on\_message$, on_init, $on\_new\_execute$, on_new_fail, $run\_selection$, $update\_interval$, $priority$, and $make\_selection$, $precondition$, $execute$, $behaviortree$

2.3.4 Unified Behavior Framework.

The Unified Behavior Framework (UBF) is an intelligent agent controller which implements capabilities in some interesting ways. Its structure allows for the emergence of behaviors. The definition of the framework allows for dynamic swapping of behaviors. Finally the features of its structure allow for platform independence of behaviors [9].

It enables emergent behavior by flipping a behavior tree's execution of actions upside down; it is tree structure, however the actions are applied by the root instead of the leaf nodes. Hence, action recommendations flow up from leaf nodes to the root. This requires resolution methods to decide how to combine or prioritize one action recommendation over another; which are called "Arbiters" in UBF. These resolution methods are free to be generic or specific to the action recommendation objects that are received; this is in contrast to Subsumption where behaviors communicate by specific inputs, suppression, or inhibition decisions. With the use of action recommendations, arbiters, and actuation occurring in the root of a tree, UBF enables behavior emergence.

The concept of dynamic swapping of behaviors is useful for various reasons but is not always present in UBF implementations. It allows a behavior structure to remain small and hence take less processing time. It allows a behavior structure to interact with other components of an intelligent agent such as planners or sequencers. This concept can be extended to many of the other intelligent agent controllers examined. However, this feature is not always implemented in UBF implementations or other controllers because it is a large endeavor focusing largely on the design of the structure. For UBF this concept has been addressed by defining various tags for a behavior defining its characteristics; this is summarized in Section 2.4.6.

Platform independence in UBF is another concept which is not always implemented. This can be seen by statically structured action recommendations scoped at the platform implemented on [6, 20, 7]. The original definition of UBF presumes platform independence of behaviors via manual mapping of action recommendations to motors in the root [9] and making no assumption of the contents of the action recommendation object. Thus, the action recommendation objects are key to the platform independence of UBF.

UBF also includes an optimization technique to identify behaviors as a leaf or a composite. This technique optimizes the execution of a UBF tree by allowing a leaf behavior to only execute once but its output be re-used at multiple places in the tree [20]. Thus, the optimization is not solely from being a leaf vs composite, but from having a reusable set of action recommendations. This can save needed processor cycles on physical implementations.

The dynamic sequencing aspect of UBF behaviors is examined in Section 2.4.6 and the behavior component of a UBF behavior is defined in Table 9:

Table 9. Unified Behavior Framework Definition

G={S, O, C, F}
S=Name
O=Tree, actions handled by parent arbiter
C=A*dd*/*Remove* Child node
F=*Arbiter*, *Children*, *Priority*, Type: *Leaf* or *Composite*

2.4 Planners and Other Behavior Languages

This section examines other tools that have been created to work with behavior based intelligent agent controllers. This examination is useful because it shows the various methods an underlying behavior based intelligent agent controller interfaces with designers and other software components.

2.4.1 A Behavior Language.

A Behavior Language (ABL) [21] defines various terms into a scripting language for a dynamic version of a behavior tree. ABL adds a level of dynamic execution to a behavior tree by allowing behaviors with the same name be defined and signature matching based on an integer, a behaviors $specificity$, as well as a parameter list. ABL also provides the ability for behaviors to remove or add behaviors to the active behavior tree dynamically at runtime. With the adding/removing of behaviors and an innovative signature matching technique ABL creates a new version of a behavior tree.

In addition to generic BTs ABL adds a few other tags for various functionality not seen in other controllers. First it differentiates behaviors that act on the environment, *Act*, versus behaviors that only calculate something for another behavior to use, a $Mental\_Act$; this is merely a convenient way to label behaviors. ABL's provides commands for creating teams of agents and synchronizing actions between those agents. Finally, ABL provides commands similar reminiscent of a finite state machine which cause behaviors to remove themselves from the active behavior tree based on certain conditions or to never remove themselves from the active tree even if they succeed. These other tags provide explicit teaming, differentiation between environmental actions and mental actions, and commands to modify the behavior tree at runtime.

The behavior component of a ABL behavior is defined in Table 10:

2.4.2 High Level Behavior Based Language.

Vu, et al. [4] at Carnegie Mellon University developed the High Level Behavior based Language (HLBL) in an effort to create a language to share common behaviors across platforms and allow reuse. HLBL is structured as a hierarchical FSM; first

Table 10. **A Behavior Language Definition**

G=$\{S, O, C, F\}$
S=Name, Parameters
O=Tree with Parameter matching
C=*Add/Remove* (not explicitly defined)
F=$Pre-Condition$, $Type$ ($Sequential$, $Parallel$, Act, $Mental\_Act$), Sub_Goal (adds child behavior), $Context\_Condition$ (exit condition), $Synchronize$, $Joint$, $Team$, $Persistent$ (always retry if fail/succeed), $Priority$, $Specificity$

a FSM is constructed and within each behavior (or state) there are children which inherit the parent's exit conditions. This gives a degree of added modularity over generic FSMs. Similar to generic FSMs this approach still suffers from the fact that each behavior needs to handle all possible behavior transitions and each needs to handle all actions (turret aiming, lights on/off, directional control, speed, etc.) that are possible.

HLBL defines various flags that provide convenience over a generic FSM construct. Two of these flags are *initialize* and $finalize$; they call specific code only at the start and only at the end, respectively, of a behavior. Various other flags are used to identify the resolution method needed, lists of following or children behaviors, conditions that must be checked, when condition checks are used at the start or exit of a behavior, and that a function is a resolution function. Finally, a behavior in HLBL has an *Action* flag representing the actual motor settings it may set; however this is optional if the behavior has children. This allows for generic behavior objects to be used as organizational units. Thus, all these tags constitute a language definition for FSMs.

The behavior component of a HLBL behavior is defined in Table 11:

2.4.3 Case Based Behavior Tool.

The Computer-Aided Software (CASE) based tool for behavior generation has a couple of interesting features above that of typical finite state machines (FSMs)

Table 11. High Level Behavior Based Language Definition

G=$\{S, O, C, F\}$
S=Name
O=Hierarchical Finite State Machine with transitions handled by explicitly defined $resolution$ methods
C=None
F=$startswhen$, $endswhen$, $children$, $following$, $ChildResolution$, $FollowingResolution$, $initialize$, $finalize$, $resolution$, $cond$, $choice$, $Action$

[8]. First it uses a graphical user interface in order to build its controller. It uses a hierarchical FSM which allows behaviors to simply be containers for one another. This tool also uses a shorthand notation for declaring the types of nodes, method children nodes are selected, and the repeatability of each node. With these features the CASE tool displays behaviors simply and succinctly.

The behavior component of a CASE tool behavior is defined in Table 12:

Table 12. Computer Aided Tool Behavior Definition

G=$\{S, O, C, F\}$
S=Name
O=Finite State Machine with execution based on node type
C=None
F=|| (AND node), | (OR node), . (sequential node), $*$ (repeat 0 or more), + (repeat at least once), ˜

2.4.4 Unified Behavior Trees Framework for Robot Control.

The Unified Behavior Trees Framework (UBTF) for Robot Control by Marzinotto et al. [22] extends generic behavior tree (BT) frameworks with a couple new node types to support interesting features. One of the features increases the efficiency of a BT by remembering the last child to execute. This leads to UBTF's modified precondition check called a 'current state space configuration check' and the option of returning 'running' instead of simply success or failure from a behavior. UBTF also added some nodes which allow checking of a condition without taking an action and a node to coordinate between team members. UBTF's nodes add to the efficiency and organization options of generic behavior trees.

The behavior component of a UBTF behavior is defined in Table 13:

Table 13. Unified Behavior Trees Framework Definition

G={S, O, C, F}
S=Name
O=Tree structure executed based on parent type
C=None
F=Type (Decorator, Decorator˜, Sequential, Condition, Node * Extended, Action, Selector, Parallel)

2.4.5 STRIPS.

The STanford Research Institute Problem Solver (STRIPS) is a planning program which solves problems by finding a sequence of needed tasks [23, 24]. To do this STRIPS requires task objects to indicate their requirements to be used and the expected effects of using them. This uses first order predicate calculus to solve the problem. Implementing these plans requires a custom sequencer be designed, scoped towards both the framework used as well as the specific implementation, i.e. two behavior trees may only effectively implement dynamic behaviors in certain locations and they may be different locations. Thus, planning programs like STRIPS are compatible with intelligent agent controllers if the building blocks of the controller have the necessary information and have custom sequencing accomplished for them.

In order for a behavior component of an intelligent agent to work with STRIPS it would need to be defined with at least the behavior component definition in Table 14:

Table 14. STRIPS Behavior Definition

G=$\{S, O, C, F\}$
S=$Name$, $Parameters$, $Effects$
O=Requires Custom Sequencer based on framework used
C=Frameworks need to activate/de-activate or add/remove behaviors
F=$Effects$ (including child effects)

2.4.6 Dynamic Behavior Sequencing in UBF.

In a thesis by Duffy [24] work was done to identify the needed components of a UBF behavior for compatibility with a dynamic sequencer. The first component identified is the "initial conditions" which represent the conditions necessary to activate the behavior. The next component is the "post conditions" which identify the effects this behavior adds and removes from the world state. Another component is the "required data" which identifies the sensors or processed data needed for a behavior. The "action settings" component identifies the motors affected in order to allow sequencers to find behaviors by the motors they affect. The "goal achieved" component is used to identify an abstract high level goal a sequencer may look for. Finally, the "vote" component is used to allow a sequencer visibility into the effectiveness of a behavior's "action settings" versus another behavior's. With all of these components a sequencer is given in-depth visibility into a behavior all the requirements a behavior may have and the ways it can affect an environment.

The behavior components of a dynamically sequence-able Duffy behavior is defined in Table 15:

2.5 Summary of Intelligent Agent Commands and Concepts

This section summarizes the commands and concepts of the preceding sections in an effort to reduce the number of synonymous terms, discuss the advantages each

Table 15. Dynamic UBF Behavior Definition

G=$\{S, O, C, F\}$
S=Name
O=Arbitrated tree structure
C=An ability to add/remove from the tree
F=Lists of *initial_conditions*, *Add_Post_Conditions*, *Delete_Post_Conditions*, *Required_Data*, *Action_Setting*, *Goal_Achieved*, *vote*

provides, and start to provide a map from other intelligent agent controllers.

1. Pre_Conditions: A pre_condition code block checks the applicability of a behavior. For behavior trees, this tool provides a small amount of modularity to a user; in many cases it is absorbed into a single code block with action generation. In hybrid finite state machines (FSM) a pre_condition code block conceptually still checks the applicability of a state or behavior; however, this tool allows transfers between states to reuse the logic without every other state knowing the specifics of the receiving state.

2. Priority: Giving behaviors priorities allows for selective execution or selection of behaviors or of a behavior's set of actions. This increases the control a user has when developing a behavior selection or action selection mechanism by providing them a qualitative criteria to work with. Thus, allowing for generic selection mechanisms to be made which do not require specific knowledge of a behavior or action. This reduces the amount of code a user would need to create and increases the potential ways behaviors can be called.

3. Votes: A vote for the action recommendation of a behavior gives criteria, i.e. a weight, by which to merge or select actions. This is slightly different from a behaviors priority which is used to select and identify the behavior which may generate the actions. This also provides a mechanism for generic selection of

action recommendations regardless of their content.

4. Name: Each behavior having a name allows for reuse of the behavior and construction of behavior structures.

5. Expected Effects: A behavior having lists of effects it may add or remove from the environment provides tools for dynamic behavior structure manipulation and planner type code to use. Allowing for planners or dynamic manipulation of a behavior structure can enable the structure to stay both small enough to be reactive and applicable to an intelligent agent's current need.

6. Required Data: A list of the sensors and data components required by a behavior provides additional criteria to a sequencer or planner allowing applicable behavior selection.

7. Action Settings: A list of the motors this behavior affects can provide additional criteria to a sequencer or planner allowing further applicable behavior selection.

8. Initial Conditions: These are a list of environmental conditions that may indicate a behavior is applicable to. This is another tool for a potential sequencer or planner program to use in finding the an applicable behavior.

9. Goal Achieved: This field can identify to a sequencer or planner the abstract goal of a behavior.

10. Behavior Library: A single repository of behaviors provides the capability to dynamically modify a structure and an efficient method to search for behaviors.

11. Parameters: Parameter lists allow for single blocks of code to be used generically. This provides code flexibility, reuse, and can even reduce the risk of errors when a user re-accomplishes the same task multiple times.

12. Action versus Mental Act: Providing an identifier to behaviors indicating if they act on the environment versus only providing calculations for other processes to use is a way to classify behavior types.

13. Global and Persistent Memory: Many behavior implementations or structures do not explicitly define memory as a component; such as ABL and SAPHIRA. Other implementations likely have global and/or persistent memory as a by product of their implementation language (C, C++, Java, C#, etc) and not as a custom grammar laid atop one of those languages. The ability to use memory accessible by other aspects of an intelligent agent (global) allows for communication between behaviors or other processes. Persistent memory in a single behavior allows tracking and more informed decisions to be made on subsequent executions. Ultimately memory unlocks a user's potential to create what ever they can imagine.

14. Action Recommendations: These come in two forms which both have merit; conceptual and motor based. Motor based recommendations are implementation specific rigid objects that contain sub-fields for each motor's setting. This allows for a user to implement a behavior structure without needing to map recommendations to actual outputs; a developer must have already done this for the rigid action object. Conceptual action recommendation objects force the user to map the concept to motor outputs; i.e. "go_left" maps to turn activate left motor for 2 seconds. Conceptual action recommendations increase the work for an analyst, but reduce their reliance on developers when new motor outputs are created. These also allow for generic behavior structures to be created and the only effort an analyst needs to expend is mapping and tuning the action recommendations to motors of the agent it is implemented on.

15. Sub Goals and Children: Giving behaviors sub_behaviors, children behaviors, or sub_goals increases code reuse, increases modularity, increases flexibility, and allows more detailed planning.

16. Reflective Access: A behavior with reflective access is able to modify it's list of children. This allows a behavior to act as a planning or sequencing element and can keep itself lean, reactive, and relevant. With other elements like expected effects lists and a library of behaviors this concept allows behavior structures to be dynamic.

17. Arbitration Methods: Arbiter and resolution methods explicitly allow for emergent behaviors by giving the user control over how actions are chosen and combined. Behaviors may also use reflective access to change the arbiter method used allowing for a change in overall behavior at run time.

18. Signature Matching: This concept is typical in programming languages; by matching parameter lists and method names. Signature matching is another tool that assists with code reuse. ABL extended this to include matching based on a pre_condition block passing. This concept allows for multiple behaviors with identical parameters and names to be created; essentially making each method call a list of potential methods.

19. Previous_Child: Tracking the last child a behavior executed on the previous cycle can increase a framework's efficiency by preventing applicability checks of all the children before it.

20. Exit_Conditions: This is a concept in FSMs which allows them to exit when certain conditions are met. This is a needed tool for state machine structures because their cycles always start in the state of the previous cycle.

21. On_Entry: This block of code is a convenience for users to execute the first time a behavior is executed.

22. On_Exit: This block of code is a convenience for users to execute the first time a behavior doesn't execute when it did execute the previous cycle.

23. Initialization: This block of code is used to initialize variables for a behavior to use. This is a convenient method for users to explicitly set default values and initialize variables.

24. Messaging interface: This type of interface allows external entities to trigger code in a behavior. For AFSIM behaviors, this is a convenient way to separate logic triggered by a message from the execution logic.

25. Synchronous Flags: These flags are used to track who an agent is on a team with and which behaviors need to synchronize between the agents. This alleviates explicit work by users to implement a teaming system. When implemented on an agent a developer would need to map these flags to outputs and map communication inputs to these flags.

26. Frequency: Giving a behavior or behavior structure a frequency allows for tunable efficiency and tunable responsiveness of an agent. This can be useful even in discrete event simulations because those simulations can take hours to compile and can use frequency to only periodically accomplish some complex calculation.

27. Activate/Deactivate: The ability to activate and deactivate behaviors is a reflective tool used to keep behavior structures lean, responsive, and relevant.

28. Execution Time Limit: In SAPHIRA the behavior structure is limited to 100ms in order to maintain the appearance of reactivity.

29. Leaf vs Composite node types: This allows optimizing the execution of behavior structures by storing leaf behavior's outputs for reuse throughout a structure.

III. Unified Behavior Language in AFSIM

Behavioral emergence in the Advanced Framework for Simulation, Integration, and Modeling (AFSIM) requires a different agent modeling capability. To enable this capability the AFSIM script language needs to be enhanced with a new framework in order to access and use it. The Unified Behavior Framework (UBF) is able to accomplish this goal. To show the methodology used to create a behavior language, which provides the capability of emergent behaviors in AFSIM, requires multiple components.

The first component is an understanding of the implementation of UBF in AFSIM. The next component is a map of the concepts seen in other intelligent agent controllers to their implementation, or lack thereof, in AFSIM and the UBF structure added to AFSIM. The final component required is the syntax definition for each term added to AFSIM. With an understanding of how UBF was implemented in AFSIM, a mapping of concepts in the intelligent agent community to AFSIM and the new behavior plug-in, and the required syntax for all added components, a reader has the tools required in order to recreate this script language extension in their coding environment of choice.

3.1 Unified Behavior Framework in the Advanced Framework for Simulation, Integration, and Modeling

In order to understand the implementation there are two main areas to investigate. First the underlying class structure is examined, how the AFSIM code interacts with this class structure, and how this implementation relates to the original conceptual class structure for UBF. Next the flow control versus the flow of information is examined. With these two components a reader can create the underlying structure of UBF as it was implemented in this thesis.

3.1.1 UBF Class Structure.

The first step to understand how UBF is implemented in AFSIM is to understand the underlying class structure versus the notional class structure. Figure 3 utilizes the Unified Modeling Language (UML) class diagram [25] standard to display the structure of UBF implemented in AFSIM; these classes are C++. This is compiled into a dynamic link library (DLL) and used as a plug-in, that can parse and compile the script into a replay file; this plug-in is used by AFSIM executables.

Figure 3 displays two occurrences of multiple inheritance where UBFBehaviors and UBFArbiters both inherit from WsfProcessor and UBFActionList. Inheriting from WsfProcessor allows UBFBehavior and UBFArbiter objects to register themselves with a WsfProcessor factory, a library of WsfProcessors, and allows the top level UBFBehavior to be called upon as if it were a WsfProcessor. Registering with the WsfProcessor factory allows dynamic and runtime referencing of UBFBehaviors and UBFArbiters. Inheriting from UBFActionList provides a vector of UBFAction objects and the necessary methods to access the UBFActions in a UBFBehavior or UBFArbiter. Thus, this multiple inheritance allows runtime access to UBFBehaviors that are created and similar functionality for working with the UBFAction objects.

The class diagram in Figure 3 utilizes some functions that may not be straight forward to non-AFSIM developers. The *ProcessInput* function instantiates an instance of the class, adds that instance to the WsfProcessor factory, and parses the input stream to store or set values in that object. The *Initialize* function associates all of the pointer objects; here stored values from the *ProcessInput* function call are used to find and store references to objects throughout the application and build the UBF tree of references to other UBFBehaviors and UBFArbiters. The *mExecute(...)*

function is called by the AFSIM application if the UBFBehavior is the root or by the parent UBFBehavior and controls executing script code blocks defined by the user. Within those three methods the entire construction and execution of UBF is possible.

The new class diagram in Figure 3 has three large differences versus the original notional class diagram for UBF seen Figure 2. The first is the inclusion of two classes for action objects. The original UBF definition omits this because the definition of an action object is dependent on the application environment; however, this thesis proposes generic action objects be used to enable greater reuse and platform independence. Hence, why those classes are included in Figure 3. The UBFActionList class was created as a convenience and to use the programming practice of inheritance for code reuse. The methods inside UBFActionList provide a variety of ways for an analyst to safely and conveniently access the actions stored in a UBFActionList object.

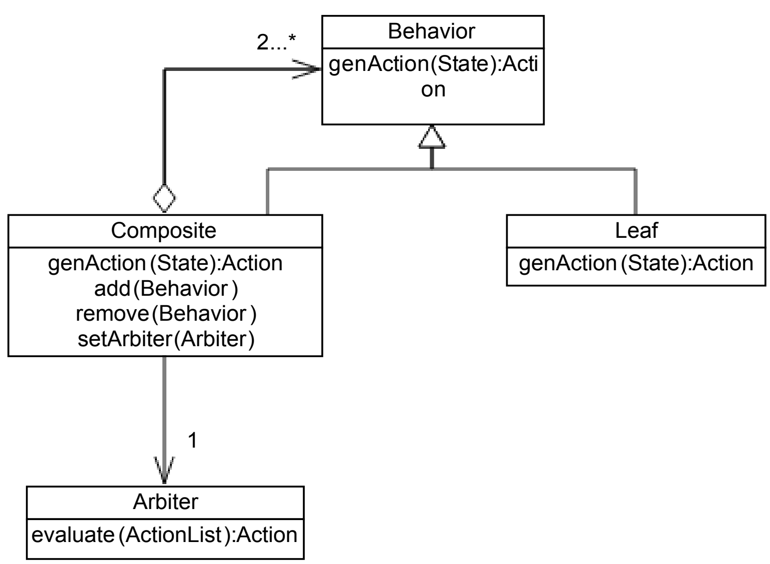


Figure 2. Notional UBF Class Structure.

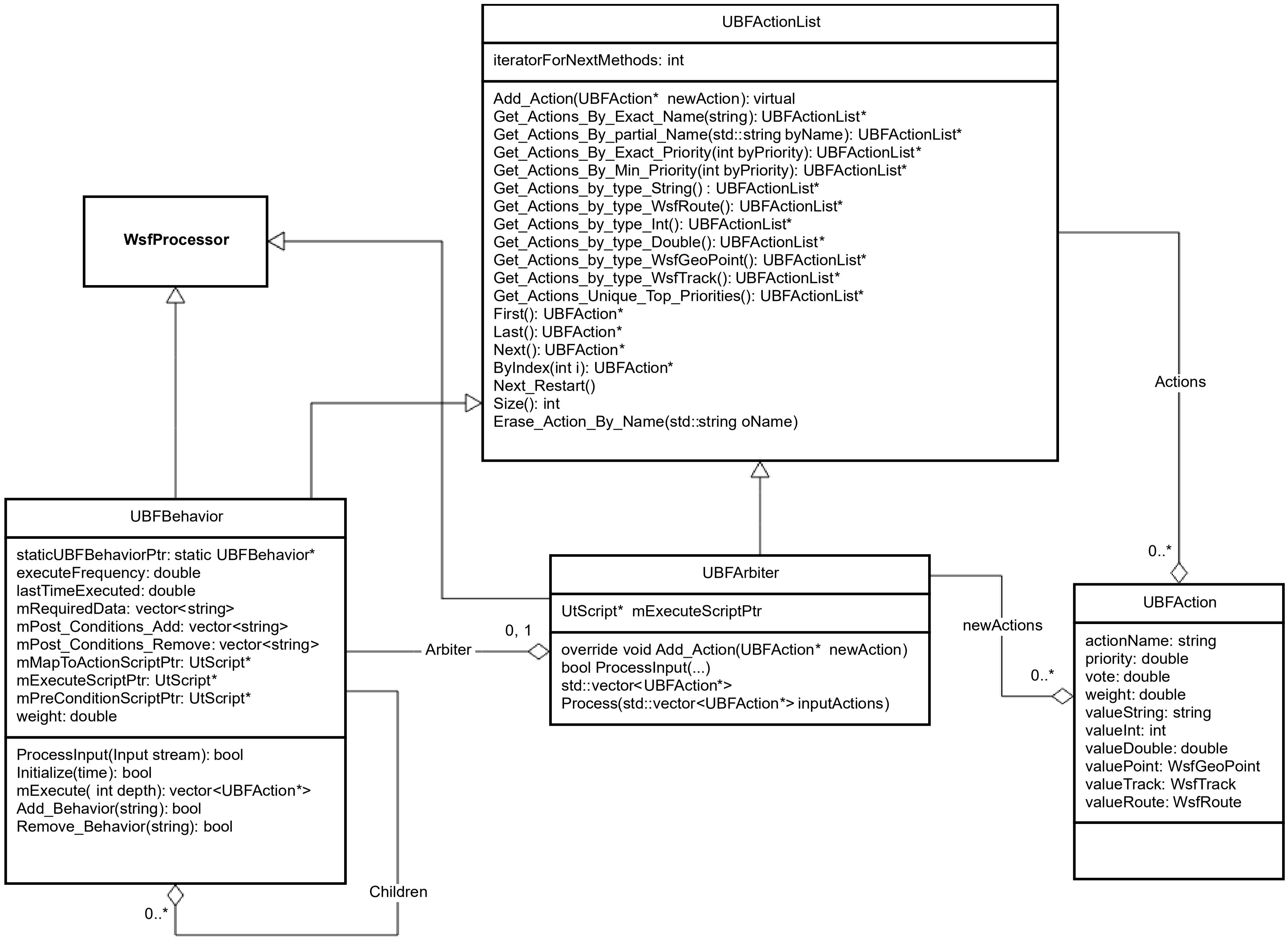


Figure 3. AFSIM UBF Class Structure.

The other large differences versus the original notional class diagram are the lack of a composite or leaf behavior distinction and the lack of a "State" variable being passed. The lack of a composite versus leaf behavior distinction is simplified into a generic behavior class because all behaviors may be composites; gaining this simplicity does cause a loss of potential optimization from automatically reusing behavior outputs if a behavior is reused in a single tree. A "State" variable is omitted in the new implementation because code blocks are automatically given access to a "PLATFORM" variable. This variable provides analysts the ability to view the state of their agent and its sensors. Being forced to use a "state" variable limits users to the sensors and components that exist at the time that "state" variable is developed. With the explanation of these three differences a user should understand the necessity for the differences and how similar the two are.

3.1.2 UBF Data Flow Chart.

The second step in understanding the UBF implementation in AFSIM is to understand the flow of data within a UBFBehavior and a UBF tree. This encompasses the order code segments are executed, the path action recommendations take through a behavior, and the effect of omitting code blocks. In order to do this Figure 4 presents the standard shapes and colors used for the various code blocks within a UBFBehavior. Using those standards, Figures 5 and 6 display the flow of data accompanied by a discussion summarizing them. With the understanding from these flow charts a user can better understand the avenues for inter-behavior communication and expose the power of UBF.

In the broadest sense all of the components presented are optional. This means that if any or all components are omitted the UBF tree will still compile; the user may be presented with warnings in the console output and the tree may not function

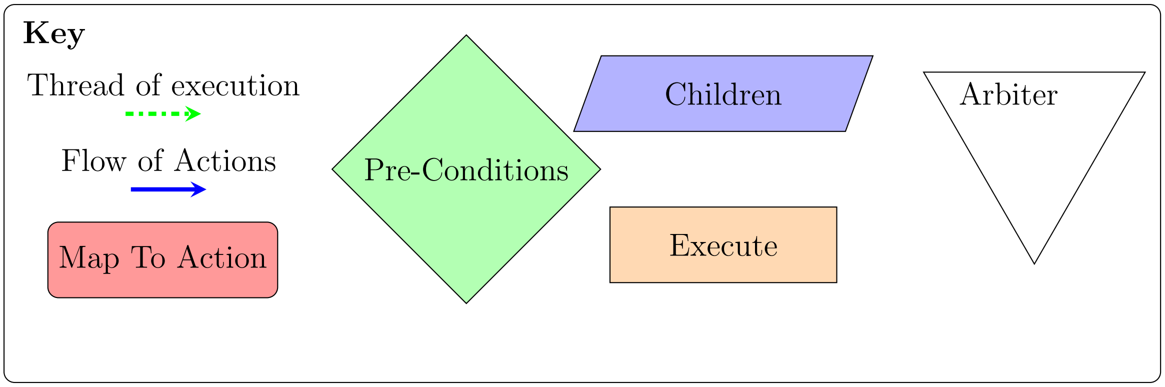


Figure 4. UBFBehavior Key.

as expected. The only code block that is conceptually required is the Map_To_Action code block since that is where the action recommendations are supposed to affect the environment; without the Map_To_Action code block a UBF tree should not affect anything. Omitting the Pre_Conditions code block results in the thread of execution following the *True* path. Omitting the Arbiter code block results in any action recommendations passing to the Map_To_Action code block or parent behavior un-affected; if an Arbiter is included an action recommendation must be explicitly passed forward or it is discarded. Omitting the Execute code block simply passes the thread of control and any action recommendations from the children forward; including an Execute code block does not explicitly stop action recommendations from children passing forward, but the Execute block does have access to modify or delete those recommendations if desired.

3.2 Mapping of Commands and Concepts to AFSIM Environment

This subsection maps the concepts explored in Section 2.5 to the AFSIM environment and the additions made to it. Commands, also known as tags, in **bold** are native to AFSIM and commands <u>underlined</u> were added to AFSIM as part of this thesis effort. The AFSIM environment can run in both discrete event mode (simulation

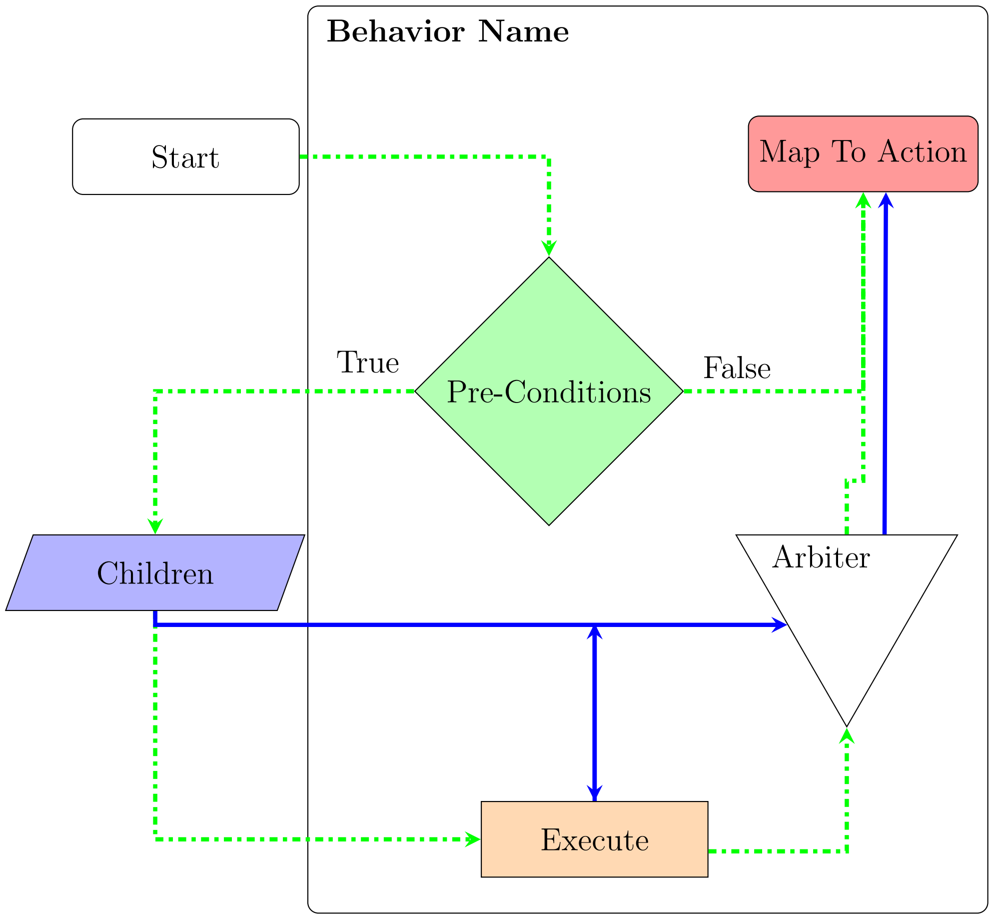


Figure 5. Root behavior flow chart.

is compiled into a replay file for later recall) and as a real time simulation; this implementation is designed towards the discrete event mode, however Section 6.2 contains comments on what code should be updated for real time use. This implementation utilizes UBF as the basis for the structure being built.

1. Pre_Conditions: A pre_condition code tag is used to implement this. While in BTs it is not strictly necessary, in this UBF implementation it provides an efficiency increase via a failed pre_condition code block that prevents children behaviors from executing and the behavior itself from executing.

2. Priority: A priority field is implemented as a component of each action object. Arbiter object use this field to select sets of action objects from behaviors, this

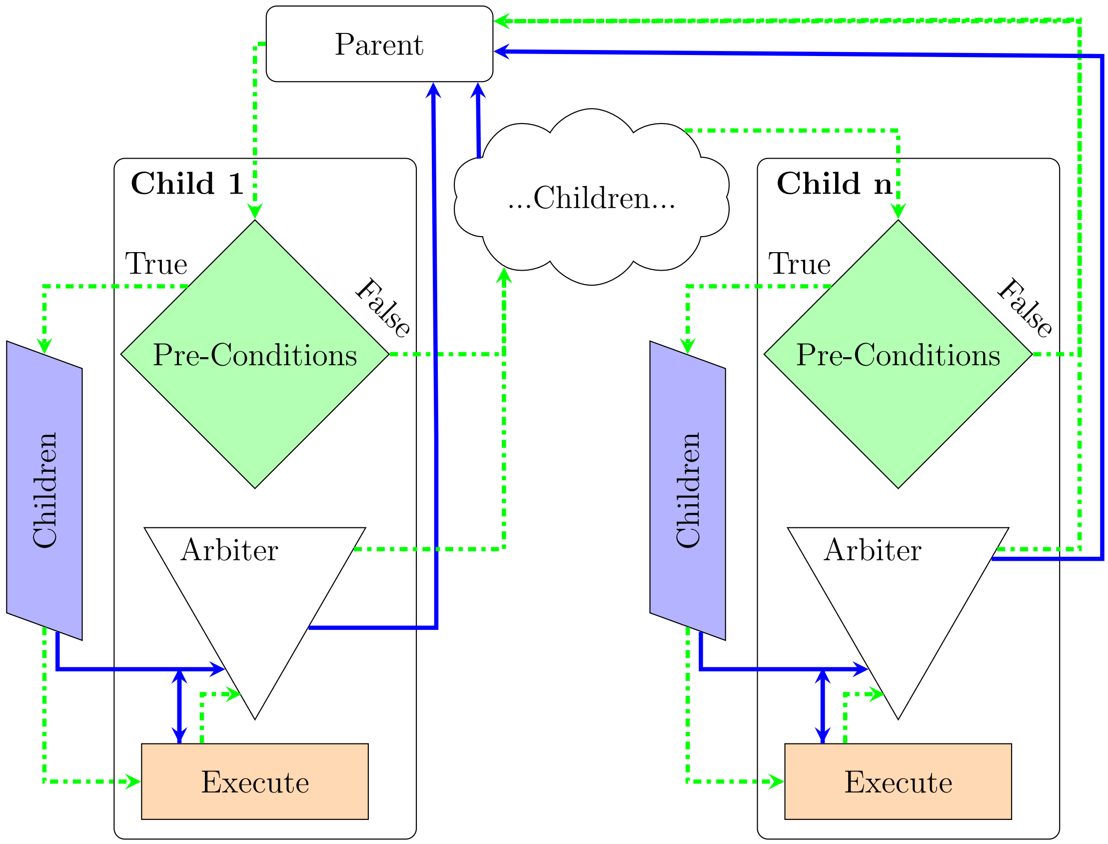


Figure 6. Children behavior flow chart.

implementation uses smaller non-negative numbers for a ‘higher’ vote.

3. Votes: A vote field is implemented as a component of each action object. Arbiter objects use this field to select between and how to merge action objects, this implementation uses larger numbers for a ‘higher’ vote.

4. Name: The **name** of each behavior is built into AFSIM because the AFSIM processors have names. This is used to go through the library of AFSIM processors and retrieve by name the UBFBehaviors and UBFArbiters desired.

5. Expected Effects: This implementation has two tags Add_Post_Condition and Remove_Post_Condition that add strings to a vector, a list, in every UBFBehavior;. Commands are implemented to access the list as well.

6. Required Data: This implementation has the tag Required_Data that adds a string to a list that is part of every UBFBehavior indicating that sensors and data may be required by that behavior; various commands are implemented to access the list as well.

7. Action Settings: An Action_Setting tag is used to add strings to a list indicating which motors a behavior affects; various commands are implemented to access the list as well.

8. Initial Conditions: An Initial_Condition tag is used to add strings to a list indicating which conditions are required by a behavior to activate; various commands are implemented to access the list as well.

9. Goal Achieved: A Goal_Achieved is a string subfield indicating the overall abstract goal a behavior is applicable towards; various commands are implemented to access the list as well.

10. Behavior Library: AFSIM has built in factories, libraries, for the components of agents. The UBFBehavior and UBFArbiter classes inherit from the WsfProcessor class making them compatible with the WsfProcessor factory. Thus, the processor factory is a behavior library. The processor type is used because that type is used for "thought process" like activities.

11. Parameters: This is not explicitly implemented; each behavior only has a name with no list of parameters after it. A workaround to this is via the communication between behaviors the UBF structure allows. When constructing the UBF tree, parameters can be the children of a behavior communicating the parameter information in the form of action objects. Section 6.2 talks about a way to extend this implementation to allow for parameter passing close to what a programmer may see as command line arguments in the C programming language.

12. Action versus Mental Act: This differentiation is not made in this implementation. This is because only the root of the UBF tree actuates on the environment and classifying this is not desired for this implementation because it adds to the complexity without notable benefits.

13. Global and Persistent Memory: These concepts are accomplished through built in features of AFSIM. Persistent memory for each UBFBehavior is accomplished via usage of AFSIM's **script_variables** tags. These variables are usable in all code blocks within a behavior but are not accessible by children behaviors or by any external code. These variables retain their value between successive calls to a behavior. A way that AFSIM implements global variables is by the use of the **Aux_Data** tag. This tag attaches to components in AFSIM and is accessible between components.

14. Action Recommendations: This implementation's action objects are conceptual. This results in a need for the root behavior to have an analyst implement a Map_To_Action code block. Each action object has name, priority, vote, integer value, string value, and double value fields. These fields are usable at the discretion of the UBF tree designer. Each UBFBehavior is not limited on the number of action recommendation it may produce.

15. Sub Goals and Children: Child behaviors are able to be added to each behavior by the Children tag. This tag starts a list of behavior names that are added to the behavior in which the tag is contained. Each name is preceded by a Behavior tag indicating a name follows that tag. Also, the Children tag can be used within itself to construct an entire tree of UBFBehaviors.

