The Unified Behavior Framework for the Simulation of Autonomous Agents

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

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

I. Introduction

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

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

In this paper, we will first discuss some relevant background concerning behavior trees, previous applications of UBF, the OpenEaagles framework, and a breakdown of a sweep mission scenario used to test UBF's implementation in OpenEaagles. We then delve into this implementation, examining the UBF structure within OpenEaagles, and the specific implementation of the sweep mission within the our UBF implementation. We will discuss the results of our implementation of the sweep mission, and end with a look at future work and a conclusion.

II. Background

A. Behavior trees

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

In the same way that recent robot architectures focus on individual tasks, or behaviors, an agent's behavior tree executes relatively short behavior scripts directly onto the NPC, so that it exhibits the specified behavior. These scripts are built into a tree structure that is traversed depth-first node-by-node. The tree is queried, or "ticked" at a certain frequency, and behaviors are executed (or not) based on the structure of the tree and the types of nodes that are being ticked. In order to facilitate decision-making, the tree contains multiple types of nodes. As Marzinotto defines them, these node types are either specified as internal or external (leaf) nodes. The internal node types are selector, sequence, parallel, and decorator, while the external/leaf nodes are either actions or conditions. In addition, after being ticked all nodes will either return a success, failure, or running condition, indicating whether the behavior was successful, or if it is still running [9]. Figure 1 provides a simple example of a behavior tree that utilizes at least one of each type of node and implements autonomous vehicle-driving behavior.