16. Reflective Access: This is accomplished by Add_Behavior and Remove_Behavior commands being added to the script language for use in a UBFBehavior's code

blocks. This is the extent of the dynamic access provided in this implementation.

17. Arbitration Methods: The Arbiter tag is used to indicate the name of an arbiter assigned to a behavior. There were no commands created to dynamically change the arbiter used on a behavior; Section 6.2 proposes such commands. Creating an UBFArbiter is done similar to defining a UBFBehavior, as a processor of type UBFArbiter with an execute code block within it.

18. Signature Matching: The signature of a behavior in this implementation is only based on the name. Section 6.2 suggests possible ways to implement parameters into the signature matching.

19. Previous_Child: This is not implemented because all children in a UBF tree conceptually execute every iteration.

20. Exit_Conditions: Since this is a concept for FSMs it is not applicable to the UBF tree structure.

21. On_Entry: This is not implemented because it is a convenience and does not align with UBF trees executing every behavior every iteration.

22. On_Exit: This is not implemented because it is a convenience and does not align with UBF trees executing every behavior every iteration.

23. Init: This is accomplished by **script_variables** tag in a UBFBehavior or UBFArbiter where a user may define variables with initial values that are usable throughout each respectively.

24. Messaging interface: This is not implemented since it is a convenience for users. A user may do this themselves if desired.

25. Synchronous Flags: The concept of teaming is not implemented explicitly because AFSIM provides the **side** tag indicating a team the platform is on and includes a messaging system based on a commander subordinate relationship.

26. Frequency: The **update_interval** command built into AFSIM processors controls the frequency an agent's UBF tree is executed; this is required for a UBF tree to execute. The frequency tag is used to control how often a child behavior may execute.

27. Activate/Deactivate: This is partially implemented. The responsibility of keeping a UBF tree lean, responsive, and relevant falls on the analyst using the Add_Behavior and Remove_Behavior commands in each behavior. This capability could be in an extension of this thesis.

28. Execution Time Limit: This is not implemented as a limiting factor because the focus is on discrete event based implementation. However, the tag debug_time is implemented to give analysts further ability tuning their UBF tree. This command prints the time a behavior takes to execute each of its code blocks in case an analyst needs to reduce the time taken to compute a simulations output.

29. Leaf vs Composite node types: This is not implemented due to the scope being towards non-real time execution. Section 6.2 presents a possible implementation for this.

3.3 Manual pages for new AFSIM commands

This section is meant to be a reference for those implementing UBF in AFSIM and familiarize readers with the grammar of UBFBehaviors and UBFArbiters. To do this, the semantics of various similar terms is defined to prevent ambiguity and provide clarity to the definitions. Example scripts are provided in Figures 7 and 8 showing full examples from the AFSIM integrated development environment. Finally definitions are provided for every command and tag created by this thesis.

```
processor UBFArbiterName UBFArbiter
#Comment indicating proper use of this Arbiter
   script_variables #Tag indicating script_variables Code Block
        #commands
   end_script_variables

   Execute #Tag indicating Execute Code Block
      #commands
      UBFAction a;
      a=UBFArbiter.Get_First_Action();
      if(a.Get_Int()==1)
         UBFArbiter.Add_Action(a);
   end_Execute
end_processor
```

Figure 7. Example Script of UBFArbiter.

The semantically similar terms are "commands", "tags", "code blocks", and "script." AFSIM uses its own language which its analysts use to create scenario files for the AFSIM executable to compile into a replay file. The generic use of text files analysts use are referred to as scripts, AFSIM script, or the AFSIM scripting language. This is in contrast to the term "code block." "Code block(s)" refer to a specific subset of the AFSIM scripting language. In these code blocks generic programming logic is used; such as, if-then statements, for loops, while loops, etc. The logic inside a code block is referred to as "command(s)". The set of commands may be augmented, hence when a "command" is created it is usable within any "code block." "Tags" are used to indicate the start and end of code blocks, flags, switches, variables, or a developer designed purpose. A tag does not have to be a code block but all code blocks are encompassed by tags. Figure 8 shows a basic behavior script in AFSIM and Figure 7 shows an example AFSIM script for a UBFArbiter object to help readers associate the semantics of these terms with their implementation.

```
processor BehaviorNameHere UBFBehavior
#Comments indicating proper usage of this behavior

   #Tag indicating the update interval
   update_interval 10 sec
   #Tag indicating Arbiter name
   Arbiter ArbiterName
   #Tag indicating frequency in seconds
   Frequency 11
   #Flag indicating time to run will be printed
   Debug_Time

   #Tag indicating list of Children to follow
   Children
      Behavior BehaviorNameB
      Behavior BehaviorNameC
         Children
            Behavior BehaviorNameD
         end_Children
   end_Children

   #Tag indicating script_variables Code Block
   script_variables
      #commands
   end_script_variables

   Pre_Condition #Tag indicating Pre_condition Code Block
      #commands
   end_Pre_Condition

   Execute #Tag indicating Execute Code Block
      #commands
      while(true)
      {
         if(true){
               #do something
               }
      }
      #Custom added commands
      UBFAction a=UBFAction.Create("name",1, "value");
      UBFBehavior.Add_Action(a);
   end_Execute

   Map_To_Action  #Tag indicating Map_To_Action Code Block
      #commands
   end_Map_To_Action
end_processor
```

Figure 8. Example Script of UBFBehavior.

3.3.1 Tag Documentation.

UBFBehavior Tags.

The tags used within a UBFBehavior indicate the start or end of code blocks and values that are associated with a UBFBehavior's fields. The tags that are usable in a UBFBehavior are:

Processor type: **UBFBehavior**
Scope: Top level AFSIM script tag or within a platform type
Description: This keyword is used to indicate the type of processor as a Behavior
Number allowed: no limit to number of UBFBehaviors, only one should be used directly in a platform type
Example Usage:

```
processor <name> UBFBehavior
   #sub-tags...
end_Processor
```

Tag Name: **update_interval**
Scope: Tag within UBFBehavior processor, only used in root behavior
Description: This tag indicates the frequency with which the UBFBehavior tree executes; pseudo optional, if omitted in root UBFBehavior the tree will never execute and it has no effect if implemented in a child
Number Allowed: 0 or 1
Example Usage:

```
   update_interval <time amount> <time unit>
```

Tag Name: **Frequency**
Scope: Tag within UBFBehavior processor, only used in child behaviors
Description: This tag indicates the frequency with which this UBFBehavior executes, if omitted UBFBehavior executes whenever its parent calls it, never used in the root
Number Allowed: 0 or 1
Example Usage:

```
  Frequency <int>
```

Tag Name: **Debug_Time**
Scope: Tag within UBFBehavior processor

Description: This tag is a flag which causes the time each code block in a UBFBehavior takes to run to be printed to the output console in the AFSIM integrated development environemnt
Number Allowed: 0 or 1
Example Usage:

```
Debug_Time
```

Tag Name: **Arbiter**
Scope: Tag within UBFBehavior processor
Description: This tag indicates the name of a UBFArbiter process to be used as an arbiter for this behavior
Optional: Optional; If omitted all UBFAction objects provided to the UBFBehavior object will automatically be sent to the parent UBFBehavior or Map_To_Action code block respectively
Number Allowed: 0 or 1
Example Usage:

```
Arbiter <name>
```

Tag Name: **Children**
Scope: Tag within UBFBehavior processor
Description: This tag indicates the UBF tree structure associated with the associated UBFBehavior; may be nested within itself defining children of children with a limited depth of 30, if omitted then a UBFBehavior simply will not be instantiated with children. Children may be added later via commands
Number Allowed: 0 or 1
Example Usage:

```
Children
   {Behavior <behavior name>}*
end_Children
```

Tag Name: **script_variables**
Scope: Tag within UBFBehavior processor
Description: This code block defines variables usable through all other code blocks of the UBFBehavior with which it is associated, if omitted the UBFBehavior will not have variables that persist between iterations or are shared between its code blocks
Number Allowed: 0 or 1
Return Type: No return type allowed
Example Usage:

```
script_variables
   int defaultSpeed=100;
end_script_variables
```

Tag Name: **Pre_Condition**
Scope: Tag within UBFBehavior processor
Description: This tag defines a code block which executes immediately when a UBFBehavior is executed; if false is returned the UBFBehavior immediately cedes control to the parent or Map_To_Action block with no UBFAction objects being provided; if true the UBFBehavior continues to execute, if omitted a value of true is assumed
Number Allowed: 0 or 1
Return Type: Boolean
Example Usage:

```
Pre_Condition
   #commands
end_Pre_Condition
```

Tag Name: **Execute**
Scope: Tag within UBFBehavior processor
Description: This code block provides the logic which outputs UBFAction objects, if omitted control passes directly from children to Arbiter code block
Number Allowed: 0 or 1
Return Type: none; UBFAction objects which are output are added via explicit commands not via return keyword
Example Usage:

```
Execute
   #commands
end_Execute
```

Tag Name: **Map_To_Action**
Scope: Tag within UBFBehavior processor
Description: This code block provides the logic which reads the UBFAction objects and maps them to commands that affect the environment and/or platform with which the UBF behavior is associated; pseudo optional; if omitted from root UBFBehavior then the tree may have no effect on the environment or platform; if in child UBFBehaviors it will never be executed
Number Allowed: 0 or 1
Return Type: none
Example Usage:

```
Map_To_Action
   #commands
end_Map_To_Action
```

Tag Name: **Add_Post_Condition**
Scope: Tag within UBFBehavior processor
Description: This tag allows users to add a single string to the post condition adder set of a UBFBehavior indicating an effect on the world it expects to have
Number Allowed: 0 or more
Return Type: none
Example Usage:

```
Add_Post_Condition "OpenBombDoors"
```

Tag Name: **Remove_Post_Condition**
Scope: Tag within UBFBehavior processor
Description: This tag allows users to add a single string to the post condition remove set of a UBFBehavior indicating an effect it expects to compensate for
Number Allowed: 0 or more
Return Type: none
Example Usage:

```
Remove_Post_Condition "OpenBombDoors"
```

Tag Name: **Action_Setting**
Scope: Tag within UBFBehavior processor
Description: This tag allows users to add a single string to the action settings set of a UBFBehavior indicating the motors it affects
Number Allowed: 0 or more
Return Type: none
Example Usage:

```
Action_Setting "UHFJammer"
```

Tag Name: **Required_Data**
Scope: Tag within UBFBehavior processor
Description: This tag allows users to add a single string to the required data set of a UBFBehavior indicating the sensors or preprocessed data the UBFBehavior needs to operate
Number Allowed: 0 or more
Return Type: none
Example Usage:

```
Required_Data "UHF_Radar"
```

Tag Name: **Goal_Achieved**
Scope: Tag within UBFBehavior processor
Description: This tag allows users to set the string value for the abstract goal a UBFBehavior is supposed to achieve
Number Allowed: 0 or 1
Return Type: none
Example Usage:

```
Goal_Achieved "FlyHome"
```

Tag Name: **Initial_Condition**
Scope: Tag within UBFBehavior processor
Description: This tag allows users to add a single string to the initial conditions set of a UBFBehavior indicating the conditions required to activate this UBFBehavior
Number Allowed: 0 or more
Return Type: none
Example Usage:

```
Initial_Conditions "EnemyInRange"
```

UBFBehavior Commands.

Commands are added to the code blocks of the UBFBehavior object to provide reflective access to the UBFBehavior objects. These commands are:

Command Name: **.Find(string)**
Scope: Method of the UBFBehavior class used via dot operator
Description: This command finds a UBFBehavior by name
Parameters: string of the name of the behavior to find
Returned Object: UBFBehavior
Example Usage:

```
UBFBehavior <behaviorName> = UBFBehavior.Find("FlyTo");
```

Command Name: **.Remove_Behavior(string)**
Scope: Method of the UBFBehavior class used via dot operator
Description: This command finds a UBFBehavior by name in the objects list of children behaviors and removes it from the list
Parameters: string of the name of the behavior to find
Returned Object: bool representing success or failure

Example Usages:

```
if(UBFBehavior.Remove_Behavior("FlyTo"))

if(<behaviorName>.Remove_Behavior("FlyTo"))
```

Command Name: **.Add_Behavior(string)**
Command Name: **.Add_Behavior(UBFBehavior)**
Scope: Method of the UBFBehavior class used via dot operator
Description: This command finds a UBFBehavior by name or takes another UBFBehavior pointer and adds it to the object in questions children list
Parameters: string or a UBFBehavior to be added
Returned Object: bool representing success or failure
Example Usages:

```
if(UBFBehavior.Add_Behavior("FlyTo"))

if(<behaviorName>.Add_Behavior("FlyTo"))
```

Command Name: **.Add_Adder_Post_Condition(string)**
Command Name: **.Add_Remove_Post_Condition(string)**
Command Name: **.Add_Add_Action_Setting(string)**
Command Name: **.Add_Add_Required_Data(string)**
Command Name: **.Add_Initial_Condition(string)**
Scope: Methods of the UBFBehavior class used via dot operator
Description: These commands add strings to their associated lists
Parameters: string to be added
Returned Object: n/a
Example Usages:

```
UBFBehavior.Add_Adder_Post_Condition("Close Bomb Doors"));
UBFBehavior.Add_Remove_Post_Condition("Open Bomb Doors"));
<behaviorName>.Add_Add_Action_Setting("Bomb Doors"));
UBFBehavior.Add_Add_Required_Data("10kHz Laser Range Finder"));
UBFBehavior.Add_Initial_Condition("In Florida"));
```

Command Name: **.Adder_Post_Condition_Exists(string)**
Command Name: **.Remove_Post_Condition_Exists(string)**
Command Name: **.Action_Setting_Exists(string)**

Command Name: **.Required_Data_Exists(string)**
Command Name: **.Initial_Condition_Exists(string)**
Scope: Methods of the UBFBehavior class used via dot operator
Description: These commands determine if a string exists in their respective lists
Parameters: string to be searched for
Returned Object: bool showing success if the string was found or not
Example Usages:

```
if(<behaviorName>.Adder_Post_Condition_Exists("Close Bomb Doors"))
if(UBFBehavior.Remove_Post_Condition_Exists("Open Bomb Doors"))
if(UBFBehavior.Action_Setting_Exists("Bomb Doors"))
if(UBFBehavior.Required_Data_Exists("10kHz Laser Range Finder"))
if(UBFBehavior.Initial_Condition_Exists("In Florida"))
```

Command Name: **.Set_GoalAchieved(string)**
Scope: Method of the UBFBehavior class used via dot operator
Description: This command sets the Goal Achieved variable to a specified string
Parameters: string to be set
Returned Object: n/a
Example Usages:

```
if(UBFBehavior.Set_GoalAchieved("Navigates Home"))
```

Command Name: **.Get_GoalAchieved()**
Scope: Method of the UBFBehavior class used via dot operator
Description: This command gets the Goal Achieved variable
Parameters:
Returned Object: string of the goal achieved which may be empty
Example Usage:

```
string temp=UBFBehavior.Get_GoalAchieved();
```

Command Name: **.Get_Adder_Post_Condition_byIndex(int)**
Command Name: **.Get_Remove_Post_Condition_byIndex(int)**
Command Name: **.Get_Action_Setting_byIndex(int)**
Command Name: **.Get_Required_Data_byIndex(int)**
Command Name: **.Get_Initial_Condition_byIndex(int)**
Scope: Methods of the UBFBehavior class used via dot operator
Description: These commands return the string at the index of the respective list or "DNE" if there is no item there

Parameters: integer of the index of the list desired
Returned Object: string of the variable at the indexed location of the respective list
Example Usages:

```
string temp=UBFBehavior.Get_Adder_Post_Condition_byIndex(1);
string temp=<behaviorName>.Get_Remove_Post_Condition_byIndex(1);
string temp=UBFBehavior.Get_Action_Setting_byIndex(1);
string temp=UBFBehavior.Get_Required_Data_byIndex(1);
string temp=UBFBehavior.Get_Initial_Condition_byIndex(1);
```

Command Name: **.Adder_Post_Condition_Size()**
Command Name: **.Remove_Post_Condition_Size()**
Command Name: **.Action_Setting_Size()**
Command Name: **.Required_Data_Size()**
Command Name: **.Initial_Condition_Size()**
Scope: Methods of the UBFBehavior class used via dot operator
Description: These commands return the size of their respective lists
Parameters: none
Returned Object: integer indicating the size of the respective list
Example Usages:

```
int temp=UBFBehavior.Adder_Post_Condition_Size();
int temp=UBFBehavior.Remove_Post_Condition_Size();
int temp=<behaviorName>.Action_Setting_Size();
int temp=UBFBehavior.Required_Data_Size();
int temp=<behaviorName>.Initial_Condition_Size();
```

UBFArbiter Tags.

The tags used within a UBFArbiter indicate the start or end of the code blocks.

The tags that are usable in a UBFArbiter are:

Processor type: **UBFArbiter**
Scope: Top level AFSIM script only
Description: This keyword is used to indicate the type of processor as an Arbiter which may later be referenced to by a UBFBehavior
Number allowed: no limit to number of UBFArbiters
Example Usage:

```
processor <name> UBFArbiter
   #sub-tags...
end_Processor
```

Tag Name: **script_variables**
Scope: Tag within UBFArbiter processor
Description: This code block defines variables usable through all other code blocks of the UBFArbiter with which it is associated
Number Allowed: 0 or 1
Return Type: No return type allowed
Example Usage:

```
script_variables
   int defaultSpeed=100;
end_script_variables
```

Tag Name: **Execute**
Scope: Tag within UBFArbiter processor
Description: This code block provides the logic which may process input UBFActions via UBFArbiter.Get... commands and output UBFActions via UBFArbiter.Add_Action(...)
Number Allowed: 1
Optional: No; if omitted, no UBFAction objects will pass through, all UBFActions input will be discarded
Return Type: none; UBFAction objects which are output are added via explicit commands not via return keyword
Example Usage:

```
Execute
   #commands
end_Execute
```

Commands - UBFActionList.

Commands are added to code blocks that expose the UBFActionList object type and its associated functions. These commands are inherited and usable to modify the UBFActionList objects inherited by UBFBehavior and UBFArbiter objects. These commands are:

Object type: **UBFActionList**
Scope: usable within UBFArbiter code blocks and UBFBehavior code blocks
Description: This object may be instantiated on its own; it is used as the default storage device for outputting UBFActions in UBFBehaviors; it is used as the default storage devices for inputting and outputting UBFActions in UBFArbiters

Example Usages:
Declare UBFActionList variable:

```
UBFActionList <listName>;
```

Command Name: **.Create()**
Scope: Method of the UBFActionList class used via dot operator
Description: This command instantiates a UBFActionList object
Parameters: None
Returned Object: UBFActionList
Example Usage:
Instantiate UBFActionList variable:

```
UBFActionList <listName> = UBFActionList.Create();
```

Command Name: **.Get_Action_By_Index(int)**
Scope: Method of a UBFActionList object used via dot operator
Description: This command retrieves a UBFAction by its index in a UBFActionList object; if index out of bounds then a null object pointer is returned
Parameters: Integer representing index of a UBFAction
Returned Object: UBFAction
Example Usages:
UBFActionList object:

```
UBFAction actionA = <listName>.Get_Action_By_Index(5);
```

UBFBehavior storage:

```
UBFAction actionA = UBFBehavior.Get_Action_By_Index(5);
```

UBFArbiter input:

```
UBFAction actionA = UBFArbiter.Get_Action_By_Index(5);
```

Command Name: **.Erase_Action_By_Name(string)**
Scope: Method of a UBFActionList object used via dot operator
Description: This command finds and removes the first UBFAction by the name supplied
Parameters: Integer representing index of a UBFAction
Returned Object: UBFAction
Example Usages:
UBFActionList object:

```
UBFAction actionA = <listName>.Erase_Action_By_Name("fly");
```

UBFBehavior storage:

```
UBFAction actionA = UBFBehavior.Erase_Action_By_Name("fly");
```

UBFArbiter input:

```
UBFAction actionA = UBFArbiter.Erase_Action_By_Name("fly");
```

Command Name: **.Get_First_Action()**
Scope: Method of a UBFActionList object used via dot operator
Description: This command retrieves the first UBFAction in the UBFActionList object; if empty a null object pointer is returned
Parameters: None
Returned Object: UBFAction
Example Usages:
UBFActionList object:

```
UBFAction actionA = <listName>.Get_First_Action();
```

UBFBehavior storage:

```
UBFAction actionA = UBFBehavior.Get_First_Action();
```

UBFArbiter input:

```
UBFAction actionA = UBFArbiter.Get_First_Action();
```

Command Name: **.Get_Last_Action()**
Scope: Method of a UBFActionList object used via dot operator
Description: This command retrieves the last UBFAction in the UBFActionList object; if empty a null object pointer is returned
Parameters: None
Returned Object: UBFAction
Example Usages:
UBFActionList object:

```
UBFAction actionA = <listName>.Get_Last_Action();
```

UBFBehavior storage:

```
UBFAction actionA = UBFBehavior.Get_Last_Action();
```

UBFArbiter input:

```
UBFAction actionA = UBFArbiter.Get_Last_Action();
```

Command Name: **.Get_Next_Action()**
Scope: Method of a UBFActionList object used via dot operator
Description: This command retrieves the next UBFAction in the UBFActionList object; if at the end of the list or the list is empty a null object pointer is returned; this is tracked behind the scenes to a user; automatically sets to the first object each time the UBF tree restarts; may be set back to the beginning by the Restart_Next_Iterator() method
Parameters: None
Returned Object: UBFAction
Example Usages:
UBFActionList object:

```
UBFAction actionA = <listName>.Get_Next_Action();
```

UBFBehavior storage:

```
UBFAction actionA = UBFBehavior.Get_Next_Action();
```

UBFArbiter input:

```
UBFAction actionA = UBFArbiter.Get_Next_Action();
```

Command Name: **.Restart_Next_Iterator()**
Scope: Method of a UBFActionList object used via dot operator
Description: This command sets the iterator used by the UBFActionList object in question back to the start of its list; this is used with the Get_Next_Action() command
Parameters: None
Returned Object: None
Example Usages:
UBFActionList object:

```
<listName>.Restart_Next_Iterator();
```

UBFBehavior storage:

```
UBFBehavior.Restart_Next_Iterator();
```

UBFArbiter input:

```
UBFArbiter.Restart_Next_Iterator();
```

Command Name: **.Get_Number_Of_Actions()**
Scope: Method of a UBFActionList object used via dot operator
Description: This command returns the number of UBFActions in the UBFActionList object in question

Parameters: None
Returned Object: integer representing number of UBFActions in the UBFActionList object in question
Example Usages:
UBFActionList object:

```
int size  = <listName>.Get_Number_Of_Actions();
```

UBFBehavior storage:

```
int size = UBFBehavior.Get_Number_Of_Actions();
```

UBFArbiter input:

```
int size = UBFArbiter.Get_Number_Of_Actions();
```

Command Name: **.Get_Actions_By_Exact_Name(string)**
Scope: Method of a UBFActionList object used via dot operator
Description: This command returns a UBFActionList object of UBFActions whose name(s) exactly match the provided string
Parameters: string representing the name to be matched
Returned Object: UBFActionList
Example Usages:
UBFActionList object:

```
UBFActionList <listNameA>  =
                <listNameB>.Get_Actions_By_Exact_Name("fly");
```

UBFBehavior storage:

```
UBFActionList <listNameA>  =
                UBFBehavior.Get_Actions_By_Exact_Name("fly");
```

UBFArbiter input:

```
UBFActionList <listNameA>  =
                UBFArbiter.Get_Actions_By_Exact_Name("fly");
```

Command Name: **.Get_Actions_By_Partial_Name(string)**
Scope: Method of a UBFActionList object used via dot operator
Description: This command returns a UBFActionList object of UBFActions which whose name(s) contain the provided string
Parameters: string representing the name to be matched
Returned Object: UBFActionList
Example Usages:
UBFActionList object:

```
UBFActionList <listNameA> =
                <listNameB>.Get_Actions_By_partial_Name("fly");
```

UBFBehavior storage:

```
UBFActionList <listNameA> =
                 UBFBehavior.Get_Actions_By_partial_Name("fly");
```

UBFArbiter input:

```
UBFActionList <listNameA> =
                 UBFArbiter.Get_Actions_By_partial_Name("fly");
```

Command Name: **.Get_Actions_By_Exact_Priority(int)**
Scope: Method of a UBFActionList object used via dot operator
Description: This command returns a UBFActionList object of UBFActions whose priority is the same as the provided integer
Parameters: integer representing the priority to be compared against
Returned Object: UBFActionList
Example Usages:
UBFActionList object:

```
UBFActionList <listNameA> =
                <listNameB>.Get_Actions_By_Exact_Priority(3);
```

UBFBehavior storage:

```
UBFActionList <listNameA> =
                UBFBehavior.Get_Actions_By_Exact_Priority(3);
```

UBFArbiter input:

```
UBFActionList <listNameA> =
                UBFArbiter.Get_Actions_By_Exact_Priority(3);
```

Command Name: **.Get_Actions_By_Min_Priority(int)**
Scope: Method of a UBFActionList object used via dot operator
Description: This command returns a UBFActionList object of UBFActions whose priority is at least the provided integer
Parameters: integer representing the priority compared against
Returned Object: UBFActionList
Example Usages:
UBFActionList object:

```
UBFActionList <listNameA> =
                <listNameB>.Get_Actions_By_Min_Priority(2);
```

UBFBehavior storage:

```
UBFActionList <listNameA>  =
              UBFBehavior.Get_Actions_By_Min_Priority(2);
```

UBFArbiter input:

```
UBFActionList <listNameA>  =
               UBFArbiter.Get_Actions_By_Min_Priority(2);
```

Command Name: **.Get_Actions_by_type_Double()**
Scope: Method of a UBFActionList object used via dot operator
Description: This command returns a UBFActionList object of UBFActions who have had the double value set
Parameters: None
Returned Object: UBFActionList
Example Usages:
UBFActionList object:

```
UBFActionList <listNameA>  =
               <listNameB>.Get_Actions_by_type_Double();
```

UBFBehavior storage:

```
UBFActionList <listNameA>  =
               UBFBehavior.Get_Actions_by_type_Double();
```

UBFArbiter input:

```
UBFActionList <listNameA>  =
               UBFArbiter.Get_Actions_by_type_Double();
```

Command Name: **.Get_Actions_by_type_Int()**
Scope: Method of a UBFActionList object used via dot operator
Description: This command returns a UBFActionList object of UBFActions who have had the integer value set
Parameters: None
Returned Object: UBFActionList
Example Usages:
UBFActionList object:

```
UBFActionList <listNameA>  =
               <listNameB>.Get_Actions_by_type_Int();
```

UBFBehavior storage:

```
UBFActionList <listNameA>  =
                UBFBehavior.Get_Actions_by_type_Int();
```

UBFArbiter input:

```
UBFActionList <listNameA>  =
                UBFArbiter.Get_Actions_by_type_Int();
```

Command Name: **.Get_Actions_by_type_String()**
Scope: Method of a UBFActionList object used via dot operator
Description: This command returns a UBFActionList object of UBFActions who have had the string value set
Parameters: None
Returned Object: UBFActionList
Example Usages:
UBFActionList object:

```
UBFActionList <listNameA>  = <listNameB>.Get_Actions_by_type_String();
```

UBFBehavior storage:

```
UBFActionList <listNameA>  = UBFBehavior.Get_Actions_by_type_String();
```

UBFArbiter input:

```
UBFActionList <listNameA>  = UBFArbiter.Get_Actions_by_type_String();
```

Command Name: **.Get_Actions_Unique_Top_Priorities()**
Scope: Method of a UBFActionList object used via dot operator
Description: This command returns a UBFActionList object of UBFActions who have unique names amongst themselves and are the highest priority of identically named UBFActions from the original UBFActionList object
Parameters: None
Returned Object: UBFActionList
Example Usages:
UBFActionList object:

```
UBFActionList <listNameA>  =
                <listNameB>.Get_Actions_Unique_Top_Priorities();
```

UBFBehavior storage:

```
UBFActionList <listNameA>  =
                UBFBehavior.Get_Actions_Unique_Top_Priorities();
```

UBFArbiter input:

```
UBFActionList <listNameA> =
                UBFArbiter.Get_Actions_Unique_Top_Priorities();
```

Command Name: **.Add_Action(UBFAction)**
Scope: Method of a UBFActionList object used via dot operator
Description: This command adds a UBFAction object to the UBFActionList in question; if invoked via UBFBehavior.Add_Action(...) it adds the UBFAction to the currently operating UBFBehavior's set of UBFActions; if invoked within a UBFArbiter code block via UBFArbiter.Add_Action(...) it adds the UBFAction to the currently operating UBFArbiter's set of UBFActions that will be output (can not explicitly remove or access this output list besides this method)
Parameters: None
Returned Object: N/A
Example Usages:
UBFActionList object:

```
<listNameB>.Add_Action(ActionA);
```

UBFBehavior storage:

```
UBFBehavior.Add_Action(ActionA);
```

UBFArbiter output:

```
UBFArbiter.Add_Action(ActionA);
```

Commands - UBFAction.

Commands are added to code blocks that expose the UBFAction object type, its associated functions, and associated fields. This allows the use of UBFAction objects. These commands are:

Object type: **UBFAction**
Scope: Usable within UBFArbiter code blocks and UBFBehavior code blocks
Description: This object may be instantiated on its own; it is used to group together a name, priority, an integer value, a double value, and a string value into a single place
Example Usage:
Declare UBFActionList variable

```
UBFAction <actionName>;
```

Command Name: **.Create()**
Scope: Method of a UBFAction object used via dot operator
Description: This command instantiates a UBFAction object
Parameters: None
Returned Object: UBFAction
Example Usage:

```
UBFAction <actionNameB> = UBFAction.Create();
```

Command Name: **.Create(string, double, double, string)**
Scope: Method of a UBFAction object used via dot operator
Description: This command instantiates a UBFAction object with values set to the provided parameters
Parameters: string representing the name, double representing the priority, double representing the vote, string representing a value
Returned Object: UBFAction
Example Usage:

```
UBFAction <actionNameB> = UBFAction.Create("Fly", 4, 10, "Dayton");
```

Command Name: **.Create(string, double, double, int)**
Scope: Method of a UBFAction object used via dot operator
Description: This command instantiates a UBFAction object with values set to the provided parameters; currently not invokable since AFSIM automatically calls the double version of the Create method and casts the value to a double
Parameters: string representing the name, double representing the priority, double representing the vote, int representing a value
Returned Object: UBFAction
Example Usage:

```
UBFAction <actionNameB> = UBFAction.Create("Fly", 4, 10, 77);
```

Command Name: **.Create(string, double, double, double)**
Scope: Method of a UBFAction object used via dot operator
Description: This command instantiates a UBFAction object with values set to the provided parameters
Parameters: string representing the name, double representing the priority, double representing the vote, double representing a value
Returned Object: UBFAction
Example Usage:

```
UBFAction  <actionNameB> = UBFAction.Create("Fly", 4, 10, 22.55);
```

Command Name: **.Create(string, double, double, WsfGeoPoint)**
Scope: Method of a UBFAction object used via dot operator
Description: This command instantiates a UBFAction object with values set to the provided parameters
Parameters: string representing the name, double representing the priority, double representing the vote, WsfGeoPoint representing a value
Returned Object: UBFAction
Example Usage:

```
UBFAction  <actionNameB> = UBFAction.Create("Fly", 4, 10, GeoPointObject);
```

Command Name: **.Create(string, double, double, WsfTrack)**
Scope: Method of a UBFAction object used via dot operator
Description: This command instantiates a UBFAction object with values set to the provided parameters
Parameters: string representing the name, double representing the priority, double representing the vote, WsfTrack object representing a value
Returned Object: UBFAction
Example Usage:

```
UBFAction  <actionNameB> = UBFAction.Create("Fly", 4, 12, TrackObject);
```

Command Name: **.Create(string, double, double, WsfRoute)**
Scope: Method of a UBFAction object used via dot operator - Currently BROKEN
Description: This command is supposed to instantiate a UBFAction object with values set to the provided parameters, however the WsfRoute object is currently broken
Parameters: string representing the name, double representing the priority, double representing the vote, WsfRoute object representing a value
Returned Object: UBFAction
Example Usage:

```
UBFAction  <actionNameB> = UBFAction.Create("Fly", 4, 10, RouteObject);
```

Command Name: **.Create(UBFAction)**
Scope: Method of a UBFAction object used via dot operator
Description: This command instantiates a UBFAction object by copying another UBFAction object
Parameters: UBFAction object to be copied
Returned Object: UBFAction
Example Usage:

```
UBFAction <actionNameB> = UBFAction.Create(ActionBravo);
```

Command Name: **.Get_Name()**
Scope: Method of a UBFAction object used via dot operator
Description: This command returns the name of the UBFAction object
Parameters: None
Returned Object: string representing name of UBFBehavior
Example Usage:

```
string tempString = <actionNameB>.Get_Name();
```

Command Name: **.Get_Priority()**
Scope: Method of a UBFAction object used via dot operator
Description: This command returns the priority of the UBFAction object, higher priority is closest to 0
Parameters: None
Returned Object: Integer representing priority of the UBFAction object
Example Usage:

```
int tempPriority = <actionNameB>.Get_Priority();
```

Command Name: **.Get_Vote()**
Scope: Method of a UBFAction object used via dot operator
Description: This command returns the vote of the UBFAction object, higher vote is the larger number
Parameters: None
Returned Object: Integer representing vote of the UBFAction object
Example Usage:

```
int tempVote = <actionNameB>.Get_Vote();
```

Command Name: **.Get_Int()**
Scope: Method of a UBFAction object used via dot operator
Description: This command returns the integer value field of the UBFAction object
Parameters: None
Returned Object: integer of the integer value field of the UBFAction object
Example Usage:

```
int tempString = UBFAction.Get_Int();
```

Command Name: **.Get_Double()**
Scope: Method of a UBFAction object used via dot operator

Description: This command returns the double value field of the UBFAction object
Parameters: None
Returned Object: Double of the double value field of the UBFAction object
Example Usage:

```
double tempString = <actionNameB>.Get_Double();
```

Command Name: **.Get_String()**
Scope: Method of a UBFAction object used via dot operator
Description: This command returns the string value field of the UBFAction object
Parameters: None
Returned Object: string of the string value field of the UBFAction object
Example Usage:

```
string tempString = <actionNameB>.Get_String();
```

Command Name: **.Get_Track()**
Scope: Method of a UBFAction object used via dot operator
Description: This command returns the WsfTrack value field of the UBFAction object
Parameters: None
Returned Object: WsfTrack of the WsfTrack value field of the UBFAction object
Example Usage:

```
WsfTrack tempTrack = <actionNameB>.Get_Track();
```

Command Name: **.Get_Geo_Point()**
Scope: Method of a UBFAction object used via dot operator
Description: This command returns the double value field of the UBFAction object
Parameters: None
Returned Object: WsfGeoPoint of the WsfGeoPoint value field of the UBFAction object
Example Usage:

```
WsfGeoPoint tempPoint = <actionNameB>.Get_Geo_Point();
```

Command Name: **.Get_Route()**
Scope: Method of a UBFAction object used via dot operator
Description: This command returns the double value field of the UBFAction object
Parameters: None
Returned Object: WsfRoute of the WsfRoute value field of the UBFAction object
Example Usage:

```
double tempString = <actionNameB>.Get_Double();
```

Command Name: **.Set_Name(string)**
Scope: Method of a UBFAction object used via dot operator
Description: This command sets the name of the UBFAction object
Parameters: String of the Name for this UBFAction
Returned Object: None
Example Usage:

```
<actionNameB>.Set_Name("FlyToDallas");
```

Command Name: **.Set_Priority(double)**
Scope: Method of a UBFAction object used via dot operator
Description: This command sets the priority of the UBFAction object, higher priority is closest to 0
Parameters: Double of the priority for this UBFAction
Returned Object: None
Example Usage:

```
<actionNameB>.Set_Priority(10);
```

Command Name: **.Set_Vote(double)**
Scope: Method of a UBFAction object used via dot operator
Description: This command sets the vote of the UBFAction object, highest vote is the larger number
Parameters: Double of the vote for this UBFAction
Returned Object: None
Example Usage:

```
<actionNameB>.Set_Vote(10);
```

Command Name: **.Set_Int(int)**
Scope: Method of a UBFAction object used via dot operator
Description: This command sets the integer value field of the UBFAction object
Parameters: Integer of the integer value field for this UBFAction
Returned Object: None
Example Usage:

```
<actionNameB>.Set_Int(10);
```

Command Name: **.Set_Double(double)**
Scope: Method of a UBFAction object used via dot operator
Description: This command sets the double value field of the UBFAction object

Parameters: Double of the double value field for this UBFAction
Returned Object: None
Example Usage:

```
<actionNameB>.Set_Double(10.2);
```

Command Name: **.Set_String(string)**
Scope: Method of a UBFAction object used via dot operator
Description: This command sets the string value field of the UBFAction object
Parameters: String of the string value field for this UBFAction
Returned Object: None
Example Usage:

```
<actionNameB>.Set_String("FLY");
```

Command Name: **.Set_Track(WsfTrack)**
Scope: Method of a UBFAction object used via dot operator
Description: This command sets the WsfTTrack value field of the UBFAction object
Parameters: WsfTrack of the WsfTrack value field for this UBFAction
Returned Object: None
Example Usage:

```
<actionNameB>.Set_Track(TrackObject);
```

Command Name: **.Set_Geo_Point(WsfGeoPoint)**
Scope: Method of a UBFAction object used via dot operator
Description: This command sets the WsfGeoPoint value field of the UBFAction object
Parameters: WsfGeoPoint of the WsfGeoPoint value field for this UBFAction
Returned Object: None
Example Usage:

```
<actionNameB>.Set_Geo_Point(GeoPointObject);
```

Command Name: **.Set_Route(WsfRoute)**
Scope: Method of a UBFAction object used via dot operator
Description: This command sets the WsfRoute value field of the UBFAction object
Parameters: WsfRoute of the WsfRoute value field for this UBFAction
Returned Object: None
Example Usage:

```
<actionNameB>.Set_Route(RouteObject);
```

3.4 Summary

This chapter examines concepts and commands necessary to extend AFSIM with UBF. An understanding of the underlying structure of UBF in AFSIM is gained through a presentation of the C++ class structure and flow charts depicting the sequence in which code blocks execute. A mapping of other frameworks to this plug-in is provided in order to show the concepts that were and were not implemented for this thesis. The concepts of other frameworks/languages that are implemented or omitted are shown by an examination of those concepts. The requirement for new commands is established through this examination via differentiating which concepts may re-utilize AFSIM components and which require new commands. Familiarity of the UBF in AFSIM syntax is gained through the documentation of all the new commands in Section 3.3.1. Through a detailed examination of the code structure, a mapping from other framework's concepts, and documentation of the syntax, the commands and concepts necessary for the Unified Behavior Framework in AFSIM is established.

IV. Experimental Implementation and Evaluation

Chapter III described the methodology to augment the Advanced Framework for Simulation, Integration, and Modeling (AFSIM) scripting language, however, that is not enough to demonstrate behavioral emergence or that this is an acceptable tool to use. This chapter demonstrates behavioral emergence through four scenarios in the AFSIM which use the new Unified Behavior Framework (UBF) implementation. The first scenario is a proof of concept demonstrating the ability to replace a Behavior Tree (BT) from training materials of AFSIM and establishes an interface for the Map_To_Action code block. The second scenario evaluates the potential benefit of behavioral emergence over discrete behavior selection and display the effects of tuning the emergence. The third scenario demonstrates behavioral emergence by implementing the Boids [11] strategy for swarming; this strategy relies on emergence for swarming to occur. The forth scenario examines the effort required to merge scenarios two and three in order to show the increase in code reuse and maintainability of UBF over BTs.

This chapter reuses the prior symbols to indicate the role of different elements in a UBFBehavior, also seen in Figure 9. The lines for the thread of execution and the flow of action objects are omitted because those processes are implied in this chapter. The individual components of UBF behaviors may also be omitted. If a figure shows all components of a behavior omitted then no assumptions are made about the contents of the behavior, typically used to show a tree structure or children of a behavior, thus a child is not considered a component of a behavior in this context. If any component of a behavior is included then it is assumed the figure shows all of the components for that behavior; this is used to prevent figures of UBF behaviors from misleading their readers by possibly omitting behavior changing components, i.e. an extreme example is omitting a pre_condition code block that always fails, effectively nullifying the rest of the components. A figure showing a UBF tree with children whom also have children does not indicate if the child's tree defined its own children or if the children were defined by the parent, or both. Figures for BTs do not use these special symbols because they only require an indication of the node type, behavior name, and representation of their structure.

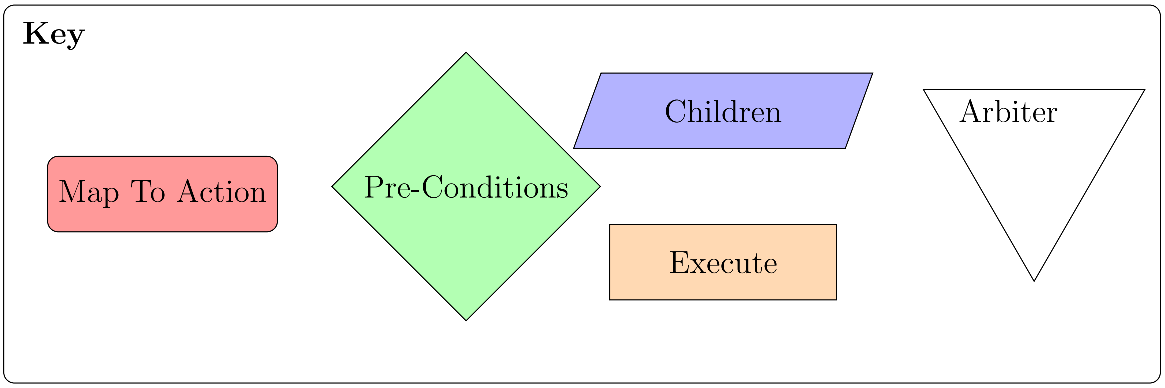


Figure 9. UBFBehavior Key.

4.1 Behavior Tree Adapted scenario

The AFSIM's analyst training includes a basic tutorial on its implementation of behavior trees; this was used as a proof of concept to show the UBF implementation can replace an existing BT. This tutorial consists of a scenario with a blue force defending against a red force. The red force consists of a command ship, 4 stand off jammer (SOJ) aircraft, and 4 unmanned combat air vehicles (UCAV) on an attack run. All red units simply follow a pre-defined route and shoot munitions when in range of pre-assigned targets. The blue force consists of 4 targets, various radars to detect the enemy, various surface to air missile (SAM) sites to shoot enemies that get close, a command post to assign units tasks, and 4 behavior tree controlled striker aircraft acting as active defenders. The behavior tree on the 4 strikers is replaced by a UBF tree and shown to have a similar effectiveness on the outcome. An image with the initial scenario is shown in Figure 10.

Figure 10. Initial Scenario BT to UBF

The tutorial BT is shown in Figure 11 and the resulting UBF tree is shown in Figure 12. Scripts for these are shown in Appendix B.1. These diagrams show the translation of a BT to a UBF tree.

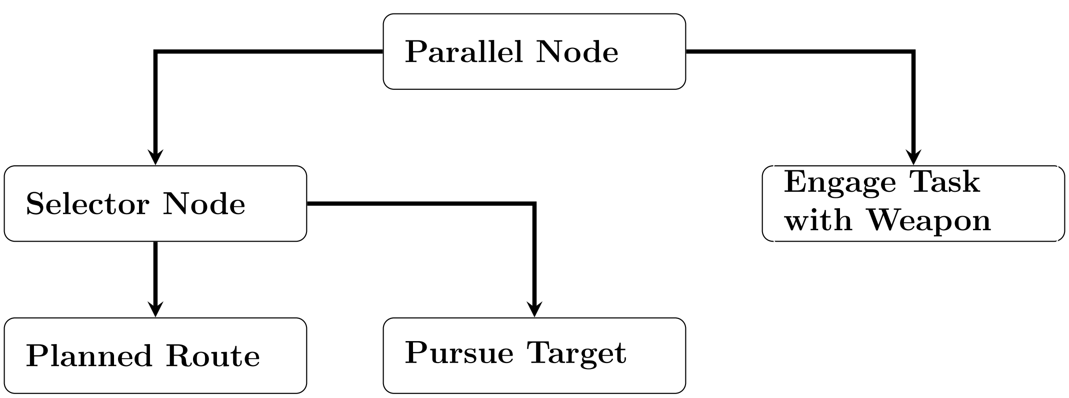


Figure 11. Tutorial Behavior Tree.

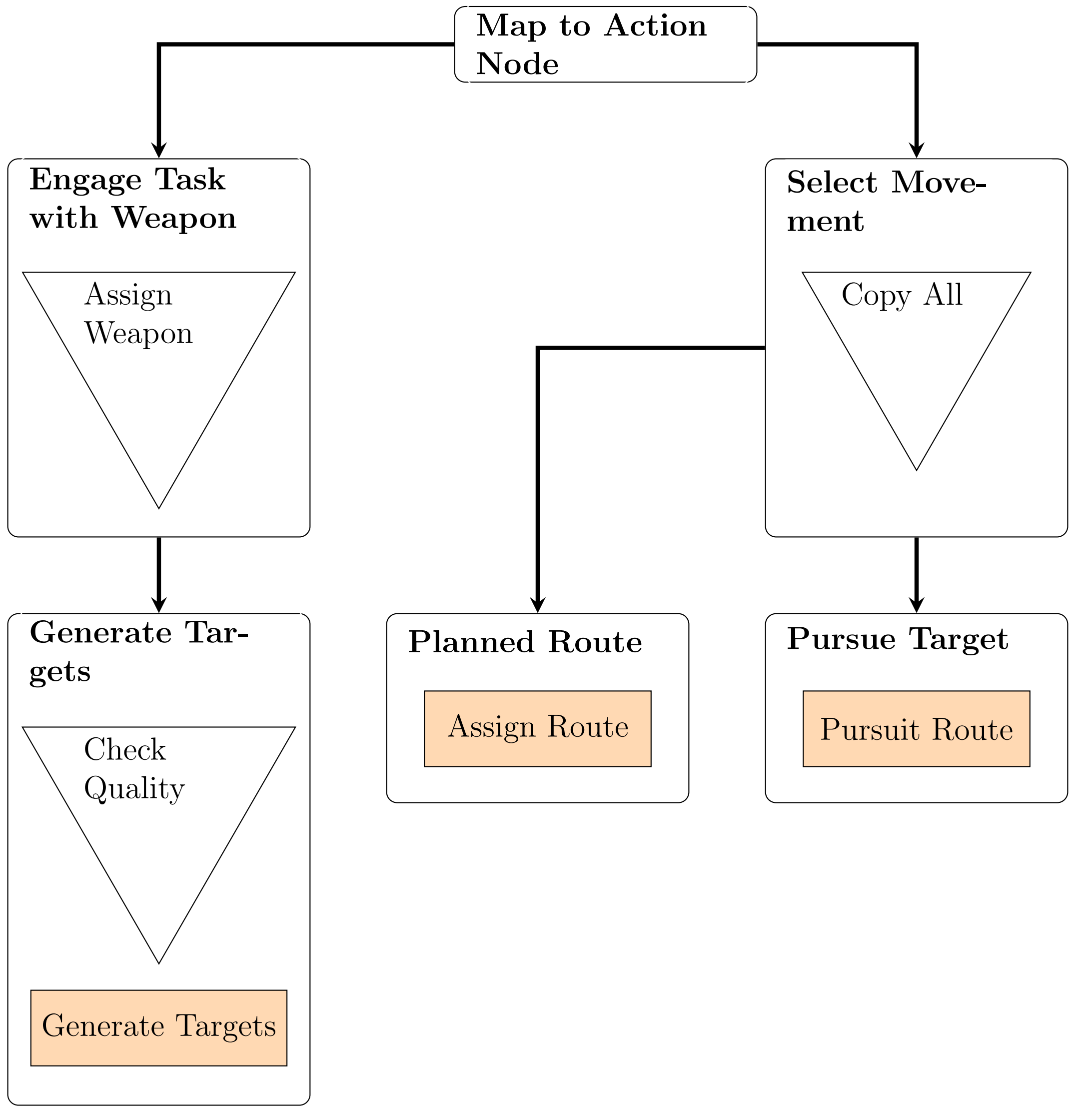


Figure 12. Tutorial UBF Tree.

4.1.1 Translating the Individual Behaviors.

In order to translate each behavior from a BT to a UBF tree the definitions for individual behaviors are provided. The normal, non-bold, text is a generic definition for a behavior whereas the **bold** text is additional detail used to specifically define the interface of a behavior in the UBF tree, the UBF Action object's format. Each behavior's definition is:

Name: Planned Route
Description: Sets the agent to the original pre-planned route if and only if it was extrapolating its path
Dependencies: The agent needs a mover object associated with it and a route object associated with it
Output: Original route

Name	**RouteLat/RouteLong**
Priority	**waypoint's index in the route**
double	**Lat/Long**
int	**altitude only for routeLong**

Name	**RouteStart**
Priority	**starting point for route index**
double	**route.size()**

Name: Pursue Target
Description: Sets the agent route to the target associated with the first task assigned
Dependencies: The agent needs a mover object and a task assigned to it
Output: Route to a Target Platform

Name	**RouteLat/RouteLong**
Priority	**waypoint's index in the route**
double	**Lat/Long**
int	**altitude only for routeLong**

Name: Engage Task with Weapon
Description: Generates an attack against the tasked target IFF in range and viable
Dependencies: Input

Name	**Target**
Priority	**n/a**
int	**WsfTrackId.Number()**
string	**WsfTrackId.Name()**

Output: Attacking of a tasked target with valid a weapon

Name	**Weapon**
Priority	**n/a**
int	**WsfTrackId.Number()**
string	**WsfTrackId.Name()**
Double	**weapon index**

Name: Generate Targets
Description: This behavior generates targets from a task list
Dependencies: n/a
Output: Targets for the agent to attack

Name	**Target**
Priority	**2**
int	**WsfTrackId.Number()**
string	**WsfTrackId.Name()**

Name: Select Movement
Description: This behavior is used to combine children
Dependencies: n/a
Output: Each item that was input

Name: Map to Action Node
Description: This behavior maps action recommendations to outputs
Dependencies: Input:

Name	**Weapon**
Priority	**n/a**
int	**WsfTrackId.Number()**
string	**WsfTrackId.Name()**
Double	**weapon index**
Name	**RouteLat/RouteLong**
Priority	**waypoint's index in the route**
double	**Lat/Long**
int	**altitude only for routeLong**

Output: effects on the agent

4.1.2 Discussion of BT translation to UBF tree.

This implementation presents multiple items regarding conversion of a BT to a UBF tree. First, more effort is required by UBF tree creators if they do not have pre-defined arbiters or an established standard Map_To_Action code block that implements action recommendations. The Map_To_Action code block starts to establish this standard with its ability to fly the agent to points and attack the target if given the correct actions. These arbiters demonstrate that an analyst is able to create completely custom arbiters for their code, however these arbiters are only applicable to a very specific set of inputs. Creating a generic set of arbiters based on the priority and vote values could accomplish that idea.

Second, this implementation shows that UBF behaviors can be used to modify and combine behaviors; it is worth noting that modifying other behavior's output should be done in the Execute code block of a parent behavior, not the Arbiter. If done in the Arbiter, the Arbiter's purpose is bound to the behavior and vice versa instead of simply having a behavior to accomplish that purpose, two items where one is necessary.

The third item this implementation presents is the ability for UBF trees to replace BTs. In Figures 13 and 14 the result of the UBF controlled and BT controlled scenarios are shown. These figures show that a squad of intelligent agent controlled blue aircraft were able to fly out, destroy their tasked targets, and return to their home routes (currently returning in the figures). As much of the script as possible is reused in creating the UBF behaviors to mirror their BT counter parts. Due to the fact that the UBF tree agents accomplish the same abstract goals of destroying their tasked targets, flying at their tasked targets, and returning home, with similar scripts this acts as a proof of concept that UBF can replace a BT inside of AFSIM.

Figure 13. BT to UBF Scenario - UBF Agents.

Figure 14. BT to UBF Scenario - BT Agents.

4.2 Established interfaces

In order to reduce the overhead of creating custom 'Map_To_Action' and 'Arbiter' code blocks for every agent standards are established. The first standard is: arbiter objects must be built generically, i.e. an arbiter should not need to know the integer field is a weapon index. Based on this, the set of arbiters created and usable are:

- Fusion_Dbl
- Fusion_Int
- Fusion_GeoPoint
- Fusion_Dbl_Int_GeoPoint
- Fusion_Vote_Dbl
- Fusion_Vote_Int
- Fusion_Vote_GeoPoint
- Fusion_Vote_Dbl_Int_GeoPoint

- WTA_Priority
- WTA_Vote
- Fusion_ByName_Dbl
- Fusion_ByName_Int
- Fusion_ByName_GeoPoint
- Fusion_ByName_Dbl_Int_GeoPoint
- Fusion_ByName_Vote_Dbl
- Fusion_ByName_Vote_Int
- Fusion_ByName_Vote_GeoPoint
- Fusion_ByName_Vote_Dbl_Int_GeoPoint
- WTA_ByName_Priority
- WTA_ByName_Vote

The arbiters all act respective of their names. The "Fusion" arbiters all average together the contents of the fields they denote, copy forward the name, priority, averaged vote fields, and drop the fields not mentioned. For example Fusion_Dbl averages the double fields of all provided action objects and does not pass forward any of the other value fields. Similarly a Fusion_Vote arbiter averages the respective fields by weighting the actions with their vote values.

Some of the arbiters act based on the priority or vote of the actions. These two concepts differ slightly. Votes are higher if the number is larger and greater than zero while priorities are higher if the number is closer to 0 and non-negative. Hence, the 0th priority is the best and a vote of 100 has more impact than a vote of 1. Priorities are also conceptually a method used to group actions together while votes are meant to show the degree to which an action should affect the agent.

The "WTA" arbiters are "winner take all" decisions which are made based on the priority or vote fields as their names suggest. The arbiters with a "ByName" substring return one action object for every unique name among the set of actions provided, with the corresponding operation being executed on each uniquely named set. The winner take all arbiters behave slightly differently when operating on a by name basis. The winner take all arbiters who do not consider the name may pass forward a set of actions. This set of actions has the same vote or priority value that is the highest among all action objects. The winner take all arbiters who do consider the name may also pass forward a set of actions. However, this resultant set of actions has a single entry for each unique name in the original set and that entry has the highest priority or vote among the subset of other behaviors with the same name.

The Map_To_Action code block standard is inspired from the implementation in Section 4.1. The scenarios in Sections 4.3 and 4.4 do not use weapons so that portion of the Map_To_Action code block is not used. Also, since a latitude and longitude are all that is needed the input of the code block uses a WsfGeoPoint instead. The name required is “Route” for each action object input to the root. The index of the point in the route’s list of points is held in the integer value field of the action object. The actual script used for this is shown in Figure 15.

Essentially the Map_To_Action code block sets the autopilot to a new point(s) every ten seconds. Vector inputs such as the “GotoLocation” method is not used in the standard because these commands cause silent run time errors that have not been solved; the simulation stops with no error message. This led to a standard, which uses commands that work and behaviors map their outputs to that standard.

The use of these standards decreases time to implement agents. These standards are not universal and are designed specifically for the scenarios in Sections 4.3 and 4.4. These are used as examples showing standards should be used when teams develop behaviors for a UBF tree.

```
platform_type  Printer_Friendly_Map_To_Action  WSF_PLATFORM
   processor rootNode UBFBehavior
      update_interval 10 sec
      Map_To_Action
         if(UBFBehavior.Get_Number_Of_Actions()==0)
         {
            return;
         }
         UBFActionList RouteList =
               UBFBehavior.Get_Actions_By_partial_Name("Route");

         if(RouteList.Get_Number_Of_Actions()>0)
         {
            #construct array of points
            Array<WsfGeoPoint> points;
            points = Array<WsfGeoPoint>();
            for(int ii=0;ii<RouteList.Get_Number_Of_Actions();ii=ii+1)
            {
                 UBFAction tempAction = RouteList.Get_Action_By_Index(ii);
                 points.Set(tempAction.Get_Int(),tempAction.Get_Geo_Point());
            }
            #current position as start
            points.Set(0,PLATFORM.Location());
            WsfRoute newRoute =WsfRoute();
            for(int ii=0;ii<points.Size();ii=ii+1)
             {
                 newRoute.Append(points.Get(ii),450.0);
             }

             if((newRoute.Size()>0)&&(newRoute.IsValid()))
             {
                 PLATFORM.FollowRoute(newRoute);
             }
          }
      end_Map_To_Action
   end_processor
end_platform_type
```

Figure 15. Map_To_Action standard.

4.3 Behavior Emergence tuning scenario

The emergence of behaviors versus the use of a behavior tree (BT) is not explored in Section 4.1. In order to explore an advantage of behavior emergence over discrete behavior selection a new scenario is used. This scenario involves a behavior tree agent in blue, the discrete behavior selection, and a Unified Behavior Language agent in red, the emergent behavior, flying to a goal point while avoiding an obstacle in the way. The emergent behavior is identified by a number overlaid on its aircraft to assist further differentiating the aircraft, this is the vote of the avoid obstacle behavior for that agent. Time to reach the goal is used as a measurement to compare the two methods. Less time results in less fuel used with vehicles at a fixed speed, a shorter distance being covered, and a smoother path being used.

4.3.1 Behavior Structures Implemented.

Similar behaviors are used between the two implementations. Each structure has a behavior to fly to the goal point and another to avoid a point. The UBF tree also has a behavior called "EmergenceNormalize" which is used to ensure the WsfGeoPoint used is not too close to the aircraft, sets it about 7 miles away in the correct direction to prevent the agent from thinking it successfully arrived at a point because the point was too close to it when placed. The UBF tree is displayed in Figure 17 and the BT is displayed in Figure 16. The actual scripts for each are included in Appendix 2.2

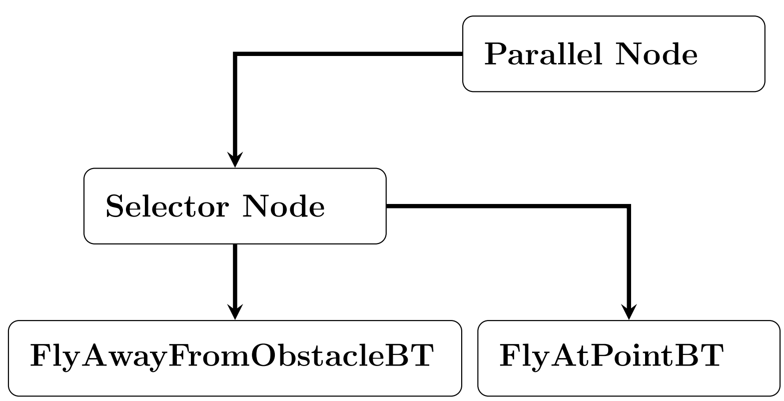


Figure 16. Behavior Tree of Fly To Goal Agent.

Similar formatting to section 4.1 is used for these behavior's definitions, which are as follows:

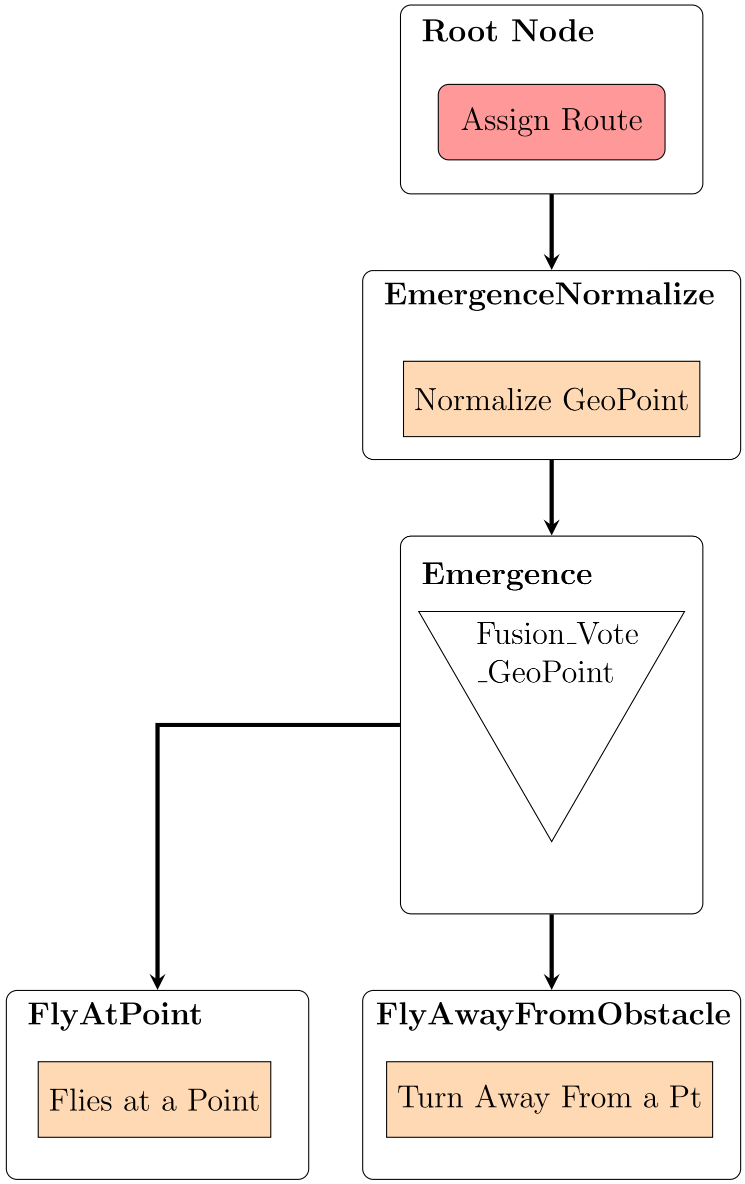


Figure 17. UBF Tree of Fly to Goal Agent.

Name: FlyAwayFromObstacleBT
Description: Behavior Tree behavior which sets the destination of an agent away from an obstacle point; the first child of the Selector Node forcing itself to be selected when within 25km of the obstacle
Dependencies: The agent needs a mover object associated with it
Output: Route away from obstacle

Name: FlyAtPointBT
Description: Behavior Tree behavior which sets the destination of an agent to a goal point if selected; always selected if nothing else is first
Dependencies: The agent needs a mover object associated with it
Output: Route to a goal point

Name: Root Node
Description: Maps an action object named 'Route' to cause an agent to follow the provided WsfGeoPoint
Dependencies: Agent needs a mover object and this behavior needs an input with 'Route' named object and WsfGeoPoint:

Name	**Route**
Priority	**n/a**
Vote	**n/a**
Int	**index of point to fly to**
WsfGeoPoint	**A point to fly to**

Output: Route to a point

Name: Emergence Normalize
Description: Takes a WsfGeoPoint and moves it to approximately 7 miles from the agent along the same heading to ensure the point is not immediately assumed visited
Dependencies: Input with 'Route' named object and WsfGeoPoint:

Name	**Route**
Priority	**n/a**
Vote	**n/a**
Int	**n/a**
WsfGeoPoint	**A point to fly to**

Output: Route to a point

Name	**Route**
Priority	**n/a**
Int	**1 - Indicates 1st point to fly to**
WsfGeoPoint	**A point to fly to**

Name: Emergence
Description: Averages two WsfGeoPoints based on vote value
Dependencies: Agent needs a mover object and this behavior needs an input with 'Route' named action object and WsfGeoPoint of the format:

Name	**Route**
Priority	**n/a**
Vote	**n/a**
Int	**1 - Indicates 1st point to fly to**
WsfGeoPoint	**A point to fly to**

Output: Action object with a point to go towards of the format:

Name	**Route**
Priority	**n/a**
Vote	**affects weight of average attained; suggest range 0-10**
Int	**1 - Indicates 1st point to fly to**
WsfGeoPoint	**A point to fly to**

Name: FlyAtPoint
Description: Provides a WsfGeoPoint in the direction of the goal point
Dependencies: n/a
Output: Action object with a point to go towards of the format:

Name	**Route**
Priority	**n/a**
Vote	**1**
Int	**1 - Indicates 1st point to fly to**
WsfGeoPoint	**A point to fly to**

Name: FlyAwayFromObstacle
Description: Provides a WsfGeoPoint in the direction away from an obstacle point
Dependencies: n/a
Output: Action object with a point to go towards of the format:

Name	**Route**
Priority	**n/a**
Vote	**Dependent on agent name**
Int	**1 - Indicates 1st point to fly to**
WsfGeoPoint	**A point to fly to**

4.3.2 Comparison: UBF agent versus BT agent.

In order to compare the behavior tree (BT) and Unified Behavior Framework (UBF) agents the time between each agent was recorded by the tree reporting its time when it reached the goal. It is worth noting that this results in a time resolution of 10 seconds because each agent's respective tree runs once per 10 seconds. Multiple UBF agents are used to test and show the effects of different votes. The vote is applied to an agent's FlyAwayFromObstacle behavior by stripping the vote value off of the name of the agent. Figure 18 shows the simulation running the described scenario with 10 UBF aircraft and votes ranging from 0-10 and one BT aircraft. As a reminder the Red aircraft are the UBF tree controlled aircraft and the number overlaid on the aircraft is the vote value used, Blue is used for the behavior tree aircraft. The times obtained are shown in Table 16.

Figure 18. Voting with 10 Aircraft UBF vs BT Scenario.

This table identifies multiple factors about behavioral emergence. The first is

Table 16. UBF vs BT Times to Reach Goal.

Agent	Vote	Time (s)
BT	n/a	840
UBF	0	750
UBF	1	760
UBF	2	780
UBF	3	830
UBF	4	910
UBF	5	1040
UBF	6	1360
UBF	7	n/a
UBF	8	n/a
UBF	9	n/a

that with the wrong vote value a goal may never be achieved, either flying over the obstacle or never reaching the goal. The second is that behavioral emergence, with tuning, can achieve goals better, faster, than discrete behavior selection. Finally, it reveals that static voting may not be the best solution, in that some violate the objective area and some never achieve the goal.

4.4 Emergent Behavior based Implementation

The scenario simulates Boids, referring to bird like objects, software because Boids is a classical example of emergent behavior [11]. The three key behaviors of Boids are separation, alignment, and cohesion; when those behaviors combine, a swarming behavior emerges. This is implemented nearly identically to the UBF tree from Section 4.3, with only the leaf movement behaviors being replaced and the others being renamed.

To implement this, a single UBF behavior is created for each and combined using a Fusion_Vote_GeoPoint arbiter. Then the resulting point is normalized, moved out from under the agent to prevent AFSIM immediately assuming the point was successfully reached. Finally, it is assigned as the active Route to follow in the

Map_To_Action code block. The resulting tree is shown in Figure 19. The actual code used for each is included in Appendix 2.3.

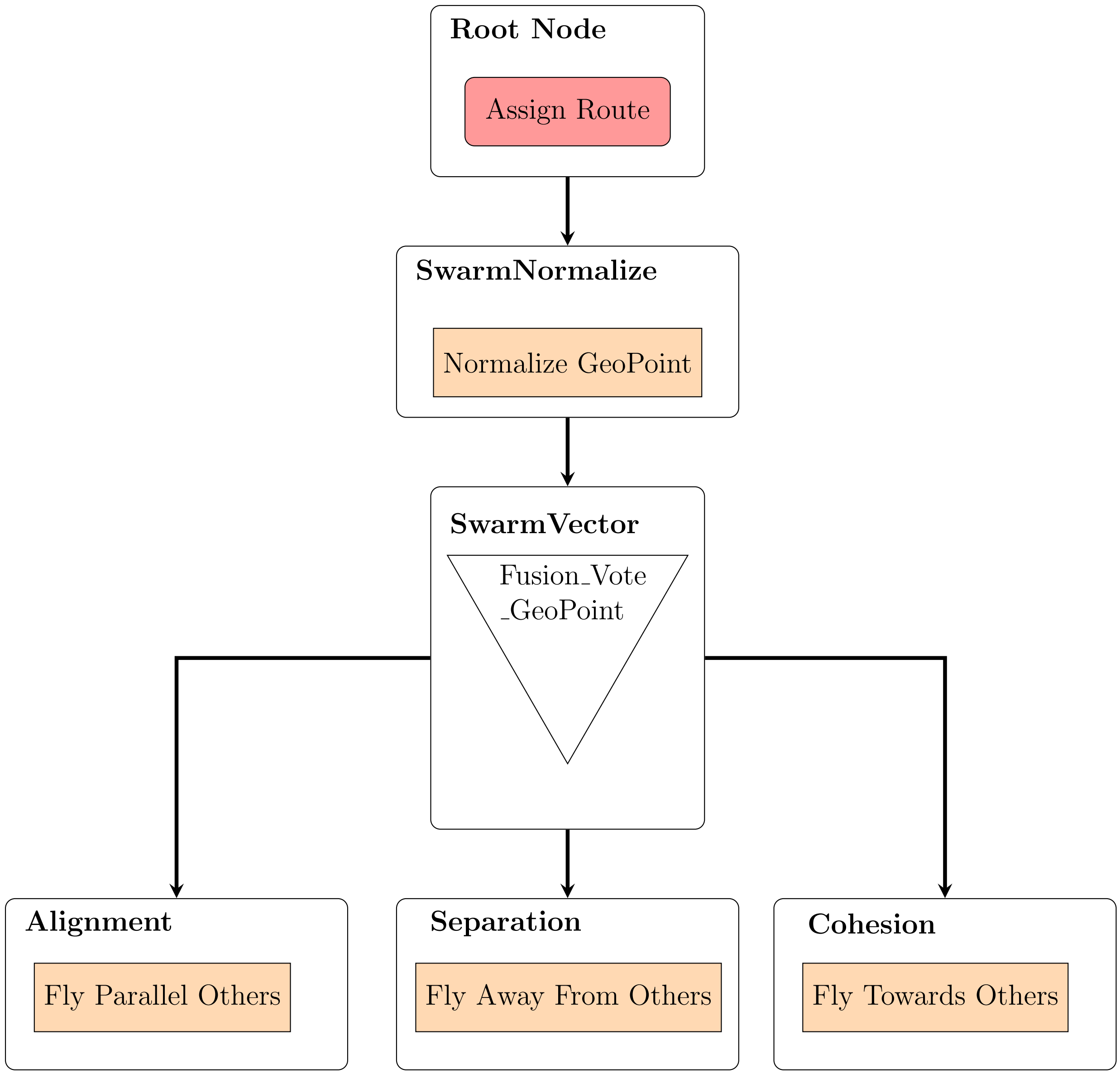


Figure 19. Swarm Agent UBF Tree.

Similar formatting to 4.1 is used for these behaviors' definitions, which are as follows:

Name: Root Node

Description: Maps an action object named 'Route' to cause an agent to follow the provided WsfGeoPoint

Dependencies: Agent needs a mover object and this behavior needs an input with 'Route' named object and WsfGeoPoint:

Name	Route
Priority	n/a
Vote	n/a
Int	index of point to fly to
WsfGeoPoint	A point to fly to

Output: Route to a point

Name: Swarm Normalize

Description: Takes a WsfGeoPoint and moves it to approximately 7 miles from the agent along the same heading to ensure the point is not immediately assumed visited

Dependencies: Input with 'Route' named object and WsfGeoPoint:

Name	Route
Priority	n/a
Vote	n/a
Int	n/a
WsfGeoPoint	A point to fly to

Output: Route to a point

Name	Route
Priority	n/a
Int	1 - Indicates 1st point to fly to
WsfGeoPoint	A point to fly to

Name: SwarmVector

Description: Averages two WsfGeoPoints based on vote value

Dependencies: Agent needs a mover object and this behavior needs an input with 'Route' named action object and WsfGeoPoint of the format:

Name	Route
Priority	n/a
Vote	n/a
Int	1 - Indicates 1st point to fly to
WsfGeoPoint	A point to fly to

Output: Action object with a point to go towards of the format:

Name	Route
Priority	n/a
Vote	effects weight of average attained
Int	1 - Indicates 1st point to fly to
WsfGeoPoint	A point to fly to

Name: Alignment
Description: Provides a WsfGeoPoint in the direction of the average heading respective of North East Down (NED) coordinate system of aircraft within 200km
Dependencies: Agent has a radar which reports enemy and friend tracks
Output: Action object with a point to go towards of the format:

Name	**Route**
Priority	**n/a**
Vote	**1**
Int	**1 - Indicates 1st point to fly to**
WsfGeoPoint	**A point to fly to**

Name: Separation
Description: Provides a WsfGeoPoint away from the center of mass of other aircraft within 50km
Dependencies: Agent has a radar which reports enemy and friend tracks
Output: Action object with a point to go towards of the format:

Name	**Route**
Priority	**n/a**
Vote	**2**
Int	**1 - Indicates 1st point to fly to**
WsfGeoPoint	**A point to fly to**

Name: Cohesion
Description: Provides a WsfGeoPoint towards the center of mass of other aircraft within 200km
Dependencies: Agent has a radar which reports enemy and friend tracks
Output: Action object with a point to go towards of the format:

Name	**Route**
Priority	**n/a**
Vote	**1**
Int	**1 - Indicates 1st point to fly to**
WsfGeoPoint	**A point to fly to**

4.4.1 Boids Scenario Behavior Emergence Discussion.

The swarm UBF tree is applied to five aircraft with the UBF tree from Section 4.3 applied to one air craft. The swarm aircraft are blue and the red aircraft with a number 0 over it is controlled by the UBF tree which seeks a goal from Section 4.3. Four of the aircraft start on top of one another offset by altitude and the other two start some distance away but still in radar range, shown in Figure 20.

Figure 20. Start of Swarm Scenario

As the scenario progresses swarm-like behavior presents itself. First the blue aircraft group together as seen in Figure 21a. Then they turn to rejoin the goal oriented aircraft, because while they grouped up, the goal oriented aircraft continued on its mission, seen in Figure 21b. The swarm aircraft remain grouped with the goal oriented aircraft and revolve around it for the remainder of the simulation, seen in 21c. This is a very basic example of swarm behavior emerging from three simple behaviors used simply as another example of this Unified Behavior Language's capability to create emergent behaviors.

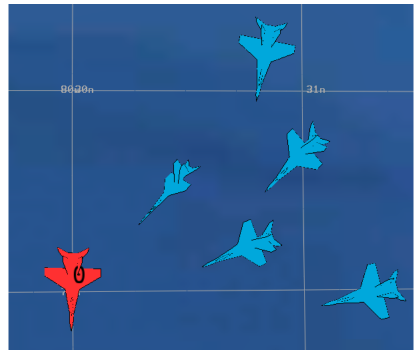
4.5 Combined Scenario

The last scenario created explores the combination of Scenarios 2 and 3 for both BTs and UBF trees identifying the differences in code reuse, tree extendability, and maintainability. In both cases, trees of Scenario 2 are used as the starting point. Then the trees, or behaviors, are augmented with the behaviors from Scenario 3. Thus, two agents are created, one using a UBF tree and the other using a BT. The resulting behavior of the two agents is similar.

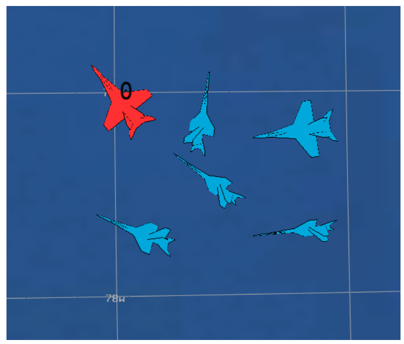
The BT agent implements an identical tree to the Scenario 2 BT seen in Figure

Figure 21. Progression of Swarm Scenario.

(a) Swarm Starting to Group.



(b) Grouped up swarm.



(c) End of Swarm Scenario.

16. However, the two leaf behaviors are augmented to include the behavior's logic from Scenario 3. This results in a behavior that flies to a point while swarming and another behavior that flies away from the obstacle point while swarming. Thus, a BT agent able to fly to a point, avoid obstacles, and swarm with other agents; the execution of said agent is shown in Figure 22.

The UBF agent re-implements the structure from Scenario 3 with the leaf behaviors from Scenario 2. However, the UBF behaviors from Scenario 2 are added as a single combined behavior to the same behavior the Boids behaviors are a descendant. The combined behavior allows "FlyAtPoint" and "FlyAwayFromObstacle" behaviors' votes to be scaled over the swarm behaviors' votes. This is shown in Figure 24. The voting method of the avoidance behavior is modified to increase the closer to the obstacle the agent becomes. Thus, a UBF agent able to fly to a point, avoid obstacles,

Figure 22. Combined BT Agent.

and swarm with other agents shown in Figure 23.

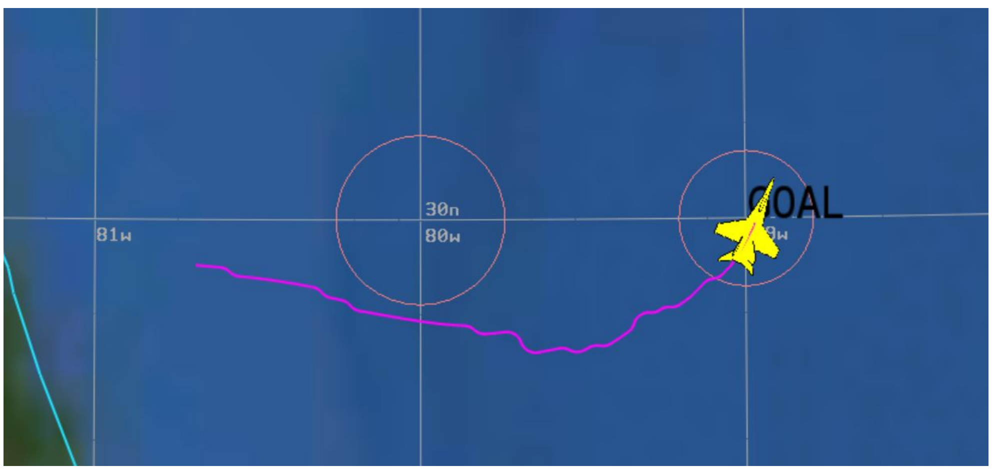


Figure 23. Combined UBF Agent.

4.5.1 Behavior Tree Modification.

In modifying the BT from Scenario 2 with the logic from Scenario 3 multiple tasks are required. The first task is to implement each of the Boids behavior's concepts in both of the BT's behaviors. The second task is to merge the Boids concepts with one another and with both behaviors' original outputs. The third task is to troubleshoot issues with creating custom logic combining these concepts. Finally, the last task is to tune the balance of the various concepts.

4.5.2 UBF Tree Modification.

In augmenting the Scenario 3 UBF structure with the UBF behaviors from Scenario 2, multiple tasks are required. The first task is to combine the Scenario 2 behaviors into a single one to allow their vote values to be scaled over the swarming behaviors. This is done by creating a new behavior whose children are the Scenario 2 UBF behaviors and an execute code block which simply scales the votes of any action it is given by 2. The second task is to change the vote mechanism of the obstacle avoidance to scale based on distance instead of a static assignment. The final task is to tune the various action recommendations.

4.5.3 Modification Comparison.

The BT implementation strategy creates issues. The first issue is the need for expertise, from the second task requiring an understanding of each of the Boids outputs, the original behavior's outputs, and how to combine them. The second issue is the need to duplicate the same code in every behavior that is effected. These result in the another issue, an increased risk of error. An increased risk of error requires the third task, troubleshooting. This BT implementation strategy requires an in depth understanding of the code, increased duplication of code, and increases the risk of errors showing that the BT is difficult to extend and maintain new concepts.

The UBF tree does not share the implementation issues of extendability and maintainability. This is because of the code reuse that UBF's structure enables. The UBF concept of an "arbiter" in combination with the decision based fields of the UBF action objects, priority and vote, are designed to allow any number of behaviors to provide their input for consideration. Hence, UBF behaviors and UBF arbiters are created once and reused. This code reuse decreases the effort required to extend a UBF tree.

The code reuse of UBF also increases the maintainability in comparison to a BT. Adding a concept to a BT requires either a new behavior to accomplish the concepts already implemented as well as the new concept or to augment every existing behavior with the new concept. This scenario, which uses the second strategy of modifying existing behaviors, is not as maintainable as UBF. This is because in UBF a single location may be modified if an update is required of the new concept, whereas every BT's behavior implementing a concept must be modified. If the other strategy for BTs is used, adding a single behavior re-implementing a BT's existing concepts, then an individual has a single point to modify for the new concept, however they now have multiple sections of code to maintain for their old concepts. In a UBF tree the original behaviors are also maintained in a single location. Hence, with either BT extension strategy, updating code requires changes proportional the number of behaviors in the BT; this demonstrates the difficulty of maintaining a BT over that of a UBF tree.

4.6 Summary

This chapter presents multiple scenarios showing that UBF can effectively implement behavioral emergence in AFSIM. The first scenario acts as a proof of concept that UBF can work and replace the existing intelligent agent controller in AFSIM. In order to mitigate the extra work UBF requires, a standardized interface is established. The necessity for tuning behavioral emergence is shown in the second scenario. Behavioral emergence is expressly displayed by the third scenario's implementation of a classic behavioral emergence technique for swarming. The last implementation demonstrates the improved maintainability and extendibility from UBF's code reuse compared to a BT. With these studies one can see UBF is capable of implementing behavioral emergence in AFSIM.

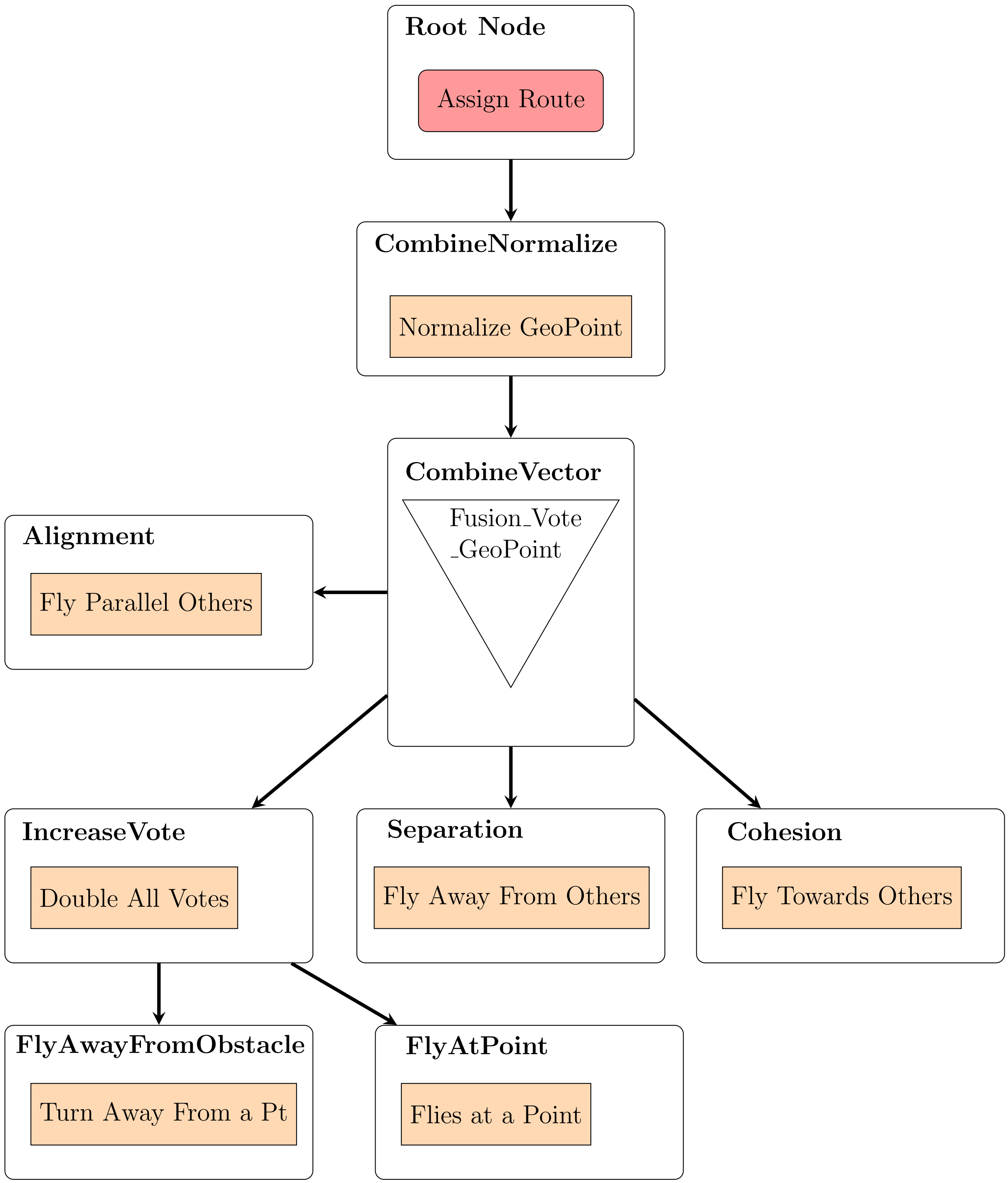


Figure 24. Combined UBF Agent Tree.

V. Results

In Chapter III the Advanced Framework for Simulation, Integration, and Modeling's (AFSIM) scripting language is extended to include the Unified Behavior Framework (UBF). Due to the fact that UBF is able to implement emergent behaviors, this extension must also demonstrate its ability to do so in AFSIM. Applications of this extension are examined through multiple scenarios in Chapter IV to demonstrate the ability to implement this behavioral emergence.

This chapter examines the results of the scenarios that were created and what traits each scenario exemplifies in extending AFSIM with UBF. Furthermore, the demonstrated advancements are discussed to explore the mapping of other frameworks' concepts to this language. Additionally, an inspection of the advantages and disadvantages of this new platform-independent UBF implementation is conducted because past implementations [6, 7, 20] of UBF were static versus the motors of their agent.

5.1 Scenario Results Summary

In order to examine the effectiveness of UBF in AFSIM, four scenarios are used. The first scenario is a proof of concept showing the UBF behaviors are able to replace a behavior tree (BT) in the AFSIM. The second scenario explores the effect of behavioral emergence and the ability to tune it. The third scenario implements a classical example of behavior emergence. The fourth scenario is a comparison of the effort to combine scenario's two and three for BTs versus UBF trees. With these case studies the ability of the UBF plug-in to implement behavioral emergence is displayed.

The first scenario is a proof of concept because it successfully replaces a behavior tree in AFSIM. A scenario is utilized from the AFSIM analyst training course with the BT replaced by a UBF tree and the UBF behaviors are derived from the BT behaviors. The result is that all aircraft on the opposing team are destroyed as shown by Figure 25. This is considered as a successful proof of concept because all of the same aircraft are destroyed. The exact result of missiles used, fuel used, and other details differ slightly because of commands that are available to the BT code blocks, but cause bugs in the plug-in code blocks. Fixing this issue is discussed in Section 6.2.

Figure 25. End of Scenario BT to UBF.

The second scenario examines the benefit and need to tune the emergence of behaviors. It does this by presenting 10 aircraft with different votes and comparing them to a similar behavior tree controlled aircraft and to one another. The result shown in Table 16 identifies that a vote of 2 provides the optimal emergent behavior for this scenario. However, the combination of these behaviors may have a different vote value if the initial conditions are changed, such as adding more obstacles. In that case it would be prudent to scale the vote value based on distance. The table also shows the emergent behavior is able to get to the objective faster than the BT agent while still avoiding the obstacle. Figure 26 shows that the path of the BT agent, blue, is more jagged versus the smooth path of the UBF agent, red with a '2' over it, granted this is a subjective statement and that is why Table 16 is also used for comparison.

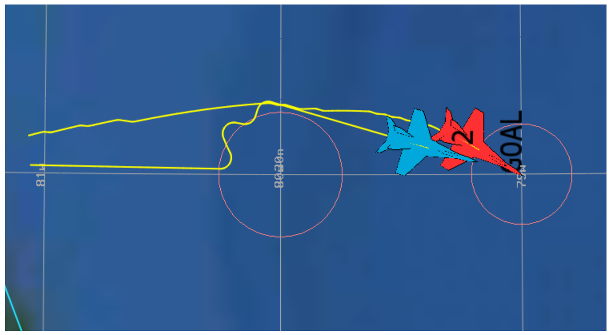


Figure 26. BT Agent vs UBF agent Smoothness.

The third example is another proof of concept showing UBF in AFSIM is able to explicitly create an emergent behavior. This is based on a classical example of behavior emergence. The classical example, Boids [11], details how to create the emergent behavior of swarming and this implementation shows how literally that technique maps to UBF. Only the three tenet behaviors of Boids are implemented, Separation, Cohesion, and Alignment. Next the established techniques from the second scenario are reused to combine the action recommendations. Then a swarming behavior emerges. This is a subjective statement and it not very easily shown via static images, however Figure 20 and 21 display the progression of the scenario. These images show the agents grouping with one another, maintaining alignment after they are grouped, maintaining a minimum distance to one another, and also maintaining their group even though the red aircraft with a '0' over it is controlled by the agent from the second scenario and ignoring the other agents.

The last scenario examines the maintainability and extendability of existing UBF and BT structures with new concepts. The behavior of both agents when employed alone and with other agents acted similarly by pursuing the goal while avoiding the obstacle and flying with the other aircraft to do so. The BT implemented requires custom code which is duplicated proportional to the number of behaviors that are affected. This increases the risk of errors and effort to extend and maintain a BT. The UBF structure's resolution technique compensates for those issues by enabling increased code reuse. In both cases there is a need to balance the old behavior with the new behavior. However, this exposes the fact that maintaining a strategy in a BT has an affect proportional to the number of behaviors this strategy effects. Whereas, in a UBF implementation a single simple behavior can be used to group, scale, and reuse other behaviors to include the new strategy.

These scenarios show a proof of concept, behavioral tuning, replication of a classic emergent behavior, maintainability, and the extensibility as capabilities of UBF in AFSIM. Thus, via these case studies the capability of behavioral emergence is shown in this implementation of the Unified Behavior Framework to AFSIM.

5.2 Coverage of other Languages and Frameworks Concepts

The first investigatory question is how the commands in this Unified Behavior Framework are able to cover the concepts observed in other intelligent agent controllers. This is a secondary goal because adoption of concepts from other frameworks and languages does not necessarily directly affect behavioral emergence. Adopting these secondary concepts has the purpose of allowing the UBF behaviors to be compatible with other concepts in the artificial intelligence community and to help find compatible methods to increase the efficiency of the UBF. Table 17 presents the concepts that were examined, if they were implemented, partially implemented, not implemented, or not compatible with the UBF structure.

In regards to Table 17, items 5-9 and 16 are the commands used in dynamic

Table 17. **Concept Implementations**

#	Concept	Implementation
1.	Pre-Condition Check	YES
2.	Priority	YES
3.	Voting	YES
4.	Name	YES
5.	Expected Effects	Partial
6.	Required Data	Partial
7.	Action Settings	Partial
8.	Initial Conditions	Partial
9.	Goal Achieved	Partial
10.	Behavior Library	Yes
11.	Parameters	Not Implemented
12.	Mental vs Motor	Not Compatible
13.	Global & Persistent Memory	YES
14.	Action Recommendations	YES
15	Children	YES
16.	Reflective Access	Partial
17.	Arbitration Methods	YES
18.	Signature Matching	Partial
19.	Previous Child	Not Compatible
20.	Exit Conditions	Not Compatible
21.	On Entry	Not Compatible
22.	On Exit	Not Compatible
23.	Initialization	YES
24.	Messaging Interface	Not Implemented
25.	Synchronous Flag	Not Implemented
26.	Frequency	YES
27.	Activate/Deactivate	Partial
28.	Execution Time Limits	Not Implemented
29.	Leaf/Composite Behavior Flags	Not Implemented

sequencing and planning of behavior structures. These are declared as partially implemented because the means of accessing them is inefficient, by behavior name then by their field. More functions should be created to dynamically share the fields between parent and child behaviors and to query for UBF behaviors by those fields directly.

Parameter lists, signature matching, and leaf vs composite behavior flags, items

11, 18, 29 respectively, are not implemented, however they have benefits that are compatible with this implementation. Parameter lists allow for increased re-usability of behaviors and leaf vs composite behavior flags allow for decreased execution time of the scenario via behavior output reuse, respectively. In Section 6.2, suggestions are made how to implement them both. The suggestion for implementing parameter lists is akin to command line argument usage, providing a dynamic vector of parameters to behaviors. This is not compatible with signature matching. If further work is done to implement parameters and/or dynamic sequencing components, then signature matching should be re-examined because those concepts provide many aspects that can be part of a behavior's signature.

The other three concepts that were not implemented are for a variety of reasons; these are a messaging interface, execution time limits, and synchronization flags. The messaging interface is not implemented because it provides a simple convenience that can be replicated by code within an execute code block of a UBF behavior. The execution time limit is not implemented because the scope is for discrete event simulations. Synchronization flags are not used because AFSIM provides various other teaming techniques and UBF behaviors themselves allow any team oriented behaviors to submit their action recommendations if desired.

Thus, users of this implementation or readers who may try to recreate their own version of a UBF language should consider not only the items that were implemented, but also the items that were partially or not implemented. This is because other implementations may provide compatible optimizations and capabilities that the original goal may not.

5.3 Platform Independent UBF Discussion

The second advancement demonstrated through this thesis regards the platform independence of this UBF implementation. The platform independent implementation of UBF refers to the action recommendations having various generic value fields within them, being the int, double, WsfRoute, WsfTrack, WsfGeoPoint, and string value fields. This is in comparison to other UBF implementations that have action objects with fields directly related to motor settings of the agent to which they are applied [6, 7, 20].

First, a disadvantage is forcing users to implement a Map_To_Action code block when other UBF implementations do this for the user. This can be mitigated by establishing reusable Map_To_Action code blocks. This does lead to the need for teams to establish and share the input and output requirements for their behaviors. When a behavior is not designed specifically for the input requirements of another, a user may add an intermediate node to translate the values to the required format.

While a user may have more work initially, they are also freed from a defined subset of possible outputs. This freedom allows compatibility for custom messages to be used between behaviors. It allows for future motors or effectors to be added to an agent that can be utilized and planned for by UBF structures without needing to re-compile the framework's or plug-in's code to add new fields. Platform independent action recommendations also allow generic actions to be used so identical behaviors can be used on completely different platforms, i.e. a path finding behavior can be used on a tank or boat and only need a small translation behavior to be added between it and the respective Map_To_Action code block. This is an advantage over other UBF implementations because they use predefined action objects.

It is worth noting that this implementation of UBF is not completely independent of the sensors on the platform. AFSIM mitigates this fact by providing generic components with which platforms may have to interface. An example of this is a "master track list" that behaviors may use. A user should use the master track list instead of trying to access a radar component directly because another platform may implement the radar with different capabilities, by a different name, or not at all.

5.4 Summary

This chapter examines the results of the case studies and discusses the demonstrated advancements of this implementation. UBF in AFSIM effectively replaces behavior trees and effectively creates emergent behaviors and effectively provides the ability to tune them. This extension implements features of many other frameworks, however there are concepts not implemented that would be beneficial in a future iteration of this work. Finally, the platform independent implementation of UBF causes additional work initially for intelligent agent creators, it allows implementation of behaviors on vastly different agents with simple translation behaviors, and it detaches the UBF plug-in code from the implementation of new effectors. With these discussions one can see UBF in AFSIM is able to provide behavioral emergence, the concepts currently implemented and those that the plug-in may still benefit from, and the benefits and difficulties of this platform independent implementation of the UBF.

VI. Conclusions

The Advanced Framework for Simulation, Integration, and Modeling (AFSIM) provides various tools for intelligent agent creation, but it does not explicitly provide the means for behavioral emergence. This thesis demonstrates the use of a plug-in built on top of the Unified Behavior Framework (UBF) expressly to provide the use of behavior emergence to AFSIM via an extension of the scripting language. This chapter reiterates the benefits of using UBF in AFSIM and discusses the future efforts that this thesis brings to light.

6.1 Recommendations

The AFSIM program office should look into a slight modification to the current behavior tree (BT) system to increase its capabilities and intuitiveness. In researching various other BT implementations [5, 18] the 'running' return type for behavior node was observed, whereas AFSIM only has 'true' and 'false' (failure) for the return types of a behavior node's pre_condition code block. The addition of this flag increases intuitiveness by reducing the reliance of behaviors on one another to check criteria relating to them and depending on the node to which they are a child. This modification is backwards compatible with previously created scripts because the other return types still function the same, only newly created behavior trees that implement it would need consider it.

6.2 Future Work Discussion

The implementation of UBF reveals additional commands that are needed to fully implement some of the originally intended concepts and new ideas on how to implement commands that are not included. The initial concept to be expanded on is the ability to sequence and dynamically construct UBF trees. The included commands merely modify (add or remove) the children of a behavior from within that behavior and examine the various fields inspired by Duffey [24]. This should be expanded to allow external access to the UBF tree; a possible solution is to allow another tag in the construction of the UBF tree that gives a name to the position it is placed at and external code could then interact with the UBF tree by modifying named positions in the tree.

The next concept not implemented is parameters for behaviors similar to function calls in many programming languages. This is a powerful concept because it allows UBF tree builders to directly add specific functionality without having to modify the underlying code. Also, this generalization of behaviors allows re-use of the same behavior for multiple tasks. With the level of knowledge this thesis has provided it is apparent that this could be implemented via a parameter list after the behavior name in the UBF tree and accessed inside the behavior similar to command line arguments in the programming language C. This does require a behavior creator to add additional checks for properly formatted and existing data to avoid run time errors, but it is possible. A more graceful and standard method is no doubt possible if done by a veteran AFSIM developer.

Another concept that provides an optimization is the by-product of the differentiation of leaf and composite behaviors. This by-product is the ability to reuse behavior outputs without re-computing the entire behavior if included in multiple places in a single UBF tree. This is a by-product of the leaf vs composite behavior concept because a leaf is always able to be reused whereas a composite behavior may not always be reused, but a composite may sometimes be reused. Hence, the true concept may be applied to a UBF behavior by a tag indicating the output is reusable, these tags being tracked by a root node, and the tagged nodes remembering their output if called again in the same iteration. This optimization could help a real time implementation of UBF maintain its reactivity.

Another change for real time implementation is to re-implement how the active UBFBehavior and active UBFArbiter is accessed. Currently the active UBFBehavior and UBFArbiter expose their contents through singleton objects, global static variables, for each. The new commands implemented simply access those variables; this is not thread safe and could also cause issues if AFSIM runs in real time. To fix this an expert AFSIM developer needs to change the commands to point to the calling UBFBehavior or UBFArbiter instead of the global static variable used. This work around was used due to the complexity of integrating with the AFSIM code.

This implementation has a couple of other issues due to the complexity of integrating with the AFSIM code that should be fixed for future work. The first is an inability to access some functions of AFSIM that are very useful inside the new code blocks, i.e. drawing shapes and accessing the global simulation object. Working with the AFSIM help desk has alleviated some of the issues but it appears there is an unknown nuance with the inherited traits that may need to be manually set. A suggested approach to attempt to fix this is to change the type of processor that is being used as the parent and eventually to implement UBF behaviors as their own entities, not as processor, similar to how behavior tree nodes are currently done in AFSIM.

The next issue is in regards to the action object's value fields. The WsfRoute object loses its details when passed up the tree. Because of this difficulty, a generic object pointer was not implemented. However, implementing a generic object pointer could increase the usefulness of action objects. This could be from passing complex data sets as vectors or passing any other object in AFSIM as a value. This change could allow for increased complexity in behavior communication.

Even though the current implementation could be improved by various commands and fixes there are other ways to continue researching increases to the capabilities of AFSIM via the current implementation. This starts with generating a large library of UBF behaviors. This allows for further exploration of the behavioral emergence landscape and could be directly useful to AFSIM analysts simulating real behaviors of pilots. The current implementation can also expose the vote fields to artificial intelligence (AI) learning techniques such as simulated annealing or neural networks that could search for and learn the optimal vote values or strategies for UBF trees in given scenarios, thus, teaching an agent how to fly missions.

A final recommendation to modify is a re-examination of the Frequency tag. This is because the current implementation allows a behavior to execute at a max speed defined by the Frequency with no output being produced in the interim. A possible way to change this would be to store the action outputs and reuse them whenever called in the interim, providing an efficiency increase.

6.3 Conclusions.

The plug-in based on the UBF provides the means to effectively implement behavioral emergence in the AFSIM. This is shown through 4 case studies of scenarios shown in Chapter IV. These scenarios show proof of concept that a UBF tree may replace a BT in AFSIM, that via tuning a UBF agent can achieve a goal faster, slower, or worse than a BT, that simple behaviors can create an emergent behavior, and that UBF improves maintainability and extendability through code reuse. This satisfies the main objective of this thesis.

This UBF implementation includes the concepts from various frameworks so that it is more than just a container around UBF. This is examined in Chapter III and Section 6.2. This is accomplished to provide the maximum number of capabilities to users. Maximizing capabilities prevents users from abandoning it for another that may have the components they are familiar with or need and it provides optimizations wherever possible. There are concepts UBF can benefit from that were partially or not implemented. These concepts are integration with sequencers and planners, parameterizing behaviors for increased re-usability, and identifying behaviors as static so the tree can be optimized to reuse a behaviors output in multiple locations without recalculation being required.

Other implementations of UBF use action recommendation objects with fields directly related to the motors of a platform. This allows a user of the framework to ignore mapping the actions to motors in the root of the tree because a developer already accomplished this for them. This UBF implementation uses generic action recommendations which force users to map them to outputs at the root of the tree. This allows for platform independence which increases the re-usability of behaviors and the ability to implement effectors which have not been invented yet.

6.4 Significance

The goal of any intelligent agent controller is to simulate intelligence in the agent on which it is implemented. This thesis provides a control structure that increases the complexity of behaviors that are possible on an agent without causing a large increase in complexity of the controller structure. This is done via the emergence of behaviors.

In the AFSIM, increasing the complexity of an agent typically involves an analyst making a new behavior which overlaps with other behaviors. This is a duplication of effort. Also, if a new strategy or tactic is invented then new behaviors are needed or any affected behavior requires modification. Those behaviors overlap the situations they check for with other previously created behaviors. The same effects can be obtained with emergent behaviors via tuning or adding in the new tactics, without overlapping considerations, which affect the resulting behavior based on their voting mechanisms.

Behavioral emergence in AFSIM can save money and analysts' time by reducing the time to create new behaviors and simulate new strategies. It can allow new capabilities that were not possible in BTs. Increased capabilities can increase the significance and confidence in the results of scenarios by possibly bringing them closer to reality.

6.5 Summary

This thesis implements UBF as a dynamic link library, a plug-in, for AFSIM providing the capability of behavioral emergence. This allows complex behaviors to emerge from simple components. The extension considers other implementations in an attempt to maximize the capabilities. However, not all of the compatible concepts are included, but these concepts are identified for any future implementation if desired. Finally, this thesis provides a new look at the action recommendation concept seen in the UBF. This UBF implementation creates more initial work for a user but provides them greater flexibility and detaches the action object from needing to be updated every time a new motor effector is added to AFSIM platforms.

Appendix A. Implementation C++ Code

This appendix includes the various C++ files that are used to create the Unified Behavior Language. The C++ files used to add commands to the AFSIM language and register the plug-in are omitted because those files simply show a mapping of function names to call functions in their respective classes, which are included.

1.1 Header Files

The C++ header files follow this page.

```
#pragma once
#include "WsfVariable.hpp"
#include <vector>
class InputTree
{
public:
    std::vector<InputTree*> mChildren;
    InputTree() {};
    InputTree(std::string behaviorName):mName(behaviorName)
    {

    };
    ~InputTree() {};
    WsfVariable<WsfStringId> GetName() { return mName; }
private:
    WsfVariable<WsfStringId> mName;

};
```

```cpp
/**
* @title UBFAction.hpp
* @Author Jeff Choate
* @email Jeff.lee.choate@gmail.com or Jeffrey.choate@us.af.mil
* @description This file defines the UBFAction class for Capt Jeffrey Choate's
* Thesis work at the Air Force Institute of Technology, 2015-2017.
* @usage A UBFAction object is used as a communication device for UBFBehaviors and UBFArbiters.
* @Modified: 28 Jan 2017
* @Change_Log:
* 28 Jan 2017: Comments
*/
#ifndef UBF_ACTION_HPP
#define UBF_ACTION_HPP

#include <string>
class WsfRoute;
class WsfGeoPoint;
class WsfTrack;

class UBFAction
{
public:
    ~UBFAction();

    UBFAction();
    UBFAction(std::string oName, int oPriority, int oVote, std::string oValStr);
    UBFAction(std::string oName, int oPriority, int oVote, WsfRoute * oValRoutePtr);
    UBFAction(std::string oName, int oPriority, int oVote, int oValInt);
    UBFAction(std::string oName, int oPriority, int oVote, double oValDbl);
    UBFAction(std::string oName, int oPriority, int oVote, WsfGeoPoint * oValWsfGeoPointPtr);
    UBFAction(std::string oName, int oPriority, int oVote, WsfTrack * oValWsfTrackPtr);
    UBFAction(UBFAction * oUBFActionPtr);

    //Getters for all traits
    int GetSourceID();
    std::string GetName();
    int GetPriority();
    int GetVote();

    //Getters for all individual values
    std::string GetValueString();
    WsfRoute * GetValueWsfRoutePtr();
    int GetValueInt();
```

```cpp
    double GetValueDouble();
    WsfGeoPoint * GetValueWsfGeoPointPtr();
    WsfTrack * GetValueWsfTrackPtr();

    //Setters for traits
    void SetName(std::string oName);
    void SetPriority(int oInt);
    void SetVote(int oInt);


    //Setters for all individual values
    void SetValueString(std::string oValString);
    void SetValueWsfRoutePtr(WsfRoute * oValWsfRoutePtr);
    void SetValueInt(int oValInt);
    void SetValueDouble(double oValDouble);
    void SetValueWsfGeoPointPtr(WsfGeoPoint * oValWsfGeoPointPtr);
    void SetValueWsfTrackPtr(WsfTrack * oValWsfTrackPtr);

private:

    //traits of an Action
    int sourceID = -1;
    std::string actionName = "ERROR NEVER INITIALIZED";
    int priority = -1;
    int vote = -1;

    //Possible Values
    std::string valueString = "ERROR NEVER INITIALIZED";
    WsfRoute * valueWsfRoutePtr = nullptr;
    int valueInt = -1;
    double valueDouble = -1.0;
    WsfGeoPoint * valueWsfGeoPointPtr = nullptr;
    WsfTrack * valueWsfTrackPtr = nullptr;
};

#endif
```

```cpp
/**
* @title UBFActionList.hpp
* @Author Jeff Choate
* @email Jeff.lee.choate@gmail.com or Jeffrey.choate@us.af.mil
* @description This file defines the UBFActionList class for Capt Jeffrey Choate's
* Thesis work at the Air Force Institute of Technology, 2015-2017.
* @usage A UBFBehaviors and UBFArbiters inherit this device.
* @Modified: 28 Jan 2017
* @Change_Log:
* 28 Jan 2017: Comments
*/
#ifndef UBF_ACTIONLIST_HPP
#define UBF_ACTIONLIST_HPP

#include <string>
#include <vector>
class UBFAction;

class UBFActionList
{
public:
    UBFActionList();

    //Note: overriden by UBFArbiter ,so they do not add to their inherited mActions vector
    virtual void Add_Action(UBFAction * newAction);

    //various get methods to retrieve actionLists from the this actionList's vector of actions
    UBFActionList * Get_Actions_By_Exact_Name(std::string byName);//returns all with a specific name
    UBFActionList * Get_Actions_By_partial_Name(std::string byName);//returns all with a specific string in the name
    UBFActionList * Get_Actions_By_Exact_Priority( int byPriority);//returns all by a specific priority
    UBFActionList * Get_Actions_By_Min_Priority( int byPriority);//returns all by a specific priority
    UBFActionList * Get_Actions_by_type_String();//returns all actions which were assigned a string
    UBFActionList * Get_Actions_by_type_WsfRoute();//returns all actions of type WsfGeoRoute
    UBFActionList * Get_Actions_by_type_Int();//returns all actions which were assigned an int
    UBFActionList * Get_Actions_by_type_Double();//returns all actions which were assigned a double
    UBFActionList * Get_Actions_by_type_WsfGeoPoint();//returns all actions of type WsfGeoPoint
    UBFActionList * Get_Actions_by_type_WsfTrack();//returns all actions of
```

```
        type WsfTrack, will return an empty ActionLists object if none are there
    UBFActionList * Get_Actions_Unique_Top_Priorities();//returns one action for each unique name with the highest priority

    //Iterator/Array inspired methods to retrieve UBFAction objects
    UBFAction * First();//returns the first UBFAction * from the vector, null if vector is empty
    UBFAction * Last();//returns the last UBFAction * from the vector, null if vector is empty
    UBFAction * Next();//returns the next UBFAction * from the vector each subsequent call, returns null if at the end
    UBFAction * ByIndex(int i);//returns the UBFAction * at the designated index (zero based array syntax), returns null if out of range
    void Next_Restart();//sets the iterator used by Next() back to the start to allow a user to restart searches using the same ActionList object
    int Size();//returns the number of UBFActions in this UBFActionList object

    bool Erase_Action_By_Name(std::string oName);//removes one Action of the specified name from the actionsList of this object

protected:
    std::vector<UBFAction*> mActions;//Storage device for the UBFAction objects this class is for
    int iteratorForNextMethods = 0;//Used when treating this class as a list
};

#endif

```

```cpp
/**
* @title UBFArbiter.hpp
* @Author Jeff Choate
* @email Jeff.lee.choate@gmail.com or Jeffrey.choate@us.af.mil
* @description This file defines the UBFArbiter class for Capt Jeffrey Choate's
* Thesis work at the Air Force Institute of Technology, 2015-2017.
* @usage This object is used as a mechanism to store a script which
* makes decisions between UBFAction objects.
* @Modified: 28 Jan 2017
* @Change_Log:
* 28 Jan 2017: Comments
*/
#ifndef UBF_ARBITER__HPP
#define UBF_ARBITER_HPP

//include because an Arbiter IS-A Processor and IS-A UBFActionList in this implementation
#include "processor\WsfProcessor.hpp"
#include "UBFActionList.hpp"

//forward declarations of object types I hold pointers to
class UtScript;
class UBFAction;
class UBFBehavior;
class UBFArbiter:public WsfProcessor,public UBFActionList
{
public:
    //Override-Now modifies the vector which is passed forward and not the
    //inherited vector
    void Add_Action(UBFAction * newAction);

    UBFArbiter(WsfScenario& aScenario);
    UBFArbiter(const UBFArbiter& oArbiter);

    static UBFArbiter * getInstancePtr();
    static void setInstancePtr(UBFArbiter * ptr);

    virtual UBFArbiter* Clone() const
    {
        return new UBFArbiter(*this);
    }
    ~UBFArbiter();
    std::vector<UBFAction*> Process(std::vector<UBFAction*> inputActions);
    bool ProcessInput(UtInput& aInput);
    void SetContext(WsfScriptContext*  newContextPtr);
    std::vector<UBFAction*> newActions;
private:

```

```
    static UBFArbiter * staticUBFArbiterPtr; //used in order to allow script's to find behaviors and find the active behavior
    //Attributes
    UtScript*           mExecuteScriptPtr;//The script this class is built around
    WsfScriptContext*    mContextPtr; //ptr associating PLATFORM and script
    UBFBehavior * mBehavior = nullptr;//behavior this Arbiter is assigned to...not used yet
    UtScript * mProcessScriptptr = nullptr; //this holds the script which will execute when
                                        // Inherited via WsfProcessor
};


#endif

```

```cpp
/**
* @title UBFBehavior.hpp
* @Author Jeff Choate
* @email Jeff.lee.choate@gmail.com or Jeffrey.choate@us.af.mil
* @description This file defines the UBFBehavior class for Capt Jeffrey Choate's
* Thesis work at the Air Force Institute of Technology, 2015-2017.
* @usage This object is used as a mechanism to store scripts and various
* other traits of a behavior and control execution of said behaviors
* @Modified: 28 Jan 2017
* @Change_Log:
* 28 Jan 2017: Comments
*/
#ifndef UBF_BEHAVIOR_HPP
#define UBF_BEHAVIOR_HPP


#include <string>
#include <vector>
//include because an Behavior IS-A Processor and IS-A UBFActionList in this implementation
#include "processor\WsfProcessor.hpp"
#include "UBFActionList.hpp"

//forward declarations of object types I hold pointers to
class UBFAction;
class UBFArbiter;
class UtScript;
class InputTree;

class UBFBehavior:public WsfProcessor, public UBFActionList
{
public:
    //getter for singleton way to access currently operating behavior
    static UBFBehavior * getInstancePtr();
    //setter for singleton way to access currently operating behavior
    static void setInstancePtr(UBFBehavior * ptr);

    UBFBehavior(WsfScenario& aScenario);
    UBFBehavior(const UBFBehavior& mUBFBehavior);
    virtual UBFBehavior* Clone() const
    {
        return new UBFBehavior(*this);
    }

    //Assigns pointers based on strings found during the ProcessInput call
    virtual bool Initialize(double aSimTime);
    ~UBFBehavior();
    //parses script into values for this UBFBehavior
```

```
    bool ProcessInput(UtInput& aInput);
    bool BuildOwnBehaviorTree(WsfScriptContext * newScriptContextPtr, int depthOfTree);
    //Called by AFSIM code for ROOT behavior only, Used to
    //call mExecute of a behavior
    void Update(double aSimTime);

    //handles this behaviors execution; basically calls its execute script and passes
    //its recommended actions up to a parent arbiter/behavior
    std::vector<UBFAction*> mExecute(int depth, double aSimTime);

    WsfScriptContext* GetContextPtr();
    void SetContextPtr(WsfScriptContext* newContextPtr);
    void SetParentContextPtr(WsfScriptContext * newContextPtr);
    bool Add_Behavior(std::string newBehaviorName);
    bool Add_Behavior(UBFBehavior * newChild);
    bool Remove_Behavior(std::string deleteName);

    //Sequencer/Planner used methods
    UBFBehavior * Find(std::string oBehaviorName);
    void Add_Adder_Post_Condition(std::string newCondition);
    void Add_Remove_Post_Condition(std::string newCondition);
    void Add_Action_Setting(std::string newCondition);
    void Add_Required_Data(std::string newCondition);
    void Add_Initial_Condition(std::string newCondition);
    void Set_GoalAchieved(std::string newGoal);

    bool Adder_Post_Condition_Exists(std::string oCondition);
    bool Remove_Post_Condition_Exists(std::string oCondition);
    bool Action_Setting_Exists(std::string oSetting);
    bool Required_Data_Exists(std::string oData);
    bool Initial_Condition_Exists(std::string oCondition);

    std::string Get_Adder_Post_Condition_byIndex( int index);
    std::string Get_Remove_Post_Condition_byIndex( int index);
    std::string Get_Action_Setting_byIndex( int index);
    std::string Get_Required_Data_byIndex( int index);
    std::string Get_Initial_Condition_byIndex( int index);
    std::string Get_GoalAchieved();

    int Adder_Post_Condition_Size();
    int Remove_Post_Condition_Size();
    int Action_Setting_Size();
    int Required_Data_Size();
    int Initial_Condition_Size();
private:
    //This variable controls how frequently the behavior's
    //execute/Arbiter blocks may be called; default is always call.
```

```
    //May not be called more frequently than the root update_interval.
    double executeFrequency = -1;
    //This variable works with the executedFrequency to control
    //how often a behavior may be executed.
    double timeLastExecuted = 0;
    //Flag controlling if time of code block execution is reported
    bool debug_time = false;
    //stores strings representing the environmental initial conditions
    std::vector<std::string> mInitialConditions;
    //stores strings representing the motors this behavior effect
    std::vector<std::string> mActionSettings;
    //stores strings representing the data required to be available to
    //this behavior for it to execute: Radar, or processor with name task -
      mgr...
    std::vector<std::string> mRequiredData;
    //stores strings representing the tasks or restraints this behavior
    //adds to a platform: i.e. bomb_doors_are_open
    std::vector<std::string> mPost_Conditions_Add;
    //stores strings representing the tasks or restraints this behavior
    //removes from a platform: i.e. bomb_doors_are_closed
    std::vector<std::string> mPost_Conditions_Remove;
    //stores string representing the goal of this behavior
    std::string mGoalAchieved;

    //used in order to allow scripts to find the active behavior
    static UBFBehavior * staticUBFBehaviorPtr;
    void ExecuteMapToOutputs();
    //default value is arbitrary, used to prevent loops in trees
    //i.e. child being its own parent
    int maxTreeDepth = 30;
    UtScript*                               mMapToActionScriptPtr;
    UtScript*                               mExecuteScriptPtr;
    UtScript*                               mPreConditionScriptPtr;
    WsfScriptContext*                       mContextPtr;
    //tracks if InputTree was used to build UBFBehavior tree of children
      pointers
    bool mbehaviorTreeBuilt = false;
    //tracks if an actual pointer was assigned to Arbiter
    bool mArbiterAssigned = false;
    bool AssignMyArbiter(WsfScriptContext* newScriptContextPtr);
    bool AddChildrenToChildren(InputTree* parent, UBFBehavior* tempBehavior,
        WsfScriptContext * newScriptContextPtr, int depthOfTree);
    bool StoreChildren(InputTree * parentPtr, UtInput & aInput);
    WsfSimulation * GetSimulation();
    UtScriptContext * GetScriptAccessibleContext();
    const char * GetScriptClassName();
    WsfPlatform * OwningPlatform();
    //stores children names between ProcessInput state and Initialize stage
    std::vector<InputTree*> mProcessInputChildren;
```

```
    //stores Arbiter name between ProcessInput stage and Initialize stage
    WsfVariable<WsfStringId> mArbiterName;
    //pointer to UBFArbiter used by this UBFBehavior
    UBFArbiter * Arbiter = nullptr;
    //holds list of UBFBehavior pointers this UBFBehavior is parent to
    std::vector<UBFBehavior*> mUBFChildren;
};

#endif

```

1.2 C++ Files

The C++ code files follow this page.

```cpp
#include "UBFAction.hpp"
#include <iostream>
#include "mover/WsfRoute.hpp"
#include "WsfGeoPoint.hpp"
#include "WsfTrack.hpp"

UBFAction::~UBFAction()
{
    //All instances of UBFAction use the UtScriptRef::cManage flag when created to allow AFSIM to manage them
    //It is assumed that subobjects of UBFActions are created in AFSIM script and are hence also managed by AFSIM
    //if those objects are cloned via the UBFAction copy constructor then it is unknown if AFSIM continues to
    //manage those objects or not: these as WsfRoute, WsfTrack, WsfGeoPoint.
}

UBFAction::UBFAction()
{

}

UBFAction::UBFAction(std::string oName, int oPriority, int oVote, std::string oValStr)
{
    actionName = oName;
    priority = oPriority;
    valueString = oValStr;
    vote = oVote;
}

UBFAction::UBFAction(std::string oName, int oPriority, int oVote, WsfRoute * oValRoutePtr)
{
    actionName = oName;
    priority = oPriority;
    valueWsfRoutePtr = oValRoutePtr;
    vote = oVote;

}

UBFAction::UBFAction(std::string oName, int oPriority, int oVote, int oValInt)
{
    actionName = oName;
    priority = oPriority;
    valueInt = oValInt;
    vote = oVote;

}
```

```

UBFAction::UBFAction(std::string oName, int oPriority, int oVote, double oValDbl)
{
    actionName = oName;
    priority = oPriority;
    valueDouble = oValDbl;
    vote = oVote;

}

UBFAction::UBFAction(std::string oName, int oPriority, int oVote, WsfGeoPoint * oValWsfGeoPointPtr)
{
    actionName = oName;
    priority = oPriority;
    valueWsfGeoPointPtr = oValWsfGeoPointPtr;
    vote = oVote;


}

UBFAction::UBFAction(std::string oName, int oPriority, int oVote, WsfTrack * oValWsfTrackPtr)
{
    actionName = oName;
    priority = oPriority;
    valueWsfTrackPtr = oValWsfTrackPtr;
    vote = oVote;

}


UBFAction::UBFAction(UBFAction * oUBFActionPtr)
{
    sourceID = oUBFActionPtr->sourceID;
    actionName = oUBFActionPtr->actionName;
    priority = oUBFActionPtr->priority;
    //Possible Values
    valueString = oUBFActionPtr->valueString;
    valueInt = oUBFActionPtr->valueInt;
    valueDouble = oUBFActionPtr->valueDouble;
    vote = oUBFActionPtr->vote;
    if (oUBFActionPtr->valueWsfGeoPointPtr!=nullptr)
    {
        valueWsfGeoPointPtr = oUBFActionPtr->valueWsfGeoPointPtr->Clone();
    }
    if (oUBFActionPtr->valueWsfRoutePtr != nullptr)
    {
```

```
        valueWsfRoutePtr = oUBFActionPtr->valueWsfRoutePtr->Clone();

    }
    if (oUBFActionPtr->valueWsfTrackPtr != nullptr)
    {
        valueWsfTrackPtr = oUBFActionPtr->valueWsfTrackPtr->Clone();

    }
}

int UBFAction::GetSourceID()
{
    return sourceID;
}

std::string UBFAction::GetName()
{
    return actionName;
}

int UBFAction::GetPriority()
{
    return priority;
}

int UBFAction::GetVote()
{
    return vote;
}

std::string UBFAction::GetValueString()
{
    return valueString;
}

WsfRoute * UBFAction::GetValueWsfRoutePtr()
{
    return valueWsfRoutePtr;
}

int UBFAction::GetValueInt()
{
    return valueInt;
}

double UBFAction::GetValueDouble()
{
    return valueDouble;
}
```

```

WsfGeoPoint * UBFAction::GetValueWsfGeoPointPtr()
{
    return valueWsfGeoPointPtr;
}

WsfTrack * UBFAction::GetValueWsfTrackPtr()
{
    return valueWsfTrackPtr;
}

void UBFAction::SetName(std::string oName)
{
    actionName = oName;
}

void UBFAction::SetPriority(int oPriority)
{
    priority = oPriority;
}

void UBFAction::SetVote(int oVote)
{
    vote = oVote;
}

void UBFAction::SetValueString(std::string oValString)
{
    valueString = oValString;
}

void UBFAction::SetValueWsfRoutePtr(WsfRoute * oValWsfRoutePtr)
{
    valueWsfRoutePtr = oValWsfRoutePtr;
}

void UBFAction::SetValueInt(int oValInt)
{
    valueInt = oValInt;
}

void UBFAction::SetValueDouble(double oValDouble)
{
    valueDouble = oValDouble;
}

void UBFAction::SetValueWsfGeoPointPtr(WsfGeoPoint * oValWsfGeoPointPtr)
{
    valueWsfGeoPointPtr = oValWsfGeoPointPtr;
```

```
}

void UBFAction::SetValueWsfTrackPtr(WsfTrack * oValWsfTrackPtr)
{
    valueWsfTrackPtr = oValWsfTrackPtr;
}

```

```
///Choate UBF Action CPP file
#include "UBFActionList.hpp"
#include "UBFAction.hpp"
#include <iostream>
#include <vector>
#include <string>
#include "WsfGeoPoint.hpp"

/**
* Default constructor
*/
UBFActionList::UBFActionList()
{
}

/**
* This function adds a UBFACtion pointer to this objects vector of UBFAction pointers
*/
void UBFActionList::Add_Action(UBFAction * newAction)
{
    mActions.push_back(newAction);
}

/**
* Gets actions with names who exactly match the input string from this object's vector of UBFAction objects
* @Param string byName is the name to be matched against
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_By_Exact_Name(std::string byName)
{
    UBFActionList * newList = new UBFActionList();
    for each (UBFAction * tempActionPtr in mActions)
    {
        if (tempActionPtr->GetName().compare(byName)==0)//then this Action matches the asked for name
        {
            newList->Add_Action(tempActionPtr);
        }
    }
    return newList;
}

/**
* Gets actions with names who partially match the input string from this object's vector of UBFAction objects
* @Param string byName is the name to be matched against
```

```
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_By_partial_Name(std::string byName)
{
    UBFActionList * newList = new UBFActionList();
    for each (UBFAction * tempActionPtr in mActions)
    {
        if (tempActionPtr->GetName().find(byName) != std::string::npos)//then this Action has a substring which matches the asked for string
        {
            newList->Add_Action(tempActionPtr);
        }
    }
    return newList;

}


/**
* Gets actions with priorities who exactly match the input integer from this object's vector of UBFAction objects
* @Param int priority  is the integer to be matched against
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_By_Exact_Priority(int byPriority)
{
    UBFActionList * newList = new UBFActionList();
    for each (UBFAction * tempActionPtr in mActions)
    {
        if (tempActionPtr->GetPriority() == byPriority)//then this Action matches the asked for priority so add it
        {
            newList->Add_Action(tempActionPtr);
        }
    }
    return newList;
}


/**
* Gets actions with priorities atleast as large as the input integer from this object's vector of UBFAction objects
* @Param string byName is the name to be matched against
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_By_Min_Priority(int minPriority)
```

```
{
    UBFActionList * newList = new UBFActionList();
    for each (UBFAction * tempActionPtr in mActions)
    {
        if (tempActionPtr->GetPriority() >= minPriority)//then this Action matches the asked for priority so add it
        {
            newList->Add_Action(tempActionPtr);
        }
    }
    return newList;
}



/**
* Gets actions which have had a string value set for them from this object's vector of UBFAction objects
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_by_type_String()
{
    UBFActionList * newList = new UBFActionList();
    for each (UBFAction * tempActionPtr in mActions)
    {
        if (tempActionPtr->GetValueString() == "ERROR NEVER INITIALIZED")//then this Action never had an assigned string so do nothing
        {
            //do nothing
        }
        else
        {//then check the substring for a match
                newList->Add_Action(tempActionPtr);
        }
    }
    return newList;
}


/**
* Gets actions which have had a WsfRoute value set for them from this object's vector of UBFAction objects
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_by_type_WsfRoute()
{
    UBFActionList * newList = new UBFActionList();
```

```
    for each (UBFAction * tempActionPtr in mActions)
    {
        if (tempActionPtr->GetValueWsfRoutePtr() == nullptr)//then this Action doesnt have an assigned int so do nothing
        {
            //do nothing
        }
        else
        {
            newList->Add_Action(tempActionPtr);
        }
    }
    return newList;
}

/**
* Gets actions which have had an int value set for them from this object's vector of UBFAction objects
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_by_type_Int()
{
    UBFActionList * newList = new UBFActionList();
    for each (UBFAction * tempActionPtr in mActions)
    {
        if (tempActionPtr->GetValueInt() == -1)//then this Action doesnt have an assigned int so do nothing
        {
            //do nothing
        }
        else
        {
            newList->Add_Action(tempActionPtr);
        }
    }
    return newList;
}

/**
* Gets actions which have had a double value set for them from this object's vector of UBFAction objects
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_by_type_Double()
{
    UBFActionList * newList = new UBFActionList();
    for each (UBFAction * tempActionPtr in mActions)
```

```
    {
        if (tempActionPtr->GetValueDouble() == -1)//then this Action doesnt have a Double value assigned so do nothing
        {
            //do nothing
        }
        else
        {
            newList->Add_Action(tempActionPtr);
        }
    }
    return newList;
}

/**
* Gets actions which have had a WsfGeoPoint value set for them from this object's vector of UBFAction objects
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_by_type_WsfGeoPoint()
{
    UBFActionList * newList = new UBFActionList();
    for each (UBFAction * tempActionPtr in mActions)
    {
        if (tempActionPtr->GetValueWsfGeoPointPtr() == nullptr)//then this Action doesnt have a WsfGeoPoint object so do nothing
        {
            //do nothing
        }
        else
        {
            newList->Add_Action(tempActionPtr);
        }
    }
    return newList;
}

/**
* Gets actions which have had WsfTrack value set for them from this object's vector of UBFAction objects
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_by_type_WsfTrack()
{
    UBFActionList * newList = new UBFActionList();
    for each (UBFAction * tempActionPtr in mActions)
    {
```

```
        if (tempActionPtr->GetValueWsfTrackPtr() ==nullptr)//then this Action doesnt have a WsfTrack object so do nothing
        {
            //do nothing
        }
        else
        {
            newList->Add_Action(tempActionPtr);
        }
    }
    return newList;
}

/**
* Gets actions which have the highest priority and unique names from this object's vector of UBFAction objects
* @return A new UBFActionList object managed by AFSIM per UtScriptRef::cManage with only the appropriate UBFAction pointers
*/
UBFActionList * UBFActionList::Get_Actions_Unique_Top_Priorities()
{
    UBFActionList * newList = new UBFActionList();
    std::vector<std::string> usedNames;

    //iterates over all of the Actions of this object UBFActionsList and inserts them into a newList which will end with
    //actions of all unique names and the highest priority possible from the previous mAction vector
    for each (UBFAction * testAction in mActions)
    {
        bool found = false;
        for each (std::string testString in usedNames)
        {
            if (testString.compare(testAction->GetName())==0)//then they are the same
            {
                found = true;//indicate there is already an action of this name found
                break;//breaks out of inner for loop
            }
            else
            {
            }
        }

        if (found==false)//then this is a new Action name so add it
        {
            newList->Add_Action(testAction);//adds the action
            usedNames.push_back(testAction->GetName());//add the name to the
```

```
                used list
            continue;//continues to the next iterations of the outer for loop
        }
        else//then this action is not new so check if it is better or worse than the one already in the list
        {
            UBFAction * tempAction = newList->Next();
            while (tempAction!=nullptr)
            {
                if (tempAction->GetName().compare(testAction->GetName())==0)//then the identical one was found
                {
                    if (tempAction->GetPriority() >= testAction->GetPriority())//then newList's Action was better so do nothing
                    {

                    }
                    else //then the oldList has a better action so replace the newLists Action with it
                    {
                        newList->Erase_Action_By_Name(tempAction->GetName());
                        newList->Add_Action(testAction);
                    }
                }
                tempAction = newList->Next();
            }//end while (tempAction!=nullptr)
            newList->Next_Restart();//resets the newList's next iterator to the start position
        }//end if (found==false) else

    }//end for each (UBFAction * testAction in mActions)
    return newList;
}

/**
* This function is used to return the first UBFAction pointer from the vector of UBFActions owned by this object
* @return UBFACtion pointer the first pointer from the vector of UBFActions
*/
UBFAction * UBFActionList::First()
{
    if (mActions.size()>0)
    {
        return mActions[0];
    }
    return nullptr;
}

/**
```

```
* This function is used to return the last UBFAction pointer from the vector of UBFActions owned by this object
* @return UBFACtion pointer the last pointer from the vector of UBFActions
*/
UBFAction * UBFActionList::Last()
{
    if (mActions.size() > 0)
    {
        return mActions[mActions.size()-1];
    }
    return nullptr;
}


/**
* This function is used to return the next UBFAction pointer from the vector of UBFActions owned by this object.  NExt is
* determined by the iteratorForNExtMethods integer
* @return UBFACtion pointer the next pointer from the vector of UBFActions
*/
UBFAction * UBFActionList::Next()
{
    if (iteratorForNextMethods >= 0 && iteratorForNextMethods<( int)mActions.size())
    {
        return mActions[iteratorForNextMethods++];
    }
    return nullptr;
}

/**
* This function is used to return the UBFAction pointer from the vector of UBFActions owned by this object by index
* @Param Integer Index which is the indicie in the vector of UBFActions to look for.
* @return UBFACtion pointer the first pointer from the vector of UBFActions; NULL if outside of range
*/
UBFAction * UBFActionList::ByIndex(int i)
{
    if (i>=0&&i<(int)mActions.size())
    {
        return mActions[i];
    }
    std::cout << "Index out of bounds, returning null for UBFActionList.ByIndex(int " << i << ") call" << std::endl;
    return nullptr;
}

```

```cpp

/**
* This function restarts the integer used to iterate over the vector of UBFAction pointers by the Next() function.
*/
void UBFActionList::Next_Restart()
{
    iteratorForNextMethods = 0;
}

int UBFActionList::Size()
{
    return (int)mActions.size();
}


/**
* This function is used to remove the first UBFAction pointer from this object's vector of UBFAction pointers by name
* @Param String name to match and remove only the first instance of from this object's vector of UBFAction pointers
*/
bool UBFActionList::Erase_Action_By_Name(std::string oName)
{
    for (int i = 0; i < (int)mActions.size(); i++)
    {
        if (oName.compare(mActions[i]->GetName())==0)//then they are the same
        {
            mActions.erase(mActions.begin()+i);
            return true;//found one of that name so exit...I do not delete all because the mActions object is now changed so continueing a for loop on it is an uncomfortable procedure.
        }
    }
    return false;
}



```

```cpp
/**
* @title UBFArbiter.cpp
* @Author Jeff Choate
* @email Jeff.lee.choate@gmail.com or Jeffrey.choate@us.af.mil
* @description This file defines the UBFArbiter class for Capt Jeffrey Choate's Thesis work at the Air Force Institute of Technology, 2015-2017.
* @usage A UBFArbiter object is used to filter UBFActions given to it by an owning UBFBehavior and returning a list of filtered UBFActions
* for the UBFBehavior to either act upon or pass up to that UBFBehavior's parent UBFBehavior.
* @Modified The date last modified: 9 Oct 2016
* @Change_Log:
* 9 Oct 2016:
*/
#include "UBFArbiter.hpp"
#include "WsfScenario.hpp"
#include "UBFActionList.hpp"
#include "UBFActionList.hpp"
#include "script/WsfScriptContext.hpp"

#include <iostream>
/**
* This initialization is required for the singleton UBFArbiter used for scripts to access the current UBFArbiter.
*/
UBFArbiter* UBFArbiter::staticUBFArbiterPtr = nullptr;//needed to prevent external symbol errors on static member variable usage.

/**
*This Function returns the singleton instance pointer for the currently executing Arbiter. Not thread safe; work around for not knowing how to access parent object of executing script.
*@return UBFArbiter The current operating UBFArbiter
*/
UBFArbiter * UBFArbiter::getInstancePtr()
{
    return staticUBFArbiterPtr;
}

/**
* This Function sets the singleton instance pointer for the currently executing Arbiter. Not thread safe; work around for not knowing how to access parent object of executing script.
* @param ptr A UBFArbiter * object pointing to the currently executing Arbiter
*/
void UBFArbiter::setInstancePtr(UBFArbiter * ptr)
{
    staticUBFArbiterPtr = ptr;
}
```

```

/**
* This is the default constructor used when the scenario creates the very first instance of the Object.
*/
UBFArbiter::UBFArbiter(WsfScenario& aScenario) :WsfProcessor(aScenario),
mContextPtr(new WsfScriptContext(*aScenario.GetScriptContext()))
{
}

/**
* This is the Copy constructor used only by the Clone() method..
* @Param UBFArbiter & mArbiter is the Arbiter being copied.
*/
UBFArbiter::UBFArbiter(const UBFArbiter & oArbiter) :WsfProcessor(oArbiter),
mContextPtr(new WsfScriptContext(*(oArbiter.mContextPtr)))
{
    if (mContextPtr!=nullptr)
    {
        //manually copy the Execute script to the new Execute script pointer.
        //Unsure if able to simply copy the mArbiter.mExecuteScript or not. Done based on AFSIM examples.
        mExecuteScriptPtr = mContextPtr->FindScript("Execute");//possibly assigned null is acceptable if there wasn't an Execute Script
    }
}

/**
* This function adds a UBFAction * to this Arbiter's set/vector of Actions that will be returned.
* No current way to remove actions from this list.
* @Param UBFAction * A pointer to an action that will
*/
void UBFArbiter::Add_Action(UBFAction * newAction)
{
    if (newAction!=nullptr)
    {
        newActions.push_back(newAction);
    }
    else
    {
        std::cout << "WARNING Attempting to add null UBFAction failed." << std::endl;
    }
}


/**
```

```
* This function is required by AFSIM and processes all text input to construct versions of a UBFArbiter object.
* @Param UtInput &: Only handles Script Block for Execute...end_Execute while passing the other commands to WsfProcessor.ProcessInput().
* still considering adding a script variables ability for UBFArbiters or not
*/
bool UBFArbiter::ProcessInput(UtInput & aInput)
{
    bool myCommand = false;

    std::string command = aInput.GetCommand();

    if (command == "Execute")
    {
        mExecuteScriptPtr = mContextPtr->Compile("Execute", "void", aInput, "end_Execute");
        myCommand = true;
    }
    else if (command == "script_variables")
    {
        myCommand = mContextPtr->ProcessInput(aInput);
    }
    else
    {
        myCommand = WsfProcessor::ProcessInput(aInput);
    }
    return myCommand;
}

/**
* This function sets the script context for a UBFArbiter.  This is necessary because the
initialize method is never explicitly called for a UBFArbiter.
* @Param WsfScriptContext * is the pointer to the script context you wish to set this UBFArbiter's script contex to.
*/
void UBFArbiter::SetContext(WsfScriptContext * newContextPtr)
{
    //mContextPtr = newContextPtr;//This method overwrites the old mContextPtr losing the script variables
    mContextPtr->Initialize(newContextPtr->GetTIME_NOW(newContextPtr->GetContext()), newContextPtr->GetPLATFORM(newContextPtr->GetContext()), this);
    mContextPtr->SetParent(newContextPtr);//does this overwrite the current UBFBehavior's Script variables though? initial tests say no but shouldn't it?
}


```

```
/**
* This function is used to initialize the UBFArbiter's list of input Actions and calls the Execute script for this UBFArbiter.
* @Param vector of UBFAction pointers.
* @returns a vector of Action* pointers.
*/
std::vector<UBFAction*> UBFArbiter::Process(std::vector<UBFAction*> inputActions)
{
    UBFArbiter::setInstancePtr(this);//allow Arbiter Scripts to execute knowing who the correct arbiter object is
    newActions.clear();
    mActions.clear();//clear all of the Action pointers from the last time this UBFArbiter was called.
    Next_Restart();

    //Assign the list of action objects from the parent behavior and its children behaviors to a local structure
    //accessible from the singleton of Arbiters
    for each (UBFAction* var in inputActions)
    {
        mActions.push_back(var);
    }
    double retVal = 0.0;
    if (mExecuteScriptPtr != 0)
    {
        UtScriptData scriptRetVal(retVal);
        UtScriptDataList scriptArgs;
        mContextPtr->ExecuteScript(mExecuteScriptPtr, scriptRetVal, scriptArgs);
    }
    UBFArbiter::setInstancePtr(nullptr);
    if (mExecuteScriptPtr != 0) {//if the arbiter's script is not null then return the arbiters desired actions
        return newActions;
    }
    else
    {
        return mActions;//return the given actions if the arbiter script is null
    }
}

/**
* This is the destructor for UBFArbiter objects.  The only pointers created in this object are WsfScriptObjects.
*/
UBFArbiter::~UBFArbiter()
{
```

```
    //delete mContextPtr;//Not deleted because at one point this may have been overwritten with a parent UBFBehavior mContextPtr
    //...how/when should i delete the original mContext pointer created by this object..currently i just lose track of it after SetContext() is called
    //Do i need to delete the mExecuteScriptPtr or is that managed by AFSIM?
}

```

```cpp
/**
* @title UBFBehavior.cpp
* @Author Jeff Choate
* @email Jeff.lee.choate@gmail.com or Jeffrey.choate@us.af.mil
* @description This file defines the UBFBehavior class for Capt Jeffrey Choate's Thesis work at the Air Force Institute of Technology, 2015-2017.
* @usage A UBFBehavior object is used to store the definitions for a UBFBehavior.  Stores an associated UBFArbiter, script pointers for Pre_Condition,
* Execute, Map_To_Actions, and a list of children UBFBehaviors as well as other features for that UBFBehavior.
* @Modified The date last modified: 9 Oct 2016
* @Change_Log:
* 23 Jan 2017: REQUIRES MOD TO previous script files.  Actio.create() was changed here.
* 9 Oct 2016: Added comments, deleted GetUniqueID() usage since need was unknown, deleted mArbiterAssigned assignment from Copy Constructor since
* the actual UBFArbiter pointer wasnt being copied.  Re-ordered the Initialize() to have the Processor::Initialize() first incase mContext::Initialize()
* required parameters set by that when it operates.
*/
#include "UBFBehavior.hpp"
#include "UBFAction.hpp"
#include "processor\WsfProcessorTypes.hpp"
#include "UBFArbiter.hpp"
#include "WsfScenario.hpp"
#include <iostream>
#include "InputTree.hpp"
#include "UtInputBlock.hpp"
#include "script/WsfScriptContext.hpp"
#include "WsfPlatform.hpp"
#include "UBFActionList.hpp"
#include "WsfSimulation.hpp"
#include "WsfApplication.hpp"
#include <time.h>

/**
* This initialization is required for the singleton UBFBehavior used for scripts to access the current UBFBehavior.
*/
UBFBehavior* UBFBehavior::staticUBFBehaviorPtr = nullptr;//needed to prevent external symbol errors on static member variable usage.

/**
*This Function returns the singleton instance pointer for the currently executing UBFBehavior. Not thread safe; work around for not knowing how to access parent object of executing script.
*@return UBFBeahvior The current operating UBFBehavior
```

```cpp
*/
UBFBehavior * UBFBehavior::getInstancePtr()
{
    return staticUBFBehaviorPtr;
}

/**
* This Function sets the singleton instance pointer for the currently executing Behavior. Not thread safe; work around for not knowing how to access parent object of executing script.
* @param ptr A UBFBehavior * object pointing to the currently executing Behavior
*/
void UBFBehavior::setInstancePtr(UBFBehavior * ptr)
{
    staticUBFBehaviorPtr = ptr;
}

/**
* This is the default constructor used when the scenario creates the very first instance of the Object.
*/
UBFBehavior::UBFBehavior(WsfScenario & aScenario) :WsfProcessor(aScenario),
mContextPtr(new WsfScriptContext(*aScenario.GetScriptContext()))
//mScenario(&aScenario)//checking if this is necessary since WsfProcessor holds a Scenario ptr
{
    WsfObject::SetType(WsfStringId("undefined"));
    WsfObject::SetName(WsfStringId("undefined"));
}

/**
* This is the Copy constructor used only by the Clone() method.
* @Param UBFBehavior & mUBFBehavior is the UBFBehavior being copied.
*/
UBFBehavior::UBFBehavior(const UBFBehavior & mUBFBehavior) : WsfProcessor(mUBFBehavior),
mArbiterName(mUBFBehavior.mArbiterName),//Copies string name of Arbiter associated..Not pointer of Arbiter because at Initilize this object will get it's own unique clone of a UBFArbiter by that name
mProcessInputChildren(mUBFBehavior.mProcessInputChildren),//Copies the tree of string names of children behavirs..Not pointers because at Initialize this object will get it's own tree of unique cloned UBFBehaviors by those names
mContextPtr(new WsfScriptContext(*(mUBFBehavior.mContextPtr)))//passes scenario context to clones
//mScenario(mUBFBehavior.mScenario)//checking if this is necessary
{
    if (mContextPtr!=nullptr)
```

```cpp
    {
        //ensure all script pointers are copied as well.
        mExecuteScriptPtr = mContextPtr->FindScript( "Execute");
        mMapToActionScriptPtr = mContextPtr->FindScript( "Map_To_Action");
        mPreConditionScriptPtr = mContextPtr->FindScript( "Pre_Condition");
    }
    debug_time = mUBFBehavior.debug_time;
    executeFrequency = mUBFBehavior.executeFrequency;
}

/**
* This function is used to construct the UBFBehavior tree structure
* of children UBFBehavior pointers, find/assign
* the UBFArbiter pointer being used, and initialize the Script
* Context to the correct PLATFORM for
* itself and it's children objects (UBFBehaviors/UBFArbiter).
* This should only be called by AFSIM code and is only called if this
* exact object is a child of a Platform.
* Further work may be done to redistribute work being done here to the
* ProcessInput method if developer is better able to use
* the FromInput methods as I was unable.
* @Param aSimTime is a double with a value provided by AFSIM for the current simulation time
* @return boolean with the status of successful initialization or lack thereof.
*/
bool UBFBehavior::Initialize(double aSimTime)
{

    bool myCommand = false;//success or failure of initializations
    myCommand |= WsfProcessor::Initialize( aSimTime);

    //TO DO: I need help with properly setting this context because I should be able to access WSFDraw() and PLATFORM.goto
    //however, these functions being used cause my simulation to hang and fail to complete.
    mContextPtr->SetParent(&GetSimulation()->GetScriptContext());//This allows use of some global functions in script (TIME_NOW) but not others (PLATFORM.goto... WsfDraw())
    myCommand&= mContextPtr->Initialize(aSimTime, GetPlatform(), this);// Allows scripts access to correct PLATFORM object.
    myCommand &= BuildOwnBehaviorTree(mContextPtr, 0); //Constructs tree of children UBFBehaviors
    myCommand &= AssignMyArbiter(mContextPtr); //Conceptually a UBFArbiter doesnt need to know it's context because it should only filter based on the list of UBFActions it is given. But this allows an Arbiter to make decisions based on it's parent PLATFORM
    return myCommand;
    GetScenario();
```

```

}


///Destructor...I still need to put more thought into this
UBFBehavior::~UBFBehavior()
{
     //i currently have terrible memory management and memory leaks...this i realize
     //delete mContextPtr;//Not deleted because at one point this may have been overwritten with a parent UBFBehavior mContextPtr
     //...how/when should i delete the original mContext pointer created by this object..currently i just lose track of it after SetContext() is called

    //Also need to think of a way to properly delete the tree of InputTree pointers because they are created by this class, however, the same
    //pointers may be used by any cloned versions of this behavior and only the parent should delete...consider a flag in each
    //constructor to denote if the object is a original or not as well as counters to simulate smart pointers so each object may
    //delete an object

    delete Arbiter;//Delete child Arbiter because it was clone()'d specifically for this objects usage and is never copied to children

    //Do I need to delete the mExecuteScriptPtr, mPreConditionScriptPtr, or mMapToActionsScriptPtr or are they managed by AFSIM?
    //If I do then only the root UBFBehavior should delete as clones may also be using the same pointers

    //Do not delete the staticUBFBehaviorPtr because that simply points to UBFBehavior objects managed by other UBFBehaviors or AFSIM

}

/**
* This function is required by AFSIM and processes all text input to construct versions of a UBFBehavior object.  UBFBehavior objects
* are then stored in an AFSIM factory which may be used to retrieve those named UBFBehaviors and clone them for usage in Children/tree structures.
* @Param UtInput &:
* Input Command Handled  ||  Description of how it handles it
* Map_To_Action          ||  Assigns/Compiles a UtScript Block which an Analyst may use to turn this UBFBehavior's UBFActions into actual outputs
* Pre_Condition          ||  Assigns/Compiles a UtScript Block which an Analyst may use as a quick check of this UBFBehavior executing or not; default is True
* Execute                ||  Assigns/Compiles a UtScript Block which an
```

```
    Analyst may use to generate custom UBFAction's to pass to parent or Map_To_Action script
 * Arbiter                 ||  Stores the name of an Arbiter to assign to the UBFBehavior at the Initialize stage
 * Frequency               ||  Stores a second value that determines max frequency a behavior may be called, only usable with  children,
 * Add_Post_Condition      ||  Adds a string in a list of Post conditions that a behavior may ADD
 * Remove_Post_Condition   ||  Adds a string in a list of Post conditions that a behavior may REMOVE
 * Initial_Condition       ||  Adds a string in a list of initial conditions a behavior is applicable towards
 * Required_Data           ||  Adds a string in a list of required data, sensors or generic data, required for a UBFBehavior
 * Action_Setting          ||  Adds a string in a list indicating motors effected
 * Goal_Acieved            ||  Assigns a string to a field in the behavior indicating the abstract goal it achieves
 * Children                ||  Stores/Handles adding UBFBehavior names to InputTree structure
 * ----Behavior            ||  Sub-Command to Children which indicates a Behavior's Name follows the command
 * script_variables        ||  Sends aInput to mContext.ProcessInput(...) to handle assigning Script variables; unsure how these are rememered when mContext is re-set for children.
 *                         ||  All other commands sent to WsfProcessor.ProcessInput(...).
*/
bool UBFBehavior::ProcessInput(UtInput & aInput)
{
    bool myCommand = false;

    std::string command = aInput.GetCommand();
    WsfVariable<WsfStringId> mChildName;

    std::string ArbiterName;
    if (command == "Map_To_Action")
    {
        mMapToActionScriptPtr = mContextPtr->Compile( "Map_To_Action", "void", aInput, "end_Map_To_Action");
        myCommand = true;
    }
    else if (command == "Add_Post_Condition")
    {
        std::string addedValue;
        aInput.ReadValue(addedValue);
        Add_Adder_Post_Condition(addedValue);
        myCommand = true;
    }
```

```
    else if (command == "Remove_Post_Condition")
    {
        std::string addedValue;
        aInput.ReadValue(addedValue);
        Add_Adder_Post_Condition(addedValue);
        myCommand = true;
    }
    else if (command == "Action_Setting")
    {
        std::string addedValue;
        aInput.ReadValue(addedValue);
        Add_Action_Setting(addedValue);
        myCommand = true;
    }
    else if (command == "Required_Data")
    {
        std::string addedValue;
        aInput.ReadValue(addedValue);
        Add_Required_Data(addedValue);
        myCommand = true;
    }
    else if (command == "Goal_Achieved")
    {
        std::string addedValue;
        aInput.ReadValue(addedValue);
        mGoalAchieved = addedValue;
        myCommand = true;
    }
    else if (command == "Initial_Condition")
    {
        std::string addedValue;
        aInput.ReadValue(addedValue);
        Add_Initial_Condition(addedValue);
        myCommand = true;
    }
    else if (command == "Debug_Time")
    {
        debug_time = true;
        myCommand = true;
        std::cout << "\nDEBUGING TIME " << std::endl;
    }
    else if (command == "Pre_Condition")
    {
        mPreConditionScriptPtr = mContextPtr->Compile("Pre_Condition", 
          "bool", aInput, "end_Pre_Condition");
        myCommand = true;
        //std::cout << "\nRead Pre_condition flag " << std::endl;
    }
    else if (command == "Execute")
```

```
    {
        //mExecuteScriptPtr = mContextPtr->CompileImplicitScript(aInput, "Execute", "void");
        mExecuteScriptPtr = mContextPtr->Compile("Execute", "void", aInput, "end_Execute");
        myCommand = true;
        std::cout << "\nRead >>>>>>>>>>>>>>>>>>>>>execute flag "<<std::endl;
    }
    else if (command == "Arbiter")
    {
        myCommand = true;
        mArbiterName.ReadValue(aInput);
    }
    else if (command == "Frequency")
    {
        myCommand = true;
        aInput.ReadValue(executeFrequency);
        std::cout << "FOUND FREQUENCY: " << executeFrequency << std::endl;
    }
    else if (command == "Children")
    {
        //reading input via readcommands and readvalues because UtInputBlock was difficult to use
        myCommand = true;
        InputTree * lastChild = nullptr;
        aInput.ReadCommand(command);//pops a command off the front of the aInput stream
        command = aInput.GetCommand();

        while (command != "end_Children")
        {
            if (command == "Behavior")
            {
                std::string behaviorName;
                aInput.ReadValue(behaviorName);
                if (behaviorName.length() > 0)
                {
                    lastChild = new InputTree(behaviorName);
                    mProcessInputChildren.push_back(lastChild);
                    myCommand = true;
                }
                else
                {
                    std::cout << "Read in blank or empty behavior name.  This is not allowed." << std::endl;
                    return false;
                }
            }
            else if (command == "Children")
```

```
                {
                    if (lastChild != nullptr)
                    {
                        myCommand = StoreChildren(lastChild, aInput);
                    }
                    else
                    {
                        std::cout << "Error, to nest children lists they must be directly under a parent and not another children flag" << std::endl;
                        return false;
                    }
                }
                else
                {
                    std::string msg = "Command not recognized within children block: " + command;
                    throw UtInput::BadValue(aInput, msg);
                    return false;
                }
                aInput.ReadCommand(command);//pop command off front of aInput stream
            }//end while loop reading in behaviors
            return myCommand;
        }
        else if (command == "script_variables")
        {
            myCommand=mContextPtr->ProcessInput( aInput);
        }
        else
        {//handles update interval call
            myCommand = WsfProcessor::ProcessInput( aInput);
        }

        return myCommand;
    }


    /**
    * This function assigns and builds the tree of UBFBehavior pointers from the InputTree structure.  This should only be called from
    * the Initialize() function or from a parent UBFBehavior in the tree structure.  Handles building subtrees defined in this UBFBehavior
    * of it's children by calling the function AddChildrenToChildren()...Elaborating on this for clarity: Initially a child is searched for,
    * the child's BuildOwnBehaviorTree() is called to process the child's InputTree structure then the AddChildrenToChildren()
    * method is called to add InputTree pointers defined by this/parent UBFBehaviors.
```

```
* @Param WsfScriptContext is passed in to allow a child UBFBehavior object to know the PLATFORM/context it is operating in
* @Param int depthOfTree is passed in as a check to dis-allow circular references in tree structures that Analysts my define
* @return A bool indicating success (true) or failure (false)
*/
bool UBFBehavior::BuildOwnBehaviorTree(WsfScriptContext* newScriptContextPtr, int depthOfTree)
{
    depthOfTree++;
    if (depthOfTree > maxTreeDepth)
    {
        std::cout << "ERROR: MAX BEHAVIOR TREE DEPTH REACHED. CHECK BEHAVIOR TREES FOR LOOPS or increase max tree depth" << std::endl;
        return false;
    }
    bool myCommand = true;
    for each (InputTree* var in mProcessInputChildren)//traverse string tree to add pointers to UBFBehaviors
    {
        WsfVariable<WsfStringId> tempname = var->GetName();
        if (tempname.GetId() != 0)
        {
            if (!WsfProcessorTypes::Get(GetScenario()).Find(tempname.GetString())) {
                std::cout << "ERROR: couldn't find Behavior: " << tempname.GetString() << std::endl;
                return false;
            }
            else
            {
                //then the behavior was defined so add it to this nodes chidren list and
                //check if it has children who need to be added
                UBFBehavior* tempBehavior = static_cast<UBFBehavior*>(WsfProcessorTypes::Get(GetScenario()).Find(tempname.GetString())->Clone());
                mUBFChildren.push_back(tempBehavior);//added to children list
                tempBehavior->SetName(tempname.GetString());
                tempBehavior->AssignMyArbiter(newScriptContextPtr);//this is because init1 does not get called on children
                tempBehavior->SetContextPtr(newScriptContextPtr);//this overwrites the UBFBehaviors script variables
                //tempBehavior->SetParentContextPtr(newScriptContextPtr);
                bool testSuccess = tempBehavior->BuildOwnBehaviorTree(newScriptContextPtr, depthOfTree);//construct child's tree based on child's ProcessInput
                if (!testSuccess)
                {
```

```
                    std::cout << "Failed to add child to child " << std::endl;
                    return false;//propogate up the failure
                }
                //add additional children to that same child iff the Input string tree contains UBFBehaviors under a children tag for this child
                if (var->mChildren.size()>0)//children check
                {
                    bool success = AddChildrenToChildren(var, tempBehavior, newScriptContextPtr, depthOfTree);
                    if (!success)
                    {
                        std::cout << "Failed to add child to child " << std::endl;
                        return false;//propogate up the failure
                    }
                }
                myCommand = true;
            }
        }
        else {
            //std::cout << "No children found for this behaivior: " << this->GetNameId() << std::endl;
        }
    }//End foreach assigning children behaviors
    return myCommand;
}

/**
* This function is called by AFSIM iff this particular UBFBehavior Object is a direct component of a platform.
* This function calls the root UBFBehavior mExecute method and the map_to_actions method.  This is called at the
* interval set by the Analyst's use of update_interval command in script. Default interval is NEVER.
* @Param aSimTime a double with the current simulation time
*/
void UBFBehavior::Update(double aSimTime)
{
    try
    {
        mExecute(0, aSimTime);//Send default value 0 as starting depth of the tree being executed, redundant
                    //because I should have also implicitly checked for this duringt he tree construction
    }
    catch (const std::exception& e)
    {
```

```
        std::cout << "UN HANDLED EXCEPTION IN EXECUTE TREE" << e.what() << std::endl;

    }


    try
    {
        //Execute script to map the actions to actual outputs in the program
        //Called here because only the root node(behavior defined in the platform)
        //map to action method matters
        ExecuteMapToOutputs();//creates error if this isnt implemented in scriptFIX ME
    }
    catch (const std::exception& e)
    {
        std::cout << "UN HANDLED EXCEPTION IN MAP TO OUTPUTS" << e.what() <<std::endl;
    }


    //TODO: add code here to clean up all pointers created on this run (Action Objects)
    //since i should be done with them...think about this more


}

/**
* This function is used to call the Execute and Pre_condition scripts of a UBFBehavior object, children UBFBehavior
* object mExecute() functions as well as this UBFBehavior object's Arbiter's object's Execute script to filter this
* UBFBehavior's actions.  This should only be called by a UBFBehavior object's Update method or a parent's mExecute() function.
* @Param depth is an integer which allow checking for circular tree structures and prevents loops.
* @return a vector of UBFAction pointers which are conceptually the output of this UBFBehavior object
*/
std::vector<UBFAction*> UBFBehavior::mExecute(int depth, double aSimTime)
{
    mActions.clear();//clear last iterations actions from the set of actions
    Next_Restart();

    time_t starttime, preConditionTime, childrenTime, ExecuteTime, TotalTime;
    if(debug_time)
        starttime =time(0);
```

```
    //check the depth of the tree to prevent infinite recursion loops
    depth++;
    if (depth > maxTreeDepth)
    {
        std::cout << "Max behavior depth reached...Check your behaviors for circles..Do a series of behaviors call each other resulting in endless depth to the tree?" << std::endl;
        return mActions;
    }
    double retVal = 0.0;
    if (executeFrequency>0 && (timeLastExecuted+executeFrequency)> aSimTime)
    {
        //std::cout << "Failed Frequency Check"<<std::endl;
        return std::vector<UBFAction*>();//return nothing
    }

    timeLastExecuted = aSimTime;

    if (mPreConditionScriptPtr != 0)
    {
        //---Execute the script for the precondition in order to find it's return value and hence check the pre-condition
        UtScriptData scriptRetVal(retVal);
        UtScriptDataList scriptArgs;
        //this->staticUBFBehaviorPtr = this;//sets singleton/static variable used to find current behavior that is executing.  Allows action script methods to find correct behavior
        UBFBehavior::setInstancePtr(this);
        mContextPtr->ExecuteScript(mPreConditionScriptPtr, scriptRetVal, scriptArgs);
        UBFBehavior::setInstancePtr(nullptr);

        if (!scriptRetVal.GetBool())        //Now check the returned value from the precondition script
        {
            if (debug_time)
            {
                std::cout << GetName() << " pre_condition time: " << starttime - time(0);
            }
            return std::vector<UBFAction*>();//return nothing
        }
    }
    if (debug_time)
        preConditionTime = time(0);

    //Execute all children and add their actions to this behavior's action subset
    for each (UBFBehavior* varBehavior in mUBFChildren)
```

```
    {

        std::vector <UBFAction*> tempActions = varBehavior->mExecute(depth, aSimTime);

        //add all children actions to this behavior's vector of actions
        for each (UBFAction* varAction in tempActions)
        {
            mActions.push_back(varAction);
        }
    }
    if (debug_time)
        childrenTime = time(0);
    //Execute the execute Script from the analyst for current behavior (above executed children mExecutes)
    if (mExecuteScriptPtr != 0)
    {

        UtScriptData scriptRetVal(retVal);
        UtScriptDataList scriptArgs;
        //this->staticUBFBehaviorPtr = this;//sets singleton/static variable used to find current behavior that is executing.  Allows action script methods to find correct behavior
        UBFBehavior::setInstancePtr(this);
        mContextPtr->ExecuteScript(mExecuteScriptPtr, scriptRetVal, scriptArgs);
        UBFBehavior::setInstancePtr(nullptr);

        //std::cout << "executed script and found this return double " << scriptRetVal.GetDouble() << std::endl;
    }
    if (debug_time)
        ExecuteTime = time(0);
    if (Arbiter!=nullptr)
    {//then this Behavior has an Arbiter hence filter all of this behavior's Actions through it's Arbiter
        mActions = Arbiter->Process(mActions);//assigning vectors over vectors may be un-kosher as it forgets some actions?
                                                //should i delete the pointers inside the arbiter for the actions not sent forward?
    }
    else
    {
        //do nothing because this will simply pass up the behavior's and it's children's Actions
    }
    if (debug_time)
    {
        TotalTime =time(0);
```

```cpp
        std::cout << GetName() << " had times: " << std::endl;
        std::cout <<"Total "<< TotalTime-starttime <<" seconds"<< std::endl;
        std::cout << "Pre_Condition: " << preConditionTime- starttime << " seconds" << std::endl;
        std::cout << "Children: " << childrenTime - preConditionTime << " seconds" << std::endl;
        std::cout << "Execute Block: " << ExecuteTime-childrenTime << " seconds" << std::endl;
        std::cout << "Arbiter: " << TotalTime- ExecuteTime << " seconds" << std::endl;
    }
    return mActions;
}



/**
* This function returns the context pointer for the UBFBehavior object in question
* @return WsfSCriptContext *
*/
WsfScriptContext * UBFBehavior::GetContextPtr()
{
    return mContextPtr;
}

/**
* This function changes the UBFBehaviors Script Context pointer
*/
void UBFBehavior::SetContextPtr(WsfScriptContext * newContextPtr)
{
    mContextPtr->Initialize(newContextPtr->GetTIME_NOW(newContextPtr->GetContext()),newContextPtr->GetPLATFORM(newContextPtr->GetContext()),this);
    mContextPtr->SetParent( newContextPtr);//does this overwrite the current UBFBehavior's Script variables though? initial tests say no but shouldn't it?
}

/**
* This function updates the parent pointer of a UBFBehaviors Script Context
*/
void UBFBehavior::SetParentContextPtr(WsfScriptContext * newContextPtr)
{
    mContextPtr->SetParent(newContextPtr);
}

/**
* This function searchs for a UBFBehavior by name and removes it from the
```

```
current UBFBehavior's set of children.  Currently there is no way to remove children from children dynamically.
* @Param string new_Behavior_Name is the name of an Analyst defined behavior to be searched for.
* @return a Bool value indicating success(true)
*/
bool UBFBehavior::Remove_Behavior(std::string deleteName)
{
    std::cout << "<<<<<<ATTEMPTING TO REMOVE BEHAVIOR " << deleteName << std::endl;
    for (int i = 0; i < (int)mUBFChildren.size(); i++)
    {
        if (deleteName.compare(mUBFChildren[i]->GetName()) == 0)//then they are the same
        {
            mUBFChildren.erase(mUBFChildren.begin() + i);
            std::cout << "<<<<<<FOUND and removed " << deleteName << std::endl;

            return true;//found one of that name so exit...I do not delete all because the mActions object is now changed so continueing a for loop on it is an uncomfortable procedure.
        }
        std::cout << "<<<<<<COMAPRING " << deleteName<< " and " << mUBFChildren[i]->GetComponentName().GetString()<< std::endl;

    }
    return false;

}

/**
* This function adds post conditions which this behavior REMOVES from the environment.
* @Param  the string condition which is added to the list
*/
void UBFBehavior::Add_Remove_Post_Condition(std::string newCondition)
{
    mPost_Conditions_Remove.push_back(newCondition);
}

/**
* This function searches for a UBFBehavior and returns a ptr to it.
* @Param  Pointer to the UBFBehavior
*/
UBFBehavior * UBFBehavior::Find(std::string oBehaviorName)
{
    if (!WsfProcessorTypes::Get(GetScenario()).Find( oBehaviorName)) {
        std::cout << "ERROR: couldn't find Behavior: " << oBehaviorName <<
```

```
            std::endl;
        return false;
    }
    else
    {
        UBFBehavior* tempBehavior = static_cast<UBFBehavior*>
          (WsfProcessorTypes::Get(GetScenario()).Find(oBehaviorName)->Clone
          ());
        if (tempBehavior != nullptr)
        {
            tempBehavior->SetName(oBehaviorName);
            return tempBehavior;
        }
    }
    return nullptr;
}

/**
* This function adds post conditions which this behavior ADDS to the
  environment.
* @Param  the string condition which is added to the list
*/
void UBFBehavior::Add_Adder_Post_Condition(std::string newCondition)
{
    mPost_Conditions_Add.push_back(newCondition);
}

/**
* This function adds a string to the Required data structure.
* @Param  the string condition which is added to the list
*/
void UBFBehavior::Add_Required_Data(std::string newCondition)
{
    mRequiredData.push_back(newCondition);
}

/**
* This function adds a string to the Required data structure.
* @Param  the string condition which is added to the list
*/
void UBFBehavior::Add_Initial_Condition(std::string newCondition)
{
    mInitialConditions.push_back(newCondition);
}
/**
* This function adds a string to the list of effected motors list.
* @Param  the string condition which is added to the list
*/
void UBFBehavior::Add_Action_Setting(std::string newCondition)
```

```
{
    mActionSettings.push_back(newCondition);
}


/**
* This function adds a string to the list of effected motors list.
* @Param  the string condition which is added to the list
*/
void UBFBehavior::Set_GoalAchieved(std::string newGoal)
{
    mGoalAchieved = newGoal;
}

bool UBFBehavior::Adder_Post_Condition_Exists(std::string oCondition)
{
    for each (std::string tempString in mPost_Conditions_Add)
    {
        if (oCondition.compare(tempString) == 0)
        {
            return true;
        }
    }
    return false;
}

bool UBFBehavior::Remove_Post_Condition_Exists(std::string oCondition)
{
    for each (std::string tempString in mPost_Conditions_Remove)
    {
        if (oCondition.compare(tempString) == 0)
        {
            return true;
        }
    }
    return false;
}

bool UBFBehavior::Action_Setting_Exists(std::string oSetting)
{
    for each (std::string tempString in mActionSettings)
    {
        if (oSetting.compare(tempString) == 0)
        {
            return true;
        }
    }
    return false;
}
```

```

bool UBFBehavior::Required_Data_Exists(std::string oData)
{
    for each (std::string tempString in mRequiredData)
    {
        if (oData.compare(tempString) == 0)
        {
            return true;
        }
    }
    return false;
}

bool UBFBehavior::Initial_Condition_Exists(std::string oCondition)
{
    for each (std::string tempString in mRequiredData)
    {
        if (oCondition.compare(tempString) == 0)
        {
            return true;
        }
    }
    return false;
}

std::string UBFBehavior::Get_Adder_Post_Condition_byIndex(int index)
{
    if (index < mPost_Conditions_Add.size())
    {
        return mPost_Conditions_Add[index];
    }
    return "DNE";
}

std::string UBFBehavior::Get_Remove_Post_Condition_byIndex(int index)
{
    if (index < mPost_Conditions_Remove.size())
    {
        return mPost_Conditions_Remove[index];
    }
    return "DNE";
}

std::string UBFBehavior::Get_Action_Setting_byIndex(int index)
{
    if (index < mActionSettings.size())
    {
        return mActionSettings[index];
    }
```

```
    return "DNE";
}

std::string UBFBehavior::Get_Required_Data_byIndex(int index)
{
    if (index < mRequiredData.size())
    {
        return mRequiredData[index];
    }
    return "DNE";
}

std::string UBFBehavior::Get_Initial_Condition_byIndex(int index)
{
    if (index < mInitialConditions.size())
    {
        return mInitialConditions[index];
    }
    return "DNE";
}

std::string UBFBehavior::Get_GoalAchieved()
{
    return mGoalAchieved;
}

int UBFBehavior::Adder_Post_Condition_Size()
{
    return mPost_Conditions_Add.size();
}

int UBFBehavior::Remove_Post_Condition_Size()
{
    return mPost_Conditions_Remove.size();
}

int UBFBehavior::Action_Setting_Size()
{
    return mActionSettings.size();
}

int UBFBehavior::Required_Data_Size()
{
    return mRequiredData.size();
}

int UBFBehavior::Initial_Condition_Size()
{
    return mInitialConditions.size();
```

```cpp
}
/**
* This function searches for a UBFBehavior by name and add's it to the current UBFBehavior's set of children.
* Currently there is no way to add children to children dynamically.
* This function should only be called via AFSIM script not from internal UBFBehavior functions.
* @Param string new_Behavior_Name is the name of an Analyst defined behavior to be searched for.
* @return a Bool value indicating success(true)
*/
bool UBFBehavior::Add_Behavior(std::string new_Behavior_Name)
{

    if (!WsfProcessorTypes::Get(GetScenario()).Find( new_Behavior_Name)) {
        std::cout << "ERROR: couldn't find Behavior: " << new_Behavior_Name << std::endl;
        return false;
    }
    else
    {
        UBFBehavior* tempBehavior = static_cast<UBFBehavior*>(WsfProcessorTypes::Get(GetScenario()).Find( new_Behavior_Name)->Clone());
        if (tempBehavior!=nullptr)
        {
            UBFBehavior::getInstancePtr()->mUBFChildren.push_back(tempBehavior);
            tempBehavior->SetName(new_Behavior_Name);
            tempBehavior->AssignMyArbiter(UBFBehavior::getInstancePtr()->GetContextPtr());
            tempBehavior->SetContextPtr(UBFBehavior::getInstancePtr()->GetContextPtr());
            if (tempBehavior->BuildOwnBehaviorTree(UBFBehavior::getInstancePtr()->GetContextPtr(), 0))
            {
                return true;
            }
            return false;
        }
        return false;
    }
}

bool UBFBehavior::Add_Behavior(UBFBehavior * newChild)
{
    mUBFChildren.push_back(newChild);
    newChild->AssignMyArbiter(GetContextPtr());
    newChild->SetContextPtr(GetContextPtr());
```

```
    if (newChild->BuildOwnBehaviorTree(GetContextPtr(), 0))
    {
        return true;
    }
    return false;
}

/**
* This function executes the Map_To_Actions script and properly sets the UBFBehavior singleton.
* Essentially Map_To_Actions script is where the script context has to be the most correct because
* this is where an Analyst will actuate on Platforms, send messages, send commands to sub ordinates, etc.
*/
void UBFBehavior::ExecuteMapToOutputs()
{
    //Actually execute the Script from an analyst to map actions from other behaviors to platform actions
    double retVal = 0.0;
    if (mMapToActionScriptPtr != nullptr)
    {
        UtScriptData scriptRetVal(retVal);
        UtScriptDataList scriptArgs;
        UBFBehavior::setInstancePtr(this);
        mContextPtr->ExecuteScript(mMapToActionScriptPtr, scriptRetVal, scriptArgs);
        UBFBehavior::setInstancePtr(nullptr);
    }
    //else call default map to action method...currently this is a warning telling the user their UBF improperly constructed
    else
    {
        std::cout << "WARNING -- MAP TO ACTION METHOD NOT DEFINED IN ROOT BEHAVIOR" << std::endl;
    }
}

/**
* This function finds an Arbiter in the AFSIM Processor factory and assigns it to this UBFBehavior's
* Arbiter pointer.  This method also shares/sets the Arbiter pointer's script context intending on allowing it
* knowledge of the calling platform.  The assigned Arbiter is based on the string assigned in the ProcessInput() stage.
* @Param WsfScriptContext * is a pointer to the context of the calling/parent UBFBehavior allowing access to the calling platform.
* @return A bool indicating the success of this method in finding and assigning the Arbiter.
```

```
*/
bool UBFBehavior::AssignMyArbiter(WsfScriptContext* newScriptContextPtr)
{
    if (!mArbiterAssigned)
    {
        mArbiterAssigned = true;
    }
    else
    {
        std::cout << "AssignMyArbiter called twice but why!" << std::endl;
        return false;
    }

    bool myCommand = false;
    //find and assign arbiter
    if (mArbiterName.GetId() != 0)
    {
        //std::cout << "arbitername.getid() is not null" << std::endl;
        if (!WsfProcessorTypes::Get(GetScenario()).Find(mArbiterName.GetString()))
        {
            std::cout << "couldn't find " << mArbiterName.GetString() << std::endl;
            return false;
        }
        Arbiter = static_cast<UBFArbiter*>(WsfProcessorTypes::Get(GetScenario()).Find(mArbiterName.GetString())->Clone());
        Arbiter->SetContext(newScriptContextPtr);
        myCommand = true;
    }
    else {
        //assign a default arbiter
        myCommand = true;
    }
    //finished assigning arbiter
    return myCommand;
}

/**
* This function is used as a sub-ordinate of BuildOwnBehaviortrees().  The purpose of this function is to add UBFBehavior
* children defined by a parent UBFBehavior to the child UBFBehavior.
* @Param InputTree * parent is the input tree of the parent which holds the names of UBFBehaviors to be added to this UBFBehavior
* @Param UBFBehavior * parentbehaviorObject the object which will be assigned children from this function
* @Param WsfScriptContext * holds a pointer to the parent's script context in order to let children access platform's
* @Param int depthofTree Is used to track the depth of a tree and prevent
```

```
loops being created by an ANalyst
* @return a bool with the success or failure of finding/assigning/cloning UBFBehaviors from the AFSIM processor factory.
*/
bool UBFBehavior::AddChildrenToChildren(InputTree * parent, UBFBehavior * parentBehaviorObject, WsfScriptContext * newScriptContextPtr, int depthOfTree)
{//should only be called when the  children list for var has items in it
    depthOfTree++;
    if (depthOfTree > maxTreeDepth)
    {
        std::cout << "ERROR: MAX BEHAVIOR TREE DEPTH REACHED. CHECK BEHAVIOR TREES FOR LOOPS or increase max tree depth" << std::endl;
        return false;
    }
    bool myCommand = true;
    for each (InputTree* var in parent->mChildren)
    {
        WsfVariable<WsfStringId> tempname = var->GetName();
        if (tempname.GetId() != 0)
        {
            if (!WsfProcessorTypes::Get(GetScenario()).Find(tempname.GetString())) {
                std::cout << "ERROR: couldn't find Behavior: " << tempname.GetString() << std::endl;
                return false;
            }
            else {//then the behavior was defined so add it to this nodes children list and check if it has children whih need to be added
                UBFBehavior* tempBehavior = static_cast<UBFBehavior*>(WsfProcessorTypes::Get(GetScenario()).Find(tempname.GetString())->Clone());
                parentBehaviorObject->mUBFChildren.push_back(tempBehavior);//added to children list
                tempBehavior->SetContextPtr(newScriptContextPtr);//this overwriting tempBehaviors sript variables?
                //tempBehavior->SetParentContextPtr(newScriptContextPtr);
                bool testSuccess = tempBehavior->BuildOwnBehaviorTree(newScriptContextPtr, depthOfTree);//construct child's tree based on child's ProcessInput
                if (!testSuccess)
                {
                    //std::cout << "Failed to add child to child " << std::endl;
                    return false;//propogate up the failure
                }
                if (var->mChildren.size()>0)
                {
                    bool success = AddChildrenToChildren(var, tempBehavior,
```

```
                        newScriptContextPtr, depthOfTree);
                        if (!success)
                        {
                            //std::cout << "Failed to add child to child " << std::endl;
                            return false;//propogate up the failure
                        }
                    }
                }
            }
    }

    return myCommand;
}

/**
* This function is used to store strings of behavior names which will later be used to build the UBFBehavior tree of pointers.
* This first level is built by the initial calling method (processInput).
* @Param InputTree * parentPtr a pointer to the parent InputTree object which will get behavior names from this method
* @Param UtInput * aInput the input stream from AFIT script
*/
bool UBFBehavior::StoreChildren(InputTree * parentPtr, UtInput & aInput)
{
    bool myCommand = true;
    InputTree * lastChild = nullptr;
    std::string command;
    aInput.ReadCommand(command);
    while (command != "end_Children")
    {
        if (command == "Behavior")
        {
            std::string behaviorName;
            aInput.ReadValue(behaviorName);
            if (behaviorName.length() > 0)
            {
                std::cout << "Adding a child to a child" << std::endl;
                lastChild = new InputTree(behaviorName);
                parentPtr->mChildren.push_back(lastChild);
            }
            else
            {
                std::cout << "Read in blank or empty behavior name.  This is not allowed." << std::endl;
                return false;
            }
        }
        else if (command == "Children")
```

```
        {
            if (lastChild != nullptr)
            {
                myCommand = StoreChildren(lastChild, aInput);
            }
            else
            {
                std::cout << "Error, to nest children lists they must be directly under a parent and not another children flag" << std::endl;
                return false;
            }
        }
        else
        {
            std::cout << "Command not recognized within children block: " << command << std::endl;
            return false;
        }
        aInput.ReadCommand(command);
    }
    return myCommand;
}


WsfSimulation* UBFBehavior::GetSimulation()
{
    WsfPlatform* platformPtr = OwningPlatform();
    return (platformPtr != 0) ? platformPtr->GetSimulation() : mContextPtr->GetSimulation();
}


UtScriptContext* UBFBehavior::GetScriptAccessibleContext()
{
    return &mContextPtr->GetContext();
}

//unsure this is necessary
const char* UBFBehavior::GetScriptClassName()
{
    return "UBFBehavior";
}



WsfPlatform* UBFBehavior::OwningPlatform()
{
    if (GetPlatform() != 0)
```

```
        return GetPlatform();
    else if (WsfScriptContext::GetPLATFORM(mContextPtr->GetContext()) != 0)
        return WsfScriptContext::GetPLATFORM(mContextPtr->GetContext());
    return 0;
}
```

Appendix B. Scripts Implemented

This appendix includes the various scripts that were used to define the

2.1 Platforms and Behaviors for Tutorial Scenario

```
include_once weapons/aam/medium_range_radar_missile.txt
include_once weapons/aam/simple_mrm_with_lc.txt

include_once
processors/quantum_agents/aiai/bt_behavior_planned_route.txt
include_once
processors/quantum_agents/aiai/bt_behavior_engage_weapon_task_target.txt
include_once
processors/quantum_agents/aiai/bt_behavior_pursue_target_route_finder.txt
include_once processors/quantum_agents/aiai/behavior_pursue_target_route_finder.txt
include_once processors/quantum_agents/behavior_controller_Fusion.txt

radar_signature SIG_RADAR_ONE_M_SQUARED
  constant 1.0 m^2
end_radar_signature

antenna_pattern ESM_ANTENNA
   constant_pattern
      peak_gain 3 db
end_antenna_pattern


platform_type  STRIKER  WSF_PLATFORM

  #indestructible
  #icon     F-22 / SU-27
  #side     blue / red

   category fighter
   radar_signature SIG_RADAR_ONE_M_SQUARED

   comm cmdr_net RED_DATALINK
      network_name <local:slave>
      internal_link data_mgr
      internal_link task_mgr
      internal_link perception
   end_comm

   mover WSF_AIR_MOVER
      roll_rate_limit               1 rad/sec
      default_linear_acceleration   1.0 g
      default_radial_acceleration   6.5 g
      default_climb_rate            400 fps
      maximum_climb_rate            400 fps
      maximum_speed                 600.0 knots
      minimum_speed                 150.0 knots
```

```
      maximum_altitude              50000 ft
      minimum_altitude              50 ft
      maximum_linear_acceleration   9 g
      at_end_of_path extrapolate
      turn_rate_limit               4.0 deg/sec
   end_mover

   processor data_mgr WSF_TRACK_PROCESSOR
      purge_interval              60 sec
      report_interval             1 sec
      fused_track_reporting       on
      raw_track_reporting         off
      report_to commander         via   cmdr_net
      circular_report_rejection   true
   end_processor

   weapon lc_mrm SIMPLE_MRM_WEAPON_LC
      quantity 10
   end_weapon

   weapon  mrm  MEDIUM_RANGE_RADAR_MISSILE
      quantity 10
   end_weapon

#   processor  task_mgr  WSF_QUANTUM_TASKER_PROCESSOR
#      script_debug_writes  on
#      update_interval 5 sec
#      behavior_tree
#         selector
#            behavior_node  bt_pursue_target_route_finder
#            behavior_node  bt_planned_route
#         end_selector
#         behavior_node  bt_engage_weapon_task_target
#      end_behavior_tree
#   end_processor
processor  task_mgr  WSF_QUANTUM_TASKER_PROCESSOR
      script_debug_writes  off
      update_interval 1 sec

            script int GetSalvoForThreat(WsfTrack track)
                 Map<string, int> ThreatTypeSalvo    = Map<string, int>();
      ThreatTypeSalvo["sam"]             = 2;
      ThreatTypeSalvo["ship"]            = 2;
      ThreatTypeSalvo["bomber"]          = 2;
      ThreatTypeSalvo["fighter"]         = 1;
      ThreatTypeSalvo["FIRE_CONTROL"]    = 1;
      ThreatTypeSalvo["primary_target"]  = 2;
```

```
       ThreatTypeSalvo["secondary_target"] = 2;
        int             DefaultAirSalvo      = 1;
       int             DefaultGndSalvo      = 1;
         #writeln_d("checking salvo size for category: ", category);
         #WsfPlatform plat = PLATFORM.FindPlatform( track.TargetIndex() );
         WsfPlatform plat = PLATFORM.FindPlatform( track.TargetName() );
         if (plat.IsValid())
         {
             foreach( string aCategory : int salvo in ThreatTypeSalvo )
             {
                 if( plat.CategoryMemberOf( aCategory ) )
                 {
                     writeln_d("salvo for type ", aCategory, " = ", salvo);
                     return salvo;
                 }
             }
         }
        #extern string GetTargetDomain(WsfTrack);
         string sTargetDomain = GetTargetDomain(track);
         if ( (sTargetDomain == "LAND") || (sTargetDomain == "SURFACE") )
         {
           return DefaultGndSalvo;
         }
         return DefaultAirSalvo;
     end_script

       aux_data
          int weaponIndex;
          string tempIDName;
          int tempIDInt;

       end_aux_data
       on_initialize
       SetAuxData("weaponIndex",-1);
       end_on_initialize
       execute at_interval_of 1 sec
#

       end_execute


    end_processor#end quantumtasker



    processor rootNode UBFBehavior
    #Debug_Time
```

```
update_interval 10 sec
script_variables
   //Example of variables that could be set for access in
   //this behavior's Execute or Map to action OR pre condition blocks.

end_script_variables
Map_To_Action
   #writeln("MTA");
   if(UBFBehavior.Get_Number_Of_Actions()==0){
         return;#no actions so do nothing
   }
    UBFActionList RouteList = UBFBehavior.Get_Actions_By_partial_Name("Route")
   if(RouteList.Get_Number_Of_Actions()>1)
   #then atleast one route lat long pair received
   {
      int routeSize  =-1;
      int routeStart =-1;

     Array<double> latitudes, longitudes, altitudes;
     UBFActionList routeLatitutes =
        RouteList.Get_Actions_By_Exact_Name("RouteLat");
     UBFActionList routeLongitudes =
        RouteList.Get_Actions_By_Exact_Name("RouteLong");

    latitudes = Array<double>();

     for(int ii=0;ii<routeLatitutes.Get_Number_Of_Actions();ii=ii+1)
     {#extract latitudes
        UBFAction tempAction = routeLatitutes.Get_Action_By_Index(ii);
        latitudes.Set(tempAction.Get_Priority(),tempAction.Get_Double());

     }
     longitudes = Array<double>();
     altitudes = Array<double>();

     for(int ii=0;ii<routeLongitudes.Get_Number_Of_Actions();ii=ii+1)
     {#extract latitudes and altitudes
        UBFAction tempAction = routeLongitudes.Get_Action_By_Index(ii);
        longitudes.Set(tempAction.Get_Priority(),tempAction.Get_Double());
       altitudes.Set(tempAction.Get_Priority(),tempAction.Get_Int());
     }

     UBFActionList routeStartList =
              RouteList.Get_Actions_By_Exact_Name("RouteStart");
     if(routeStartList!=null)
     {
        if(routeStartList.Get_Number_Of_Actions()>0)
```

```
   {#requires arbiters giving this to insure
   //there is only one set of route waypoints
      routeStart=routeStartList.Get_Action_By_Index(0).Get_Priority();
      routeSize = routeStartList.Get_Action_By_Index(0).Get_Double();
   }
}
if(routeSize<latitudes.Size())
{
   #then the analyst may not want to go to the end of the route actions g
}
else
{
   routeSize=latitudes.Size();#prevents reading past the end of the array
}
if(latitudes.Size()!=longitudes.Size())
{
   routeSize=0;
   writeln("route array mismatch check logic generating routes");
}
if(routeSize==-1)
{
   routeSize=2;
}

#set current position to the first route point
longitudes.Set(0, PLATFORM.Longitude());
latitudes.Set(0, PLATFORM.Latitude());
altitudes.Set(0, PLATFORM.Altitude());

WsfRoute newRoute=WsfRoute();




for(int ii=0;ii<routeSize;ii=ii+1)
{
    newRoute.Append(WsfGeoPoint.Construct(latitudes.Get(ii),
                longitudes.Get(ii),altitudes.Get(ii)), 450.0);
}
if((newRoute.Size()>0)&&(newRoute.IsValid()))
{
   if(routeStart!=-1)
   {
      if(routeStart>=newRoute.Size())
      {
          PLATFORM.FollowRoute(newRoute);
      }
```

```
            else
            {
               PLATFORM.FollowRoute(newRoute, routeStart);
            }

         }
         else
         {
             PLATFORM.FollowRoute(newRoute);
            # writeln("follow route");
         }

      }

    }

    UBFActionList wpnList = UBFBehavior.Get_Actions_By_Exact_Name("Weapon");
    if(wpnList.Get_Number_Of_Actions()>0)
    {
       UBFAction wpnAction =wpnList.Get_Action_By_Index(0);
       if(wpnAction==null)
          return;
       int weaponIndex=(int)wpnAction.Get_Double();

       WsfWeapon wpn = PLATFORM.WeaponEntry(weaponIndex);
       WsfTrackId tempID=
             WsfTrackId.Construct(wpnAction.Get_String(), wpnAction.Get_Int());
       WsfLocalTrack targetTrack = PLATFORM.MasterTrackList().FindTrack(tempID)
       writeln("targetTrack ; "+targetTrack.TargetName());
      #wpn.Fire(targetTrack.Target().MakeTrack());
    if(wpn.Name()!="mrm1")
    {

       PLATFORM.Processor("task_mgr").SetAuxData("weaponIndex", weaponIndex);
       PLATFORM.Processor("task_mgr").SetAuxData("tempIDName", tempID.Name());
       PLATFORM.Processor("task_mgr").SetAuxData("tempIDInt",tempID.Number());
       wpn.Fire(targetTrack);
    }

  }
   # UBFBehavior.Add_Action(UBFAction.Create("21",1213,22));

 #writeln("actions: "+(string)UBFBehavior.Get_Number_Of_Actions());
 end_Map_To_Action

 Execute
    #Empty because this Behavior is being
```

```
        #used to show an example Map To Action block.
#    WsfRouteFinder mRouteFinder = WsfRouteFinder();  did this work before?

      end_Execute
      #Arbiter CopyAll#Default Arbiter passes all actions
      #up to Map to action block or parent behaviors

      Children #list of the children that this behavior has
         Behavior B_UBF_Engage_Task_With_Weapon
         Behavior B_UBF_SelectMovement
      end_Children
   end_processor



   processor perception WSF_PERCEPTION_PROCESSOR
      on
      script_debug_writes  off
      report_interval      5 sec
      reporting_self       true
      report_to            commander:peers via cmdr_net
      asset_perception     status_messages
   end_processor

   sensor geo_sensor WSF_GEOMETRIC_SENSOR
      on
      azimuth_field_of_view   -180.0 degrees  180.0 degrees
      elevation_field_of_view  -90.0 degrees   90.0 degrees
      minimum_range 0 m
     #maximum_range 277800 m     //about 150 nm
      maximum_range 175940 m     //about  95 nm
      frame_time    0.5 sec
      reports_location
      reports_velocity
      reports_iff
      track_quality 1.0
      internal_link data_mgr
      ignore_same_side
   end_sensor

end_platform_type
```

```
processor B_UBF_SelectMovement UBFBehavior
#This Behavior is meant to take multiple
#Behaviors recommendations of Routes and
# combine them into one set of UBFAction
#Recommendations for the parent UBFBehavior to implement.
#INPUT/OUTPUT:
#N/A-passes up all UBFActions given.  Children
#should only send up recommendations if others arent or
#this arbiter needs to change.

 #  Execute
      #This Behavior is used as a logical connector
      #of other Behaviors so it doesnt need an Execute block
 #  end_Execute

   Arbiter CopyAllActionsUp
   Children
      Behavior B_UBF_Planned_Route
      Behavior  B_UBF_PursueTarget
   end_Children
end_processor
```

```
processor B_UBF_PursueTarget  UBFBehavior
#This Behavior is meant to produce waypoints
#as Actions based on a target from a task
#EXPECTATIONS: parent platform has a
#QuantumTaskerProcessor with name "task_mgr"
#INPUT: N/A
#OUTPUT:
#       Name      || RouteLat/RouteLong
#      Priority   ||  waypoint's index in the route
#      double     || Lat or Long
#        int      || altitude only for routeLong

   script_variables
      //expected global externs
      #extern Array<WsfGeoPoint> gAvoidPoints;
      #extern Array<double>      gAvoidRadii;
     double          cDEFAULT_ALTITUDE = 9144; // ~30,000 feet
 #     WsfRouteFinder mRouteFinder = WsfRouteFinder();
      bool            mDebugDraw = true;
      WsfGeoPoint     mTargetPoint;
      string          aTarget;
      double          mTargetSpeed = 300;  //300 ms (~600 knots)
      bool            mForceRePath = true;

      WsfGeoPoint     mCurrentAvoidancePt = WsfGeoPoint();
      WsfRoute        mCurrentRoute       = WsfRoute();
      UBFAction actionTarget, actionTarget1  ;
   end_script_variables

   Execute
#      mRouteFinder.SetImpossibleRouteResponse("SHIFT");
#      mRouteFinder.SetMaxArcLength(1852*5);   //max of 5 mile long arcs
      WsfQuantumTaskerProcessor proc =
                  (WsfQuantumTaskerProcessor)PLATFORM.Processor("task_mgr");

      #-------Precondition portion----------
      if (!proc.IsA_TypeOf("WSF_QUANTUM_TASKER_PROCESSOR")&&proc!=null)
      {
         return;
      }

      WsfTaskList tasks =
            ((WsfQuantumTaskerProcessor)proc).TasksReceivedOfType("WEAPON");
      if (tasks.Count() <= 0)
      {
         return;#no tasks so do nothing
```

```
}

aTarget="";
double desiredAlt;
for (int i=0; i<tasks.Count(); i=i+1)
{
   WsfTask task = tasks.Entry(i);
   WsfLocalTrack aTrack =
         PLATFORM.MasterTrackList().FindTrack(task.LocalTrackId());


   if (aTrack.IsValid())
   {
      //check if the target platform is terminated
     # if (aTrack.Target()!=NULL) #Can not access atarget.Target()
     # {
#              ((WsfQuantumTaskerProcessor)proc).SetTaskComplete(task, "SUCCESSFUL"
#              continue;
             #if target is deleted then it should no longer be a task for this pla
   #     }
      mTargetPoint = aTrack.CurrentLocation();
     # writeln("Current target Name"+aTrack.TargetName());
      //set altitude

      desiredAlt = MATH.Max(PLATFORM.Altitude(),
               MATH.Max(cDEFAULT_ALTITUDE, mTargetPoint.Altitude()));
      mTargetPoint.Set(mTargetPoint.Latitude(),
                              mTargetPoint.Longitude(), desiredAlt);
      aTarget = aTrack.TargetName();
      break;
   }
}


if(aTarget=="")
{
   return;#no valid target so return
}

 // if we are more than 2 seconds away from our target
if (mForceRePath || PLATFORM.SlantRangeTo(mTargetPoint) > (3*mTargetSpeed))
{#only send an action up if it is further than 2 seconda way
   double linearAccel = 7.5 * Earth.ACCEL_OF_GRAVITY();

   actionTarget = UBFAction.Create("RouteLat", 1, 1,mTargetPoint);
   actionTarget.Set_String(aTarget);
   actionTarget.Set_Double(mTargetPoint.Latitude());
```

```
            actionTarget1 = UBFAction.Create("RouteLong", 1,1, mTargetPoint);
            actionTarget1.Set_String(aTarget);
            actionTarget1.Set_Double(mTargetPoint.Longitude());
            actionTarget1.Set_Int(desiredAlt);
            UBFBehavior.Add_Action(actionTarget);
            UBFBehavior.Add_Action(actionTarget1);
        }
    end_Execute

end_processor
```

```
processor B_UBF_Planned_Route UBFBehavior
#This Behavior is meant to produce waypoints as
#Actions based on the platform not having a route active
#EXPECTATIONS: parent platform has a
#QuantumTaskerProcessor with name "task_mgr"
#INPUT: N/A
#OUTPUT:
#        Name     || RouteLat/RouteLong
#       Priority  ||  waypoint's index in the route
#       double    || Lat/Long
#         int     || altitude only for routeLong
#----
#        Name     || RouteStart
#       Priority  || starting point for route index
#       double    || route.size()

   script_variables
      bool     mDrawRoute          = false;
      double   cDEFAULT_SPEED      = 450.0 * MATH.MPS_PER_NMPH();
      double   cDEFAULT_ACCEL      =   7.5 * Earth.ACCEL_OF_GRAVITY(); // 7.5 G  (m/s
   end_script_variables

   Execute
   WsfMover aMover = PLATFORM.Mover();

   if(aMover.IsValid())
   {
         if(aMover.IsExtrapolating())
         {#then all other routes have ended and the platform needs
         #a new one or it will extrapolate(fly straight)
             #           writeln(PLATFORM.Name(), " is Extrapolating");
            WsfGeoPoint pt = PLATFORM.Location();
            WsfRoute ro = aMover.DefaultRoute().Copy();
            #now we have a modifiable route
            if (!ro.IsValid())
               return;

            WsfGeoPoint close = ro.LocationAtDistance(ro.DistanceAlongRoute(pt));
            if (!close.IsValid()) {
               return;
            }
            close.SetAltitudeAGL(pt.Altitude());

            double d1 = ro.DistanceFromRoute(pt);
            double d2 = pt.GroundRangeTo(close);
            double d3 = -1;
```

```
Array<double> turnRad = aMover.PropertyDouble("turn_radius");
if (turnRad.Size() > 0) {
   d3 = 2*turnRad[0];
}
int i = 0;
for (; i < ro.Size(); i = i+1)#FIND THE CLOSEST POINT
#TO ME AND DIRECT ME TO IT
{
   WsfWaypoint wpt = ro.Waypoint(i);
   WsfGeoPoint rpt = wpt.Location();
   //check if we are close to an existing waypoint,
   #if so... break & fly at that one
   if (rpt.GroundRangeTo(close) < 926) {
      break;
   }
   double dist = ro.DistanceAlongRoute(rpt);
   if (dist > d1) {
      if (d2 > d3) {
         ro.Insert(i, WsfWaypoint.Create(close, wpt.Speed()));
      }
      break;
   }
}


if (i >= ro.Size()) {
   i = ro.Size() - 1;
}
//go at default speed; this gets overwritten if route
#waypoint has defined a speed
UBFBehavior.Add_Action(UBFAction.Create("Speed",1,1,cDEFAULT_SPEED));
UBFBehavior.Add_Action(UBFAction.Create("Accell",1,1,cDEFAULT_ACCEL));
UBFAction routeStartAction =
                     UBFAction.Create("RouteStart",i,1,ro.Size());
UBFBehavior.Add_Action(routeStartAction);

#Add all the points of the route
int index=0;
for (; index < ro.Size(); index = index+1)
{
   WsfWaypoint tempPoint=ro.Waypoint(index);
   UBFAction tempLatAction =
          UBFAction.Create("RouteLat",index,1,tempPoint.Latitude());
   UBFAction tempLongAction =
          UBFAction.Create("RouteLong",index,1,tempPoint.Longitude());
   tempLongAction.Set_Int(tempPoint.Altitude());
```

```
            UBFBehavior.Add_Action(tempLatAction);
            UBFBehavior.Add_Action(tempLongAction);
            }
        }
    }
    else
    {
      writeln("invalid mover on platform: "+PLATFORM.Name());
    }
    end_Execute
end_processor
```

```
processor B_UBF_GenerateTargetsFromTasks UBFBehavior
#This Behavior is meant to pass up target recommendations based
#on the tasks assigned to this platform
#Dependancy: parent platform has a QuantumTaskerProcessor
#with name "task_mgr"
#INPUT: all children input will be passed forward
#OUTPUT: UBFActions with
#        Name      || Target
#       Priority   || 2
#       int        || WsfTrackId.Number()
#       string     || WsfTrackId.Name()

   Frequency 11
   Execute

         //specify orientation limits for shooting
         //dont shoot if rolled more/less than this
         double  mMaxFiringRollAngle    =  10.0;
         //dont shoot if pitched more than this
         double  mMaxFiringPitchAngle   =  15.0;
         //dont shoot if pitched less than this
         double  mMinFiringPitchAngle   = -10.0;
         bool    mCoopEngageOne        = false;


      double pitch = PLATFORM.Pitch();
      WsfQuantumTaskerProcessor proc =
                  (WsfQuantumTaskerProcessor)PLATFORM.Processor("task_mgr");
      if(!proc.IsValid())
      {
           writeln_d("Invalid Processor on platform, no weapons will fire");
      }

      WsfTaskList tasks = proc.TasksReceivedOfType("WEAPON");
      if (MATH.Fabs(PLATFORM.Roll()) > mMaxFiringRollAngle ||
          pitch > mMaxFiringPitchAngle                     ||
          pitch < mMinFiringPitchAngle)
      {
        string msgStr = write_str("    ", PLATFORM.Name(),
                    " orientation too far off to fire!  (roll or pitch)");
        writeln_d(msgStr);
        return;#return nothing since you are turning too much to fire
      }

      if(tasks.Count()==0)
      {
```

```
         return;#No tasks so nothing for this behavior to attack.
      }

       foreach (WsfTask task in tasks)
      {
         WsfLocalTrack targetTrack =
                  PLATFORM.MasterTrackList().FindTrack(task.LocalTrackId());
         if (targetTrack.IsNull() || !targetTrack.IsValid())
         {
            writeln_d("target track not valid");
            continue;
         }


         #Copied from example code-I think this
         #checks if the target is also from a sensor or not
         #Hence, this behavior will not return
         #a target if the platform already sensed it
         if (mCoopEngageOne == false)
         {
            WsfLocalTrack targetLocalTrack = (WsfLocalTrack)targetTrack;
            if (targetLocalTrack.IsValid())
            {
               if(!targetLocalTrack.ContributorOf(PLATFORM) &&
                  !targetLocalTrack.IsPredefined())
               {
                  return;
               }
            }
         }


#                  string nameholder=task.LocalTrackId().Name();
#         int idholder = task.LocalTrackId().Number();
         writeln("Weapons Pending : " +
                     (string)PLATFORM.WeaponsPendingFor(task.LocalTrackId()));
         writeln("Weapons active : " +
                     (string)PLATFORM.WeaponsActiveFor(task.LocalTrackId()));
         writeln(task.LocalTrackId().ToString());
         if ((PLATFORM.WeaponsPendingFor(task.LocalTrackId()) +
                       PLATFORM.WeaponsActiveFor(task.LocalTrackId())) > 0)
         {
            writeln("already have weapons assigned for target track");
            continue;
         }
         #Now add action objects as this UBFBehaviors recommendations
         UBFAction a =
```

```
                 UBFAction.Create("Target", 2, 1,task.LocalTrackId().Name());
         a.Set_Int(task.LocalTrackId().Number());
         UBFBehavior.Add_Action(a);
      }#END-foreach (WsfTask task in tasks)

   end_Execute

   Arbiter UBF_A_CheckTrackQualityWeaponsPending
end_processor
```

```
processor B_UBF_AddValidWeaponsToTargets UBFBehavior
#This Behavior is meant to take multiple Behaviors recommendations of Targets and
#assign weapons to them only passing up the first valid combination found
#EXPECTATIONS: parent platform has a QuantumTaskerProcessor with name "task_mgr"
#INPUT:
#      Name      || Target
#      Priority  || n/a
#      int       || WsfTrackId.Number()
#      string    || WsfTrackId.Name()

#OUTPUT: UBFActions from the UBFArbiter, Execute block is empty
#       Name     || Weapon
#      Priority  || n/a
#      int       || WsfTrackId.Number()
#      string    || WsfTrackId.Name()
#      Double    || weapon index
   Execute

   end_Execute

   Arbiter UBF_A_AssignWeaponFromFirstTarget
   Children
      Behavior B_UBF_GenerateTargetsFromTasks
   end_Children
end_processor
```

2.2 Platforms and Behaviors for Tuning Scenario

```
# New file created by AFSIM IDE
include_once Platforms/Striker_Type_Emergence.txt

#Default Route for Blue aircraft that gets modified by each individual Plane
route cap_orbit
  label start
     offset  20   0 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
     offset  20   5 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
     offset  0    5 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
     offset  0    0 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
   goto start
end_route
platform Blu0 STRIKER_Emergence
  side blue
  icon F-18
  position 30:02n 81:35:32.42w
   altitude 27000 feet
  execute at_interval_of  10 sec
     WsfDraw f=WsfDraw();
     f.SetTextSize(20);
     f.SetColor(0,0,0);
     f.SetId(PLATFORM.Name().Strip("Blu"));
     f.Erase(PLATFORM.Name().Strip("Blu"));
     f.BeginText(PLATFORM.Name().Strip("Blu"));
     WsfGeoPoint newp= PLATFORM.Location();
     newp.SetAltitudeAGL(newp.Altitude()+50);
     f.Vertex(newp);     f.End();
  end_execute
end_platform

platform Blu1 STRIKER_Emergence
  side blue
  icon F-18
  position 30:02n 81:35:32.42w
     execute at_interval_of  10 sec
     WsfDraw f=WsfDraw();
     f.SetTextSize(20);
     f.SetColor(0,0,0);
     f.SetId(PLATFORM.Name().Strip("Blu"));
     f.Erase(PLATFORM.Name().Strip("Blu"));
     f.BeginText(PLATFORM.Name().Strip("Blu"));
     WsfGeoPoint newp= PLATFORM.Location();
```

```
      newp.SetAltitudeAGL(newp.Altitude()+50);
      f.Vertex(newp);
      f.End();
   end_execute
   route
       position 30:02n 81:35:32.42w
          altitude 35000 feet
      transform_route cap_orbit reference_heading  180.0 deg
   end_route
end_platform

platform Blu2 STRIKER_Emergence
   side blue
   icon F-18
   position 30:02n 81:35:32.42w
    altitude 27000 feet
   execute at_interval_of  10 sec
      WsfDraw f=WsfDraw();
      f.SetTextSize(20);
      f.SetColor(0,0,0);
      f.SetId(PLATFORM.Name().Strip("Blu"));
      f.Erase(PLATFORM.Name().Strip("Blu"));
      f.BeginText(PLATFORM.Name().Strip("Blu"));
      WsfGeoPoint newp= PLATFORM.Location();
      newp.SetAltitudeAGL(newp.Altitude()+50);
      f.Vertex(newp);     f.End();
   end_execute
end_platform

platform Blu3 STRIKER_Emergence
   side blue
   icon F-18
   position 30:02n 81:35:32.42w
    altitude 28000 feet
      execute at_interval_of  10 sec
      WsfDraw f=WsfDraw();
      f.SetTextSize(20);
      f.SetColor(0,0,0);
      f.SetId(PLATFORM.Name().Strip("Blu"));
      f.Erase(PLATFORM.Name().Strip("Blu"));
      f.BeginText(PLATFORM.Name().Strip("Blu"));
      WsfGeoPoint newp= PLATFORM.Location();
      newp.SetAltitudeAGL(newp.Altitude()+50);
      f.Vertex(newp);     f.End();
   end_execute
end_platform
platform Blu4 STRIKER_Emergence
```

```
  side blue
  icon F-18
  position 30:02n 81:35:32.42w
   altitude 29000 feet
     execute at_interval_of  10 sec
     WsfDraw f=WsfDraw();
     f.SetTextSize(20);
     f.SetColor(0,0,0);
     f.SetId(PLATFORM.Name().Strip("Blu"));
     f.Erase(PLATFORM.Name().Strip("Blu"));
     f.BeginText(PLATFORM.Name().Strip("Blu"));
     WsfGeoPoint newp= PLATFORM.Location();
     newp.SetAltitudeAGL(newp.Altitude()+50);
     f.Vertex(newp);     f.End();
  end_execute
end_platform
platform Blu5 STRIKER_Emergence
  side blue
  icon F-18
  position 30:02n 81:35:32.42w
   altitude 30000 feet
     execute at_interval_of  10 sec
     WsfDraw f=WsfDraw();
     f.SetTextSize(20);
     f.SetColor(0,0,0);
     f.SetId(PLATFORM.Name().Strip("Blu"));
     f.Erase(PLATFORM.Name().Strip("Blu"));
     f.BeginText(PLATFORM.Name().Strip("Blu"));
     WsfGeoPoint newp= PLATFORM.Location();
     newp.SetAltitudeAGL(newp.Altitude()+50);
     f.Vertex(newp);     f.End();
  end_execute
end_platform
platform Blu6 STRIKER_Emergence
  side blue
  icon F-18
  position 30:02n 81:35:32.42w
   altitude 31000 feet
     execute at_interval_of  10 sec
     WsfDraw f=WsfDraw();
     f.SetTextSize(20);
     f.SetColor(0,0,0);
     f.SetId(PLATFORM.Name().Strip("Blu"));
     f.Erase(PLATFORM.Name().Strip("Blu"));
     f.BeginText(PLATFORM.Name().Strip("Blu"));
     WsfGeoPoint newp= PLATFORM.Location();
     newp.SetAltitudeAGL(newp.Altitude()+50);
```

```
      f.Vertex(newp);     f.End();
   end_execute
end_platform
platform Blu7 STRIKER_Emergence
   side blue
   icon F-18
   position 30:02n 81:35:32.42w
    altitude 32000 feet
      execute at_interval_of  10 sec
      WsfDraw f=WsfDraw();
      f.SetTextSize(20);
      f.SetColor(0,0,0);
      f.SetId(PLATFORM.Name().Strip("Blu"));
      f.Erase(PLATFORM.Name().Strip("Blu"));
      f.BeginText(PLATFORM.Name().Strip("Blu"));
      WsfGeoPoint newp= PLATFORM.Location();
      newp.SetAltitudeAGL(newp.Altitude()+50);
      f.Vertex(newp);     f.End();
   end_execute
end_platform
platform Blu8 STRIKER_Emergence
   side blue
   icon F-18
   position 30:02n 81:35:32.42w
    altitude 33000 feet
      execute at_interval_of  10 sec
      WsfDraw f=WsfDraw();
      f.SetTextSize(20);
      f.SetColor(0,0,0);
      f.SetId(PLATFORM.Name().Strip("Blu"));
      f.Erase(PLATFORM.Name().Strip("Blu"));
      f.BeginText(PLATFORM.Name().Strip("Blu"));
      WsfGeoPoint newp= PLATFORM.Location();
      newp.SetAltitudeAGL(newp.Altitude()+50);
      f.Vertex(newp);     f.End();
   end_execute
end_platform
platform Blu9 STRIKER_Emergence
   side blue
   icon F-18
   position 30:02n 81:35:32.42w
    altitude 34000 feet
      execute at_interval_of  10 sec
      WsfDraw f=WsfDraw();
      f.SetTextSize(20);
      f.SetColor(0,0,0);
      f.SetId(PLATFORM.Name().Strip("Blu"));
```

```
    f.Erase(PLATFORM.Name().Strip("Blu"));
    f.BeginText(PLATFORM.Name().Strip("Blu"));
    WsfGeoPoint newp= PLATFORM.Location();
    newp.SetAltitudeAGL(newp.Altitude()+50);
    f.Vertex(newp);     f.End();
  end_execute
end_platform
```

```
radar_signature SIG_RADAR_ONE_M_SQUARED
  constant 1.0 m^2
end_radar_signature

platform_type  STRIKER_Emergence  WSF_PLATFORM
   category fighter
   radar_signature SIG_RADAR_ONE_M_SQUARED

   sensor geo_sensor WSF_GEOMETRIC_SENSOR
      on
      azimuth_field_of_view   -180.0 degrees  180.0 degrees
      elevation_field_of_view  -90.0 degrees   90.0 degrees
      minimum_range 0 m
     #maximum_range 277800 m     //about 150 nm
      maximum_range 175940 m     //about  95 nm
      frame_time    0.5 sec
      reports_location
      reports_velocity
      reports_iff
      track_quality 1.0
      internal_link data_mgr
      ignore_same_side
   end_sensor

   processor data_mgr WSF_TRACK_PROCESSOR
      purge_interval             60 sec
      report_interval            1 sec
      fused_track_reporting      on
      raw_track_reporting        off
      circular_report_rejection  true
   end_processor

   mover WSF_AIR_MOVER
      roll_rate_limit               1 rad/sec
      default_linear_acceleration   1.0 g
      default_radial_acceleration   6.5 g
      default_climb_rate            400 fps
      maximum_climb_rate            400 fps
      maximum_speed                 600.0 knots
      minimum_speed                 150.0 knots
      maximum_altitude              50000 ft
      minimum_altitude              50 ft
      maximum_linear_acceleration   9 g
      at_end_of_path extrapolate
      turn_rate_limit               4.0 deg/sec
   end_mover
```

```
    processor rootNode UBFBehavior
       update_interval 10 sec
       Map_To_Action
          if(UBFBehavior.Get_Number_Of_Actions()==0)
          {
             return;
          }
          UBFActionList RouteList = UBFBehavior.Get_Actions_By_partial_Name("Route");

          if(RouteList.Get_Number_Of_Actions()>0)
          {
             #construct array of points
             Array<WsfGeoPoint> points;
             points = Array<WsfGeoPoint>();
             for(int ii=0;ii<RouteList.Get_Number_Of_Actions();ii=ii+1)
             {
                  UBFAction tempAction = RouteList.Get_Action_By_Index(ii);
                  points.Set(tempAction.Get_Int(),tempAction.Get_Geo_Point());#  *.Se
             }
             #current position as start
             points.Set(0,PLATFORM.Location());
             WsfRoute newRoute =WsfRoute();
             for(int ii=0;ii<points.Size();ii=ii+1)
              {
                  newRoute.Append(points.Get(ii),450.0);
              }

              if((newRoute.Size()>0)&&(newRoute.IsValid()))
              {
                  PLATFORM.FollowRoute(newRoute);
              }
           }
       end_Map_To_Action
       Children
          Behavior EmergenceNormalize
       end_Children
    end_processor
 end_platform_type
```

```
processor Emergence UBFBehavior
   Arbiter Fusion_Vote_GeoPoint
   Children
      Behavior FlyAwayFromObstacle
      Behavior FlyAtPoint
   end_Children
end_processor
```

```
processor EmergenceNormalize UBFBehavior
   Execute
      if(UBFBehavior.Get_Number_Of_Actions()>0)
         {
            UBFAction tempAction=UBFBehavior.Get_Action_By_Index(0);
            WsfGeoPoint tempPt=tempAction.Get_Geo_Point();
            Vec3 toPt=Vec3.Construct(PLATFORM.Latitude()-tempPt.Latitude(),
                                     PLATFORM.Longitude()-tempPt.Longitude(),
                                      0 );
            toPt=toPt.Normal();
            WsfGeoPoint newPt=
                  WsfGeoPoint.Construct(PLATFORM.Latitude()-toPt.X(),
                                        PLATFORM.Longitude()-toPt.Y(),
                                        PLATFORM.Altitude());
           UBFAction newAction = UBFAction.Create(tempAction.Get_Name(),
                                                 tempAction.Get_Priority(),
                                                 tempAction.Get_Vote(),
                                                 newPt);
           newAction.Set_Int(1);
           if( UBFBehavior.Delete_Action_By_Name(tempAction.Get_Name()))
           {
              UBFBehavior.Add_Action(newAction);
           }
         }
   end_Execute
   Children
         Behavior  Emergence
   end_Children
end_processor
```

```
#This behavior is meant to show behavioral emergence
processor FlyAtPoint UBFBehavior
script_variables
   bool home=false;
end_script_variables
   Execute
      WsfGeoPoint goalPoint =WsfGeoPoint.Construct(30,-79,10668);
      Vec3 toGoal = Vec3.Construct(PLATFORM.Latitude()-goalPoint.Latitude(),
                                   PLATFORM.Longitude()-goalPoint.Longitude(),
                                   0);
      toGoal=toGoal.Normal();
      goalPoint = WsfGeoPoint.Construct(PLATFORM.Latitude()-toGoal.X(),
                                        PLATFORM.Longitude()-toGoal.Y(),
                                        0);
      UBFAction destinationAction = UBFAction.Create("Route",1,1, goalPoint);
      destinationAction.Set_Int(1);#this is the index of the point it should fly to
      UBFBehavior.Add_Action(destinationAction);
      if(WsfGeoPoint.Construct(30,-79,10668).GroundRangeTo(
                                             PLATFORM.Location())<25000 && !home)
      {
         writeln(PLATFORM.Name()+" Reached GOAL AT1: " + (string)TIME_NOW);
         home=true;
      }
   end_Execute
end_processor
```

```
#This behavior is meant to show behavioral emergence
processor FlyAwayFromObstacle UBFBehavior
   Execute
      WsfGeoPoint choice;
      double currentHeading= PLATFORM.Heading();
      double  choiceDist=-1;
      WsfGeoPoint obstacle = WsfGeoPoint.Construct(30,-80,1000);
      WsfGeoPoint platPoint= PLATFORM.Location();
      WsfGeoPoint currentDirectionPt=
         WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading)/7,
                               platPoint.Longitude()+MATH.Sin(currentHeading)/7,
                               platPoint.Altitude()),
   turnDirRight=
         WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading-90)/7,
                               platPoint.Longitude()+MATH.Sin(currentHeading-90)/7,
                               platPoint.Altitude()),
   turnDirLeft=
         WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading+90)/7,
                               platPoint.Longitude()+MATH.Sin(currentHeading+90)/7,
                               platPoint.Altitude());
      double obstacleTo1dist= obstacle.GroundRangeTo(currentDirectionPt),
            obstacleTo2dist= obstacle.GroundRangeTo(turnDirRight),
            obstacleTo3dist= obstacle.GroundRangeTo(turnDirLeft);
        if(obstacleTo1dist>=obstacleTo2dist && obstacleTo1dist >= obstacleTo3dist)
      {
         choiceDist=obstacleTo1dist;
         choice=currentDirectionPt;
      }
      else if(obstacleTo2dist>=obstacleTo1dist && obstacleTo2dist >= obstacleTo3dist
      {
         choiceDist=obstacleTo2dist;
         choice=turnDirRight;
      }
      else
      {
         choiceDist=obstacleTo3dist;
         choice=turnDirLeft;
      }
      int vote=(int)PLATFORM.Name().Strip("Blu");
      UBFAction destinationAction = UBFAction.Create("Route",1,vote, choice);
      destinationAction.Set_Int(1);#this is the index of the point it should fly to
      UBFBehavior.Add_Action(destinationAction);
   end_Execute
end_processor
```

```
include_once Platforms/Striker_Type_Behavior_Tree.txt
    .
#Default Route for Blue aircraft that gets modified by each individual Plane
route cap_orbit_BT
  label start
     offset  20   0 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
     offset  20   5 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
     offset  0    5 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
     offset  0    0 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
   goto start
end_route

platform BlueLead_BT STRIKER_Behavior_Tree
  side blue
  icon F-18
  position 30:02n 81:35:32.42w

  route
      position 30:02n 81:35:32.42w
         altitude 35000 feet
     transform_route cap_orbit_BT reference_heading  180.0 deg
  end_route
end_platform
```

```
radar_signature SIG_RADAR_ONE_M_SQUARED_BT
  constant 1.0 m^2
end_radar_signature

platform_type  STRIKER_Behavior_Tree  WSF_PLATFORM

   category fighter
   radar_signature SIG_RADAR_ONE_M_SQUARED_BT

   sensor geo_sensor WSF_GEOMETRIC_SENSOR
      on
      azimuth_field_of_view   -180.0 degrees  180.0 degrees
      elevation_field_of_view  -90.0 degrees   90.0 degrees
      minimum_range 0 m
     #maximum_range 277800 m     //about 150 nm
      maximum_range 175940 m     //about  95 nm
      frame_time    0.5 sec
      reports_location
      reports_velocity
      reports_iff
      track_quality 1.0
      internal_link data_mgr
      ignore_same_side
   end_sensor

   processor data_mgr WSF_TRACK_PROCESSOR
      purge_interval             60 sec
      report_interval            1 sec
      fused_track_reporting      on
      raw_track_reporting        off
      circular_report_rejection  true
   end_processor

   execute at_interval_of 10 sec
      WsfDraw draw = WsfDraw();
      draw.SetId(10);
      draw.Erase(10);
      draw.SetEllipseMode("line");
      draw.BeginCircle(0, 25000.0);
      WsfGeoPoint obstacle = WsfGeoPoint.Construct(30,-80,1000);
      draw.Vertex(obstacle);
      draw.End();
      draw.BeginCircle(0, 20000.0);
      WsfGeoPoint Goal = WsfGeoPoint.Construct(30,-79,1000);
      draw.Vertex(Goal);
      draw.SetTextSize(20);
      draw.SetColor(0,0,0);
```

```
      draw.BeginText("GOAL");
      draw.Vertex(Goal);
      draw.End();
   end_execute

   mover WSF_AIR_MOVER
      roll_rate_limit                  1 rad/sec
      default_linear_acceleration      1.0 g
      default_radial_acceleration      6.5 g
      default_climb_rate               400 fps
      maximum_climb_rate               400 fps
      maximum_speed                    600.0 knots
      minimum_speed                    150.0 knots
      maximum_altitude                 50000 ft
      minimum_altitude                 50 ft
      maximum_linear_acceleration      9 g
      at_end_of_path extrapolate
      turn_rate_limit                  4.0 deg/sec
   end_mover

   processor BT WSF_SCRIPT_PROCESSOR
   update_interval 10 sec
      behavior_tree
         selector
            behavior_node FlyAwayFromObstacleBT
            behavior_node FlyAtPointBT
         end_selector
      end_behavior_tree
   end_processor
end_platform_type
```

```
behavior FlyAtPointBT
   script_variables
      bool home=false;
   end_script_variables
   precondition
      return true;
   end_precondition
   execute
      WsfGeoPoint goalPoint =WsfGeoPoint.Construct(30,-79,10668);
      PLATFORM.GoToSpeed(450.0);
      PLATFORM.GoToLocation(goalPoint);
      if(goalPoint.GroundRangeTo(PLATFORM.Location())<25000 &&!home)
      {
         writeln(PLATFORM.Name()+" Reached GOAL AT: " + (string)TIME_NOW);
         home=true;
      }
   end_execute
end_behavior
```

```
behavior FlyAwayFromObstacleBT
   precondition
      WsfGeoPoint obstacle = WsfGeoPoint.Construct(30,-80,1000);
      if(obstacle.GroundRangeTo(PLATFORM.Location())<25000)
      {
         return true;
      }
      return false;
   end_precondition
   execute
       WsfGeoPoint choice;
      double currentHeading= PLATFORM.Heading();
      double  choiceDist=-1;
      WsfGeoPoint obstacle = WsfGeoPoint.Construct(30,-80,10668);
      WsfGeoPoint platPoint= PLATFORM.Location();
WsfGeoPoint currentDirectionPt=
         WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading)/7,
                               platPoint.Longitude()+MATH.Sin(currentHeading)/7,
                               platPoint.Altitude()),
turnDirRight=
          WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading-90)/7,
                                platPoint.Longitude()+MATH.Sin(currentHeading-90)/7,
                                platPoint.Altitude()),
turnDirLeft=
         WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading+90)/7,
                               platPoint.Longitude()+MATH.Sin(currentHeading+90)/7,
                               platPoint.Altitude());
      double obstacleTo1dist= obstacle.GroundRangeTo(currentDirectionPt),
            obstacleTo2dist= obstacle.GroundRangeTo(turnDirRight),
            obstacleTo3dist= obstacle.GroundRangeTo(turnDirLeft);
        if(obstacleTo1dist>=obstacleTo2dist && obstacleTo1dist >= obstacleTo3dist)
      {
         choiceDist=obstacleTo1dist;
         choice=currentDirectionPt;
      }
      else if(obstacleTo2dist>=obstacleTo1dist && obstacleTo2dist >= obstacleTo3dist
      {
         choiceDist=obstacleTo2dist;
         choice=turnDirRight;
      }
      else
      {
         choiceDist=obstacleTo3dist;
         choice=turnDirLeft;
      }
      PLATFORM.GoToSpeed(450.0);
```

```
         PLATFORM.GoToLocation(choice);
   end_execute
end_behavior
```

2.3 Platforms and Behaviors for Swarm Scenario

```
include_once Platforms/Striker_Type_Swarming.txt

#Default Route for Blue aircraft that gets modified by each individual Plane
route cap_orbit_Swarmer
  label start
     offset  20   0 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
     offset  20   5 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
     offset  0    5 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
     offset  0    0 km speed 450 kts altitude 35000 ft msl
        radial_acceleration 2 g
   goto start
end_route


platform BlueLead_Swarmer STRIKER_Swarmer
  side blue
  icon F-18
  position 30:02n 81:35:32.42w
  route
      position 30:02n 81:35:32.42w
         altitude 35000 feet
     transform_route cap_orbit_Swarmer reference_heading  180.0 deg
  end_route
end_platform

platform Blue2 STRIKER_Swarmer
  side blue
  icon F-18
  position 30:03n 81:35:32.42w
  altitude 35000 ft
end_platform

platform Blue3 STRIKER_Swarmer
  side blue
  icon F-18
  position 30:04n 81:35:32.42w
  altitude 35000 ft
end_platform

platform Blue4 STRIKER_Swarmer
  side blue
  icon F-18
  position 33:04n 79:35:32.42w
  altitude 35000 ft
```

```
end_platform

platform Blue5 STRIKER_Swarmer
  side blue
  icon F-18
  position 30:00n 79:00:00.42w
  altitude 35000 ft
end_platform

platform Blue6 STRIKER_Swarmer
  side blue
  icon F-18
  position 31:04n 29:35:32.42w
  altitude 35000 ft
end_platform
```

```
radar_signature SIG_RADAR_ONE_M_SQUARED
  constant 1.0 m^2
end_radar_signature

platform_type  STRIKER_Swarmer  WSF_PLATFORM

   category fighter
   radar_signature SIG_RADAR_ONE_M_SQUARED

   sensor geo_sensor WSF_GEOMETRIC_SENSOR
      on
      azimuth_field_of_view   -180.0 degrees  180.0 degrees
      elevation_field_of_view  -90.0 degrees   90.0 degrees
      minimum_range 0 m
     maximum_range 500800 m     //about 150 nm
      #maximum_range 175940 m     //about  95 nm
      frame_time    0.5 sec
      reports_location
      reports_velocity
      reports_iff
      track_quality 1.0
      internal_link data_mgr
   end_sensor

   processor data_mgr WSF_TRACK_PROCESSOR
      purge_interval            60 sec
      report_interval           1 sec
      fused_track_reporting     on
      raw_track_reporting       off
      circular_report_rejection  true
   end_processor



   mover WSF_AIR_MOVER
      roll_rate_limit                1 rad/sec
      default_linear_acceleration    1.0 g
      default_radial_acceleration    6.5 g
      default_climb_rate             400 fps
      maximum_climb_rate             400 fps
      maximum_speed                  600.0 knots
      minimum_speed                  150.0 knots
      maximum_altitude               50000 ft
      minimum_altitude               50 ft
      maximum_linear_acceleration    9 g
      at_end_of_path extrapolate
      turn_rate_limit                4.0 deg/sec
```

```
   end_mover

   processor rootNode UBFBehavior
      update_interval 10 sec
      Map_To_Action
         if(UBFBehavior.Get_Number_Of_Actions()==0)
         {
            return;
         }
         UBFActionList RouteList = UBFBehavior.Get_Actions_By_partial_Name("Route");

         if(RouteList.Get_Number_Of_Actions()>0)
         {
            #construct array of points
            Array<WsfGeoPoint> points;
            points = Array<WsfGeoPoint>();
            for(int ii=0;ii<RouteList.Get_Number_Of_Actions();ii=ii+1)
            {
                 UBFAction tempAction = RouteList.Get_Action_By_Index(ii);
                 points.Set(tempAction.Get_Int(),tempAction.Get_Geo_Point());#  *.Se
            }
            points.Set(0,PLATFORM.Location());#  current position as start,children

            WsfRoute newRoute =WsfRoute();
            for(int ii=0;ii<points.Size();ii=ii+1)
             {
                 newRoute.Append(points.Get(ii),450.0);
             }

             if((newRoute.Size()>0)&&(newRoute.IsValid()))
             {
                 PLATFORM.FollowRoute(newRoute);
             }
          }
      end_Map_To_Action
Children
   Behavior SwarmNormalize
end_Children
   end_processor
end_platform_type
```

```
processor SwarmVector UBFBehavior
   · Children
        Behavior Alignment
        Behavior Cohesion
        Behavior Seperation
     end_Children
     Arbiter Fusion_Vote_GeoPoint
end_processor
```

```
processor SwarmNormalize UBFBehavior
   Execute
      if(UBFBehavior.Get_Number_Of_Actions()>0)
         {
            UBFAction tempAction=UBFBehavior.Get_Action_By_Index(0);
            WsfGeoPoint tempPt=tempAction.Get_Geo_Point();
            Vec3 toPt=Vec3.Construct(PLATFORM.Latitude()-tempPt.Latitude(),
                                     PLATFORM.Longitude()-tempPt.Longitude(),
                                      0 );
            toPt=toPt.Normal();
            WsfGeoPoint newPt=WsfGeoPoint.Construct(PLATFORM.Latitude()-toPt.X(),
                                                    PLATFORM.Longitude()-toPt.Y(),
                                                    PLATFORM.Altitude());
           UBFAction newAction = UBFAction.Create(tempAction.Get_Name(),
                                                  tempAction.Get_Priority(),
                                                  tempAction.Get_Vote(),
                                                  newPt);
           newAction.Set_Int(tempAction.Get_Int());
           if( UBFBehavior.Delete_Action_By_Name(tempAction.Get_Name()))
           {
              UBFBehavior.Add_Action(newAction);
           }

         }
   end_Execute
   Children
      Behavior SwarmVector
   end_Children
end_processor
```

```
processor Alignment UBFBehavior
#This behavior passes up name and tracklist entry index if an enemy was detected
#Dependancy: parent platform is able to detect tracks
#INPUT: all children input will be passed forward
#OUTPUT: UBFActions with
#       Name      || Enemy
#      Priority   || 2
#      Track      || The enemy track

#inspired from https://gamedevelopment.tutsplus.com/tutorials/3-simple-rules-of-
#flocking-behaviors-alignment-cohesion-and-separation--gamedev-3444
   Execute
   Vec3 ff =Vec3.Construct(0,0,0);
   int neighborCount=0;
      for(int i=0;i<PLATFORM.MasterTrackList().Count();i=i+1)
      {
        WsfLocalTrack tempTrack= PLATFORM.MasterTrackList().Entry(i);
        ff=Vec3.Add(ff,tempTrack.VelocityNED());
        neighborCount=neighborCount+1;
      }
      if(neighborCount==0)
      {
         return;
      }
      ff.Set(ff.X()/neighborCount,ff.Y()/neighborCount,0);
      ff=ff.Normal();
     WsfGeoPoint ptRelativePlatformDirectionOfTrackVelocityVector =
                       WsfGeoPoint.Construct(
                       PLATFORM.Latitude()+ff.X(),
                       PLATFORM.Longitude()+ff.Y(),
                       0);
      UBFBehavior.Add_Action(UBFAction.Create("Route",1,1,
                     ptRelativePlatformDirectionOfTrackVelocityVector));
   end_Execute
end_processor
```

.

```
processor Cohesion UBFBehavior
#This behavior passes up name and tracklist entry index if an enemy was detected
#Dependancy: parent platform is able to detect tracks
#INPUT: all children input will be passed forward
#OUTPUT: UBFActions with
#       Name      || Enemy
#      Priority   || 2
#      Track      || The enemy track

#inspired from https://gamedevelopment.tutsplus.com/tutorials/3-simple-rules
#-of-flocking-behaviors-alignment-cohesion-and-separation--gamedev-3444
   Execute
   Vec3 ff =Vec3.Construct(0,0,0);
   int neighborCount=0;
      for(int i=0;i<PLATFORM.MasterTrackList().Count();i=i+1)
      {
        WsfLocalTrack tempTrack= PLATFORM.MasterTrackList().Entry(i);
        ff=Vec3.Add(ff,Vec3.Construct(
                       tempTrack.Latitude(),tempTrack.Longitude(),0));
        neighborCount=neighborCount+1;
      }
      if(neighborCount==0)
      {
         return;
      }
      ff.Set(ff.X()/neighborCount,ff.Y()/neighborCount,0);
      ff.Set(ff.X()-PLATFORM.Latitude(),ff.Y()-PLATFORM.Longitude(),0);
      ff=ff.Normal();
     WsfGeoPoint ptRelativePlatformDirectionOfTracksCenterOfMass =
                       WsfGeoPoint.Construct(
                       PLATFORM.Latitude()+ff.X(),
                       PLATFORM.Longitude()+ff.Y(),
                       0);
      UBFBehavior.Add_Action(UBFAction.Create("Route",1,1,
                        ptRelativePlatformDirectionOfTracksCenterOfMass));
   end_Execute
end_processor
```

```
processor Seperation UBFBehavior
#This behavior passes up name and tracklist entry index if an
# enemy was detected
#Dependancy: parent platform is able to detect tracks
#INPUT: all children input will be passed forward
#OUTPUT: UBFActions with
#        Name      || Enemy
#       Priority   || 2
#       Track      || The enemy track

#inspired from https://gamedevelopment.tutsplus.com/tutorials/
#3-simple-rules-of-flocking-behaviors-alignment-cohesion-and-
#separation--gamedev-3444
   Execute
   Vec3 ff =Vec3.Construct(0,0,0);
   int neighborCount=0;
      for(int i=0;i<PLATFORM.MasterTrackList().Count();i=i+1)
      {
        WsfLocalTrack tempTrack= PLATFORM.MasterTrackList().Entry(i);

        if(tempTrack.GroundRangeTo(PLATFORM)<50000)
        {
         writeln(PLATFORM.Name()+" "+tempTrack.TargetName()+" "+
                          (string)tempTrack.GroundRangeTo(PLATFORM));

        ff=Vec3.Add(ff,Vec3.Construct(tempTrack.Latitude(),
                                   tempTrack.Longitude(),0));
        ff=Vec3.Add(ff,Vec3.Construct(-1*PLATFORM.Latitude(),
                                   -1*PLATFORM.Longitude(),0));
        neighborCount=neighborCount+1;
        }
      }
      if(neighborCount==0)
      {
         return;
      }
      ff.Set(ff.X()/neighborCount,ff.Y()/neighborCount,0);
      ff.Set(-1*ff.X(),-1*ff.Y(),0);
      ff=ff.Normal();
     WsfGeoPoint ptAwayFromAllNeighorbors = WsfGeoPoint.Construct(
                        PLATFORM.Latitude()+ff.X(),
                        PLATFORM.Longitude()+ff.Y(),
                        0);
      UBFBehavior.Add_Action(UBFAction.Create("Route",1,2,
                                    ptAwayFromAllNeighorbors));
   end_Execute
end_processor
```

2.4 Behaviors for Combined Scenario

The platforms are omitted as well as various behaviors because they do not substantially change from the previous examples.

```
behavior FlyAtPointBT_Combined
   script variables
      bool home=false;
   end script variables
   precondition

      return true;
   end precondition
   execute
      WsfGeoPoint goalPoint =WsfGeoPoint.Construct(30,-79,10668);

      //Weighting system
      double separationWeight=2, cohesionWeight=1, alignmentWeight=1,goalWeight=2
      totalWeight=separationWeight+cohesionWeight+alignmentWeight+goalWeight;

      //Cohesion code
       Vec3 cohesionVec =Vec3.Construct(0,0,0);
       int cohesionNeighborCount=0;
      for(int i=0;i<PLATFORM.MasterTrackList().Count();i=i+1)
      {
        WsfLocalTrack tempTrack= PLATFORM.MasterTrackList().Entry(i);
        cohesionVec=Vec3.Add(cohesionVec,Vec3.Construct(
                       tempTrack.Latitude(),tempTrack.Longitude(),0));
        cohesionNeighborCount=cohesionNeighborCount+1;
      }
      if(cohesionNeighborCount==0)
      {
         cohesionWeight=0;
         cohesionNeighborCount=1;
      }
      cohesionVec.Set(cohesionVec.X()/cohesionNeighborCount,cohesionVec.Y()/cohes
      cohesionVec.Set(-cohesionVec.X()+PLATFORM.Latitude(),-cohesionVec.Y()+PLATF
      cohesionVec=cohesionVec.Normal();

      //Separation Code
      Vec3 separationVec =Vec3.Construct(0,0,0);
      int separationNeighborCount=0;
      for(int i=0;i<PLATFORM.MasterTrackList().Count();i=i+1)
      {
        WsfLocalTrack tempTrack= PLATFORM.MasterTrackList().Entry(i);

        if(tempTrack.GroundRangeTo(PLATFORM)<50000)
        {
      #   writeln(PLATFORM.Name()+" "+tempTrack.TargetName()+" "+
       #                  (string)tempTrack.GroundRangeTo(PLATFORM));
```

```
      separationVec=Vec3.Add(separationVec,Vec3.Construct(tempTrack.Latitude(),
                               tempTrack.Longitude(),0));

      separationNeighborCount=separationNeighborCount+1;
      }
    }
    if(separationNeighborCount==0)
    {
       separationWeight=0;
       separationNeighborCount=1;
    }
    separationVec.Set(separationVec.X()/separationNeighborCount,separationVec.Y
    separationVec.Set(separationVec.X()-PLATFORM.Latitude(),separationVec.Y()-P
    separationVec=separationVec.Normal();

    //Alignment Code
    Vec3 alignmentVec =Vec3.Construct(0,0,0);
    int neighborCount=0;
    for(int i=0;i<PLATFORM.MasterTrackList().Count();i=i+1)
    {
      WsfLocalTrack tempTrack= PLATFORM.MasterTrackList().Entry(i);
      alignmentVec=Vec3.Add(alignmentVec,tempTrack.VelocityNED());
      neighborCount=neighborCount+1;
    }
    if(neighborCount==0)
    {
       alignmentWeight=0;
       neighborCount=1;
    }
    alignmentVec.Set(-alignmentVec.X()/neighborCount,-alignmentVec.Y()/neighbor
    alignmentVec=alignmentVec.Normal();

    //Need custom code to merge them all now.  Inspired from the fusion vector
    Vec3 goalVec = Vec3.Construct(PLATFORM.Latitude()-goalPoint.Latitude(),PLAT
    goalVec=goalVec.Normal();
    double fusedLat=0, fusedLong=0, fusedAlt=0;
    fusedLat=alignmentVec.X()*alignmentWeight/totalWeight+cohesionVec.X()*cohes
             separationVec.X()*separationWeight/totalWeight + goalVec.X()*goalW
    fusedLong=alignmentVec.Y()*alignmentWeight/totalWeight+cohesionVec.Y()*cohe
             separationVec.Y()*separationWeight/totalWeight + goalVec.Y()*goalW
    fusedAlt=alignmentVec.Z()*alignmentWeight/totalWeight+cohesionVec.Z()*cohes
             separationVec.Z()*separationWeight/totalWeight + goalVec.Z()*goalW
  goalVec.SetX(fusedLat);
  goalVec.SetY(fusedLong);
  goalVec.SetZ(0);
```

```
    //Normalize the output to the platforms current position
      goalVec=goalVec.Normal();
   WsfGeoPoint newPt=WsfGeoPoint.Construct(PLATFORM.Latitude()-goalVec.X(),
                                            PLATFORM.Longitude()-goalVec.
                                            PLATFORM.Altitude());


      PLATFORM.GoToSpeed(450.0);
      WsfRoute newRoute =WsfRoute();
      newRoute.Append(newPt,PLATFORM.Altitude());
      PLATFORM.FollowRoute(newRoute);
   end_execute
end_behavior
```

.

```
behavior FlyAwayFromObstacleBT_Combined
   precondition
      WsfGeoPoint obstacle = WsfGeoPoint.Construct(30,-80,1000);
      if(obstacle.GroundRangeTo(PLATFORM.Location())<25000)
      {
         return true;
      }
      return false;
   end_precondition
   execute
       WsfGeoPoint choice;
      double currentHeading= PLATFORM.Heading();
      double  choiceDist=-1;
      WsfGeoPoint obstacle = WsfGeoPoint.Construct(30,-80,10668);
      WsfGeoPoint platPoint= PLATFORM.Location();
WsfGeoPoint currentDirectionPt=
         WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading)/7,
                               platPoint.Longitude()+MATH.Sin(currentHeading)/7,
                               platPoint.Altitude()),
turnDirRight=
          WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading-90)/
                                platPoint.Longitude()+MATH.Sin(currentHeading-90)
                                platPoint.Altitude()),
turnDirLeft=
         WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading+90)/7
                               platPoint.Longitude()+MATH.Sin(currentHeading+90)/
                               platPoint.Altitude());
      double obstacleTo1dist= obstacle.GroundRangeTo(currentDirectionPt),
            obstacleTo2dist= obstacle.GroundRangeTo(turnDirRight),
            obstacleTo3dist= obstacle.GroundRangeTo(turnDirLeft);
        if(obstacleTo1dist>=obstacleTo2dist && obstacleTo1dist >= obstacleTo3dist
      {
         choiceDist=obstacleTo1dist;
         choice=currentDirectionPt;
      }
      else if(obstacleTo2dist>=obstacleTo1dist && obstacleTo2dist >= obstacleTo3c
      {
         choiceDist=obstacleTo2dist;
         choice=turnDirRight;
      }
      else
      {
         choiceDist=obstacleTo3dist;
         choice=turnDirLeft;
```

```
            writeln("righting");

    }

    Vec3 temp = Vec3.Construct(choice.Latitude(), choice.Longitude(),0);
    temp=temp.Normal();
    Vec3 avoidVec = Vec3.Construct(PLATFORM.Latitude()-choice.Latitude(),PLATFO
    avoidVec=avoidVec.Normal();

    //weighting system
    double separationWeight=2, cohesionWeight=2, alignmentWeight=2,avoidWeight=
    totalWeight=separationWeight+cohesionWeight+alignmentWeight+avoidWeight;


    //Cohesion code
     Vec3 cohesionVec =Vec3.Construct(0,0,0);
     int cohesionNeighborCount=0;
    for(int i=0;i<PLATFORM.MasterTrackList().Count();i=i+1)
    {
      WsfLocalTrack tempTrack= PLATFORM.MasterTrackList().Entry(i);
      cohesionVec=Vec3.Add(cohesionVec,Vec3.Construct(
                     tempTrack.Latitude(),tempTrack.Longitude(),0));
      cohesionNeighborCount=cohesionNeighborCount+1;
    }
    if(cohesionNeighborCount==0)
    {
       cohesionWeight=0;
       cohesionNeighborCount=1;
    }
    cohesionVec.Set(cohesionVec.X()/cohesionNeighborCount,cohesionVec.Y()/cohes
    cohesionVec.Set(-cohesionVec.X()+PLATFORM.Latitude(),-cohesionVec.Y()+PLATF
    cohesionVec=cohesionVec.Normal();

    //Separation Code
    Vec3 separationVec =Vec3.Construct(0,0,0);
    int separationNeighborCount=0;
    for(int i=0;i<PLATFORM.MasterTrackList().Count();i=i+1)
    {
      WsfLocalTrack tempTrack= PLATFORM.MasterTrackList().Entry(i);

      if(tempTrack.GroundRangeTo(PLATFORM)<50000)
      {
    #   writeln(PLATFORM.Name()+" "+tempTrack.TargetName()+" "+
     #                   (string)tempTrack.GroundRangeTo(PLATFORM));

      separationVec=Vec3.Add(separationVec,Vec3.Construct(tempTrack.Latitude(),
```

```
                                tempTrack.Longitude(),0));

    separationNeighborCount=separationNeighborCount+1;
    }
  }
  if(separationNeighborCount==0)
  {
     separationWeight=0;
     separationNeighborCount=1;
  }
  separationVec.Set(separationVec.X()/separationNeighborCount,separationVec.Y
  separationVec.Set(separationVec.X()-PLATFORM.Latitude(),separationVec.Y()-P
  separationVec=separationVec.Normal();

  //Alignment Code
  Vec3 alignmentVec =Vec3.Construct(0,0,0);
  int neighborCount=0;
  for(int i=0;i<PLATFORM.MasterTrackList().Count();i=i+1)
  {
    WsfLocalTrack tempTrack= PLATFORM.MasterTrackList().Entry(i);
    alignmentVec=Vec3.Add(alignmentVec,tempTrack.VelocityNED());
    neighborCount=neighborCount+1;
  }
  if(neighborCount==0)
  {
     alignmentWeight=0;
     neighborCount=1;
  }
  alignmentVec.Set(-alignmentVec.X()/neighborCount,-alignmentVec.Y()/neighbor

  alignmentVec=alignmentVec.Normal();


  //Need custom code to merge them all now.  Inspired from the fusion vector


  double fusedLat=0, fusedLong=0, fusedAlt=0;
  fusedLat=alignmentVec.X()*alignmentWeight/totalWeight+cohesionVec.X()*cohes
           separationVec.X()*separationWeight/totalWeight + avoidVec.X()*avoi
  fusedLong=alignmentVec.Y()*alignmentWeight/totalWeight+cohesionVec.Y()*cohe
           separationVec.Y()*separationWeight/totalWeight + avoidVec.Y()*avoi
  fusedAlt=alignmentVec.Z()*alignmentWeight/totalWeight+cohesionVec.Z()*cohes
           separationVec.Z()*separationWeight/totalWeight + avoidVec.Z()*avoi
avoidVec.SetX(fusedLat);
avoidVec.SetY(fusedLong);
avoidVec.SetZ(0);
```

```
     //Normalize the output to the platforms current position
       avoidVec=avoidVec.Normal();
    WsfGeoPoint newPt=WsfGeoPoint.Construct(PLATFORM.Latitude()-avoidVec.X(),
                                                             PLATFORM.Longitude()-avoidVec
                                                             PLATFORM.Altitude());


    PLATFORM.GoToSpeed(450.0);
          PLATFORM.GoToLocation(newPt);
    end_execute
end_behavior
```

```
processor CombineVector UBFBehavior
     Children
        Behavior Alignment
        Behavior Cohesion
        Behavior Seperation
        Behavior IncreaseVote
     end Children
     Arbiter Fusion_Vote_GeoPoint
end processor
```

```
processor IncreaseVote UBFBehavior
   Execute
      if(UBFBehavior.Get_Number_Of_Actions()<=0)
         {
             return;
         }
         for(int i=0;i<UBFBehavior.Get_Number_Of_Actions();i=i+1)
         {
            UBFBehavior.Get_Action_By_Index(i).Set_Vote(
                  UBFBehavior.Get_Action_By_Index(i).Get_Vote()*2);
                  writeln("    " + (string)UBFBehavior.Get_Action_By_Index(i).Get_
         }
   end Execute
  Children
      Behavior FlyAtPoint
      Behavior FlyAwayFromObstacle2
  end Children
end processor
```

```
#This behavior is meant to show behavioral emergence
processor FlyAwayFromObstacle2 UBFBehavior
   Execute
      WsfGeoPoint choice;
      double currentHeading= PLATFORM.Heading();
      double  choiceDist=-1;
      WsfGeoPoint obstacle = WsfGeoPoint.Construct(30,-80,1000);
      WsfGeoPoint platPoint= PLATFORM.Location();
      WsfGeoPoint currentDirectionPt=
   WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading)/7,
                         platPoint.Longitude()+MATH.Sin(currentHeading)/7,
                         platPoint.Altitude()),
turnDirRight=
   WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading-90)/7,
                         platPoint.Longitude()+MATH.Sin(currentHeading-90)/7,
                         platPoint.Altitude()),
turnDirLeft=
   WsfGeoPoint.Construct(platPoint.Latitude()+MATH.Cos(currentHeading+90)/7,
                         platPoint.Longitude()+MATH.Sin(currentHeading+90)/7,
                         platPoint.Altitude());
      double obstacleTo1dist= obstacle.GroundRangeTo(currentDirectionPt),
            obstacleTo2dist= obstacle.GroundRangeTo(turnDirRight),
            obstacleTo3dist= obstacle.GroundRangeTo(turnDirLeft);
        if(obstacleTo1dist>=obstacleTo2dist &&
                              obstacleTo1dist >= obstacleTo3dist)
      {
         choiceDist=obstacleTo1dist;
         choice=currentDirectionPt;
      }
      else if(obstacleTo2dist>=obstacleTo1dist &&
                              obstacleTo2dist >= obstacleTo3dist )
      {
         choiceDist=obstacleTo2dist;
         choice=turnDirRight;
      }
      else
      {
         choiceDist=obstacleTo3dist;
         choice=turnDirLeft;
      }
      double vote=70000/choiceDist;
      vote=vote*vote;
      writeln("Vote "+(string)vote + " " + (string)choiceDist);
      UBFAction destinationAction = UBFAction.Create("Route",1,vote, choice);
      destinationAction.Set_Int(1);#this is the index of the point it should fly
      UBFBehavior.Add_Action(destinationAction);
```

```
   end Execute
end processor
```

2.5 Arbiter Scripts Used

This section contains the arbiter scripts that were used.

```
processor Fusion_Vote_GeoPoint UBFArbiter
#This arbiter passes up an Action with the GeoPoint's alt, lat, and long values
#combined based on the vote value
#The name and priority fields of the last Action object are reused,
#the other fields are discarded
#Dependancy: none
#INPUT:  Will not respond without valid vote, >0, and valid geopoint, !=null.
#OUTPUT: Single action with fused GeoPoint field according to vote values
   Execute
   int totalVote=0;
   int numofVotes=0;
   string actionName="";
   int actionPriority=-1;
   #Find highest vote number
   if(UBFArbiter.Get_Number_Of_Actions()==0)
   {
      return;
   }
   double fusedLat=0, fusedLong=0, fusedAlt=0;

   for(int i=0;i<UBFArbiter.Get_Number_Of_Actions();i=i+1)
   {
      double tempVote=UBFArbiter.Get_Action_By_Index(i).Get_Vote();
      WsfGeoPoint tempPoint = UBFArbiter.Get_Action_By_Index(i).Get_Geo_Point();
      if(tempVote>0 && tempPoint.IsValid())
      {
         totalVote=totalVote+tempVote;
         actionName=UBFArbiter.Get_Action_By_Index(i).Get_Name();
         actionPriority=UBFArbiter.Get_Action_By_Index(i).Get_Priority();
         numofVotes=numofVotes+1;
         fusedLat=fusedLat + tempPoint.Latitude()* tempVote;
         fusedLong=fusedLong + tempPoint.Longitude()* tempVote;
         fusedAlt=fusedAlt + tempPoint.Altitude()* tempVote;
      }
   }

   if(totalVote<=0)
   {
      return;#no behaviors have valid votes
      #also returns if all votes were 0 since that
      #means no confidence in that action!
   }

   #normalize for the total vote value
   fusedLat=fusedLat/totalVote;
   fusedLong=fusedLong/totalVote;
   fusedAlt=fusedAlt/totalVote;
```

```
    WsfGeoPoint newpoint=WsfGeoPoint.Construct(fusedLat,fusedLong,fusedAlt);
    totalVote=totalVote/numofVotes;
    UBFAction newAction = UBFAction.Create(actionName,
                                actionPriority, totalVote, newpoint);
    newAction.Set_Int(1);
    UBFArbiter.Add_Action(newAction);
    end_Execute
end_processor
```

```
processor CopyAllActionsUp UBFArbiter
 Execute
      UBFAction ab = UBFArbiter.Get_Next_Action();
       while(ab!=null)
         {
            UBFArbiter.Add_Action(ab);
            ab = UBFArbiter.Get_Next_Action();
         }
 end_Execute
end_processor
```

```
processor UBF_A_CheckTrackQualityWeaponsPending UBFArbiter
   Execute
      if(UBFArbiter.Get_Number_Of_Actions()==0)
         return;

      UBFActionList aList = UBFArbiter.Get_Actions_By_Exact_Name("Target");
      UBFAction tempAction =UBFArbiter.Get_Next_Action();
      while(tempAction!=null)
      {
        WsfTrackId tempID=WsfTrackId.Construct(tempAction.Get_String(),
                                                   tempAction.Get_Int());
        if(tempID.IsNull())
        {
           tempAction =UBFArbiter.Get_Next_Action();
           return;
        }
       WsfLocalTrack targetTrack = PLATFORM.MasterTrackList().FindTrack(tempID);
         if(!targetTrack.IsValid())
         {
            tempAction =UBFArbiter.Get_Next_Action();
            return;
         }
         writeln_d ("    targetTrack.TrackQuality == ", targetTrack.TrackQuality());
         if (targetTrack.TrackQuality() < 0.49)
         {
            writeln_d(" FAIL: track quality not good enough to fire on target");
            tempAction =UBFArbiter.Get_Next_Action();
            continue;
         }


         if ((PLATFORM.WeaponsPendingFor(tempID) +
                                         PLATFORM.WeaponsActiveFor(tempID)) > 0)
         {
            writeln("already have weapons assigned for target track");
            tempAction =UBFArbiter.Get_Next_Action();
            continue;
         }
#         else
#         {
#         writeln("no weapons active for target "+tempID.Name());
#         }
         UBFArbiter.Add_Action(tempAction);
         #tempAction= aList.Get_Next_Action();#currently broken
         tempAction =UBFArbiter.Get_Next_Action();
      }
   end_Execute
```

end_processor

```
processor UBF_A_AssignWeaponFromFirstTarget UBFArbiter
   Execute
   // don't launch unless within this percent of Rmax
      double  DefaultPercentRangeMax  = 0.80;
      // don't launch unless beyond this percent of Rmin
      double  DefaultPercentRangeMin  = 1.20;
      Map<string, struct> gWeaponDefs =  Map<string, struct>();

   gWeaponDefs["MEDIUM_RANGE_MISSILE"] = struct.New("WeaponData");
   gWeaponDefs["MEDIUM_RANGE_MISSILE"]->type          = "MEDIUM_RANGE_MISSILE";
   gWeaponDefs["MEDIUM_RANGE_MISSILE"]->rangeMin      = 50;       // (meters)
   gWeaponDefs["MEDIUM_RANGE_MISSILE"]->rangeMax      = 111120;   // ~60 nm (meters
   gWeaponDefs["MEDIUM_RANGE_MISSILE"]->averageSpeed  = 1657.283; //mach 5 (m/s)
   gWeaponDefs["MEDIUM_RANGE_MISSILE"]->maxTimeFlight = 67.05;    //for 60 nm range
   gWeaponDefs["MEDIUM_RANGE_MISSILE"]->numActiveMax  = 2;
   gWeaponDefs["MEDIUM_RANGE_MISSILE"]->domainAir     = true;
   gWeaponDefs["MEDIUM_RANGE_MISSILE"]->domainLand    = false;
   gWeaponDefs["MEDIUM_RANGE_MISSILE"]->maxFiringAngle = 45.0;

   gWeaponDefs["MEDIUM_RANGE_RADAR_MISSILE"] = struct.New("WeaponData");
   gWeaponDefs["MEDIUM_RANGE_RADAR_MISSILE"]->type  = "MEDIUM_RANGE_RADAR_MISSILE";
   gWeaponDefs["MEDIUM_RANGE_RADAR_MISSILE"]->rangeMin      = 50;         // (meter
   gWeaponDefs["MEDIUM_RANGE_RADAR_MISSILE"]->rangeMax      = 111120;     // ~60 nm
   gWeaponDefs["MEDIUM_RANGE_RADAR_MISSILE"]->averageSpeed  = 1657.283;   //mach 5
   gWeaponDefs["MEDIUM_RANGE_RADAR_MISSILE"]->maxTimeFlight = 67.05;      //for 60
   gWeaponDefs["MEDIUM_RANGE_RADAR_MISSILE"]->numActiveMax  = 2;
   gWeaponDefs["MEDIUM_RANGE_RADAR_MISSILE"]->domainAir     = true;
   gWeaponDefs["MEDIUM_RANGE_RADAR_MISSILE"]->domainLand    = false;
   gWeaponDefs["MEDIUM_RANGE_RADAR_MISSILE"]->maxFiringAngle = 45.0;

    gWeaponDefs["SIMPLE_MRM_WEAPON_LC"] = struct.New("WeaponData");
   gWeaponDefs["SIMPLE_MRM_WEAPON_LC"]->type          = "SIMPLE_MRM_WEAPON_LC";
   gWeaponDefs["SIMPLE_MRM_WEAPON_LC"]->rangeMin      = 50;        // (meters)
   gWeaponDefs["SIMPLE_MRM_WEAPON_LC"]->rangeMax      = 111120;    // ~60 nm (mete
   gWeaponDefs["SIMPLE_MRM_WEAPON_LC"]->averageSpeed  = 1657.283;  //mach 5 (m/s)
   gWeaponDefs["SIMPLE_MRM_WEAPON_LC"]->maxTimeFlight = 67.05;     //for 60 nm rar
   gWeaponDefs["SIMPLE_MRM_WEAPON_LC"]->numActiveMax  = 2;
   gWeaponDefs["SIMPLE_MRM_WEAPON_LC"]->domainAir     = true;
   gWeaponDefs["SIMPLE_MRM_WEAPON_LC"]->domainLand    = false;
   gWeaponDefs["SIMPLE_MRM_WEAPON_LC"]->maxFiringAngle = 45.0;


      if(UBFArbiter.Get_Number_Of_Actions()==0)
         return;

      UBFActionList aList = UBFArbiter.Get_Actions_By_Exact_Name("Weapon");
```

```
UBFAction tempAction = UBFArbiter.Get_Next_Action();

while(tempAction!=null)
{
   #first weapon found will be use
    WsfTrackId tempID=WsfTrackId.Construct(tempAction.Get_String(),
                                             tempAction.Get_Int());
    WsfLocalTrack targetTrack = PLATFORM.MasterTrackList().FindTrack(tempID);



    WsfWeapon weapon;
    bool weaponUsable = false;
    int weaponIndex=-1;
    #Check the set of weapons on the platform for one
    #that is compatible with the target
    for (int i=0; i < PLATFORM.WeaponCount(); i+=1)
    {
         weaponIndex=i;
         weapon = PLATFORM.WeaponEntry(i);
         //WeaponCapableAvailableAgainstThreat(weapon, targetTrack) call
         bool WCAAT =false;
         writeln_d("checking if weapon ", weapon.Name(), " is usable.");
         if (weapon.IsNull() || !weapon.IsValid() ||
                  targetTrack.IsNull() || !targetTrack.IsValid())
         {
            writeln_d("weapon or track is not valid!");
            continue;
         }
         if ((weapon.QuantityRemaining()-
                        weapon.WeaponsPendingFor(WsfTrackId())) <= 0)
         {
            writeln_d("no unassigned weapons left to fire!");
            continue;
         }

         //check manually input user data first
         struct weaponData;
         if (gWeaponDefs.Exists(weapon.Type()))
         {
            weaponData= gWeaponDefs.Get(weapon.Type());
         }
         else
         {
            #writeln("TYPE: "+ weapon.Type());
            continue;
            weaponData= struct.New("WeaponData");
```

```
}
if (weaponData->type == weapon.Type())
{
   if ((targetTrack.AirDomain()  && !weaponData->domainAir) ||
       (targetTrack.LandDomain() && !weaponData->domainLand)  )
   {
      #writeln("weapon not capable against target domain!");
      continue;
   }
}
else
{
   writeln("could not find weapon type ", weapon.Type() ,
       " in weapon database; query returned type ", weaponData->type)
   //check if it has a launch computer of the necessary type
   WsfLaunchComputer lcPtr = weapon.LaunchComputer();
   if (lcPtr.IsValid())
   {
      if (targetTrack.AirDomain() &&
                  lcPtr.IsA_TypeOf("WSF_AIR_TO_AIR_LAUNCH_COMPUTER"))
      {
      }
      else if (targetTrack.LandDomain() &&
                         lcPtr.IsA_TypeOf("WSF_ATG_LAUNCH_COMPUTER"))
      {
      }
      else{
         continue;
      }
   }
   else
   {
      #writeln("nor could an applicable launch computer be found!");
      continue;   //dont have weapon data
   }
}
WsfLaunchComputer lcPtr = weapon.LaunchComputer();

if (lcPtr.IsValid() &&
    lcPtr.IsA_TypeOf("WSF_AIR_TO_AIR_LAUNCH_COMPUTER"))
{
   #writeln("            using air-to-air launch computer");


   Array<double> returnedValues = lcPtr.LookupResult(targetTrack);

   // Now have to consider whether we have enough
```

```
#information to continue with a weapon shot:
double theRmax    = returnedValues[0]; //"Rmax";
double theRmaxTOF = returnedValues[1]; //"RmaxTOF";
double theRne     = returnedValues[2]; //"Rne";
double theRneTOF  = returnedValues[3]; //"RneTOF";
double theRmin    = returnedValues[4]; //"Rmin";
double theRminTOF = returnedValues[5]; //"RminTOF";
double range      = targetTrack.GroundRangeTo(PLATFORM);

// Check for track range less than
#Rmin * scaleFactor, if not, return.
// But do not check for min range constraint at
#all unless we are likely to be needing it.
if (range < 5000)
{
   if (theRmin == -1.0)
   {
      continue;
   }
   double RminConstraint = theRmin * DefaultPercentRangeMin;
   if (range < RminConstraint)
   {
      continue;
   }
}

// Check for track range less than Rne,
#if so, FORCE a weapon fire.
bool forceWeaponFire = false;
if (range < theRne)
{
  ## writeln("    Engagement is forcing a
  #weapon fire due to inside Rne.");
  # writeln("    Range versus Rne constraint
  # = ", range, ", ", theRne);
   weaponUsable=true;
   break;
   forceWeaponFire = true;
}

if (forceWeaponFire == false)
{
   theRmax = (theRmax + theRne)/2.0;
   //for highly maneuverable fighter targets
   // Check for track range less than k * Rmax, if not, return.
   if (theRmax == -1.0)
   {
```

```
         #  writeln("    Engagement did not shoot
         #since Rmax was not valid.");
           continue;
        }
        //double RmaxConstraint = theRmax * DefaultPercentRangeMax;
        if (range > (theRmax * DefaultPercentRangeMax))
        {
         #  writeln("    Engagement did not shoot
         # since outside the k * Rmax constraint distance.");
         #  writeln("    Range versus Rmax constraint
         #= ", range, ", ", (theRmax * DefaultPercentRangeMax));
           continue;
        }


     }

#   writeln("    Engagement meets constraints for
#firing a weapon (continue).");
   weaponUsable=true;
   break;
}
else if (lcPtr.IsValid() &&
         lcPtr.IsA_TypeOf("WSF_ATG_LAUNCH_COMPUTER"))
{
   writeln_d("            using air-to-ground launch computer");
   if (lcPtr.CanIntercept(targetTrack))
   {
      //intercept works, this weapon is a candidate
      weaponUsable=true;
      #writeln("weaponusable ------------SET");
      break;
   }
   else
   {
      continue; #continue for loop (int i=0; i <
      #PLATFORM.WeaponCount(); i+=1)
   }
}
else
{
   struct weaponData1;
   if (gWeaponDefs.Exists(weapon.Type()))
   {
      weaponData1= gWeaponDefs.Get(weapon.Type());
   }
   else
```

```
        {
           weaponData1= struct.New("WeaponData");
        }
        writeln_d("            using input WeaponData struct values");

        double effectiveRange      = (PLATFORM.GroundRangeTo(targetTrack)
        PLATFORM.RelativeAltitudeOf(targetTrack)) +
                 PLATFORM.ClosingSpeedOf(targetTrack) * 15; //look ahead 1

        double absRelativeBearing =
                    MATH.Fabs(PLATFORM.RelativeBearingTo( targetTrack ));

        if ((weaponData1->rangeMin *
                             DefaultPercentRangeMin) > effectiveRange)
        {
           writeln_d("            target too close");
           continue;
        }
        if (absRelativeBearing > weaponData1->maxFiringAngle)
        {
           writeln_d("            target firing angle too large");
           continue;
        }
        if (weaponData1->rangeMax *
                          DefaultPercentRangeMax < effectiveRange)
        {
           writeln_d("            target too far away");
           continue;
        }

        double range = PLATFORM.SlantRangeTo(targetTrack);
        double relBearing =
                 targetTrack.RelativeBearingTo(PLATFORM);
        if (relBearing > 90.0)
        {
          if (targetTrack.Speed() > weaponData1->averageSpeed)
          {
            continue;
          }
          double speedDiff = weaponData1->averageSpeed -
                                            targetTrack.Speed();
          if ((range/speedDiff) > weaponData1->maxTimeFlight)
          {
            continue;
          }
        }
     }
```

```
            //END-INRANGETOFIRE
            weaponUsable=true;
            break;#if it made it this far it is usable,
            #continues above will skip this break
       }#End for loop (int i=0; i < PLATFORM.WeaponCount(); i+=1)

       #then no usable weapon was found so try the next target
      if (weaponUsable == false)
      {
         writeln_d("no usable weapon found!");
         tempAction= UBFArbiter.Get_Next_Action();
         continue;#continue While loop on actions
      }

      if (weapon.IsTurnedOn())
      {
         #launched = weapon.Fire(targetTrack);
         UBFAction newAction = UBFAction.Create(
                              "Weapon",2, 1,tempAction.Get_String());
         newAction.Set_Int(tempAction.Get_Int());
         newAction.Set_Double(weaponIndex);
         UBFArbiter.Add_Action(newAction);

         break;#break the while loop because you suggested one action and target
      }

       tempAction= UBFArbiter.Get_Next_Action();
   }#end while loop

 end_Execute

end_processor
```

2.6 Grammar File

This grammar file is used to provide formatting and highlighting to the AFSIM IDE for the new tags. It does not include the new commands because those are automatically created by the AFSIM software. It could be improved to allow for auto-complete of UBFArbiter and UBFBehavior names. It is on the following page.

```
(rule child
{
     Behavior <string>
})
(rule children_block
{
     Children <child>* end_Children
})

(struct UBFBehavior :symbol (type processorType UBFBehavior)
                             :base_type Processor
   (script-var WsfPlatform PLATFORM)
   (script-var WsfProcessor PROCESSOR :this 1)
   (script-var WsfMessage MESSAGE)
   {
        update_interval <real> <time-unit>
  | Execute <ScriptBlock>* end_Execute
  | Map_To_Action <ScriptBlock>* end_Map_To_Action
  | Pre_Condition <ScriptBlock>* end_Pre_Condition
  | <children_block>
  | Arbiter <string>
  | <script-variables-block>
   })


(struct UBFArbiter :symbol (type processorType UBFArbiter)
                             :base_type Processor
   (script-var WsfPlatform PLATFORM)
   (script-var WsfProcessor PROCESSOR :this 1)
   (script-var WsfMessage MESSAGE)
   {
  Execute <ScriptBlock>* end_Execute
   })
```

Bibliography

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

2. E. Winsberg, "Simulated experiments: Methodology for a virtual world," *Philosophy of science*, vol. 70, no. 1, pp. 105–125, 2003.

3. J. Zeh, B. Birkmire, P. D. Clive, A. W. Krisby, J. E. Marjamaa, L. B. Miklos, M. J. Moss, and S. P. Yallaly, "Advanced Framework for Simulation, Integratin, and Modeling(AFSIM) Version 1.8 OVERVIEW Oct 2014," 2014.

4. T. Vu, "Behavior Programming Language and Automated Code Generation for Agent Behavior Control," 2004.

5. M. Colledanchise and P. Ögren, "How behavior trees modularize robustness and safety in hybrid systems," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems.* IEEE, 2014, pp. 1482–1488.

6. A. J. Kamrud, D. D. Hodson, G. L. Peterson, and B. G. Woolley, "Unified behavior framework in discrete event simulation systems," *The Journal of Defense Modeling and Simulation*, 2015.

7. J. B. J. Ziegler and G. Peterson, "An introduction to behavior-based robotics," 2017. [Online]. Available: http://modelai.gettysburg.edu

8. A. Topalidou-Kyniazopoulou, N. I. Spanoudakis, and M. G. Lagoudakis, "A case tool for robot behavior development," in *RoboCup 2012: Robot Soccer World Cup XVI.* Springer, 2013, pp. 225–236.

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

10. J. Goldstein, "Emergence as a construct: History and issues," *Emergence*, vol. 1, no. 1, pp. 49–72, 1999.

11. V. Pemmaraju, "3 simple rules of flocking behaviors: Alignment, cohesion, and separation," https://gamedevelopment.tutsplus.com/tutorials/3-simple-rules-of-flocking-behaviors-alignment-cohesion-and-separation--gamedev-3444, 2013, accessed: 23-Jan-2017.

12. E. Gat, "On three-layer architectures," *Artificial intelligence and mobile robots*, vol. 195, 1998.

13. J. H. Connell, "A behavior-based arm controller," *IEEE Transactions on Robotics and Automation*, vol. 5, no. 6, pp. 784–791, 1989.

14. R. C. Arkin, "Motor schema based navigation for, a mobile robot: An approach to programming by behavior," pp. 264–271, 1987.

15. K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti, "The saphira architecture: A design for autonomy," *Journal of experimental & theoretical artificial intelligence*, vol. 9, no. 2-3, pp. 215–235, 1997.

16. K. L. Myers, "User guide for the procedural reasoning system," *SRI International AI Center Technical Report. SRI International, Menlo Park, CA*, 1997.

17. A. Skorkin, "Why developers never use state machines," 2011, [Accessed 2-Jan-2017]. [Online]. Available: http://www.skorks.com/2011/09/why-developers-never-use-state-machines/

18. C. Simpson, "Behavior trees for ai: How they work," 2014. [Online]. Available: http://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

19. J. Zeh and B. Birkmire, "Advanced framework for simulation, integration, and modeling (afsim) version 1.8 overview," *Wright Patterson Air Force Base, OH: Air Force Research Laboratory, Aerospace Systems*, 2014.

20. 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.

21. M. Mateas and A. Stern, "A behavior language for story-based believable agents," *IEEE Intelligent Systems and Their Applications*, vol. 17, no. 4, pp. 39–47, 2002.

22. A. Marzinotto, M. Colledanchise, C. Smith, and P. Ogren, "Towards a unified behavior Trees framework for robot control," *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 5420–5427, 2014.

23. R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artificial intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

24. J. P. Duffy, "Dynamic behavior sequencing in a hybrid robot architecture," Master's thesis, Air Force Institute of Technology, 2950 Hobson Way, Wright-Patterson AFB, OH 45433, 2008.

25. M. Fowler, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*
23–03–2017

2. REPORT TYPE
Master's Thesis

3. DATES COVERED *(From — To)*
Sept 2015 — Mar 2017

4. TITLE AND SUBTITLE
Extending AFSIM with Behavioral Emergence

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Choate, Jeffrey, L, Capt USAF

5d. PROJECT NUMBER
16 ENG224-24

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-17-M-014

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Aerospace Systems Directorate
Aerospace Vehicles Technology Assessment and Simulation Branch
2180 8th St., B145
WPAFB, OH 45433-7511
Email: brian.birkmire.1@us.af.mil Phone: 937-255-2441

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RQQD

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

12. DISTRIBUTION / AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Advanced Framework for Simulation, Integration, and Modeling (AFSIM) provides a capability to evaluate mission level scenarios described in its scripting language. The AFSIM scripting language includes multiple intelligent agent modeling techniques, none of which explicitly provide the ability to have behaviors emerge. Behavioral emergence occurs when a system composed of many simple behaviors working together exhibits a complex pattern not directly attributable to the simpler components. Without behavioral emergence an intelligent agent designer must explicitly write methods for every combination of circumstances that their agent may encounter. A priori consideration of every possible configuration of the world state is intractable. This problem can be solved by adding the Unified Behavior Framework (UBF) to AFSIM which provides a means to explicitly control behavioral emergence. This thesis adds a unified behavior language built on UBF to AFSIM's scripting language and demonstrates behavioral emergence via a case study of these new behaviors in AFSIM.

15. SUBJECT TERMS
Unified Behavior Framework, Behavior Language, Advanced Framework for Simulation Integration and Modeling

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
U	U	U	U	259

19a. NAME OF RESPONSIBLE PERSON
Dr. G. L. Peterson, AFIT/ENGC

19b. TELEPHONE NUMBER *(include area code)*
(937) 255-3636, x4281; Gilbert.Peterson@afit.edu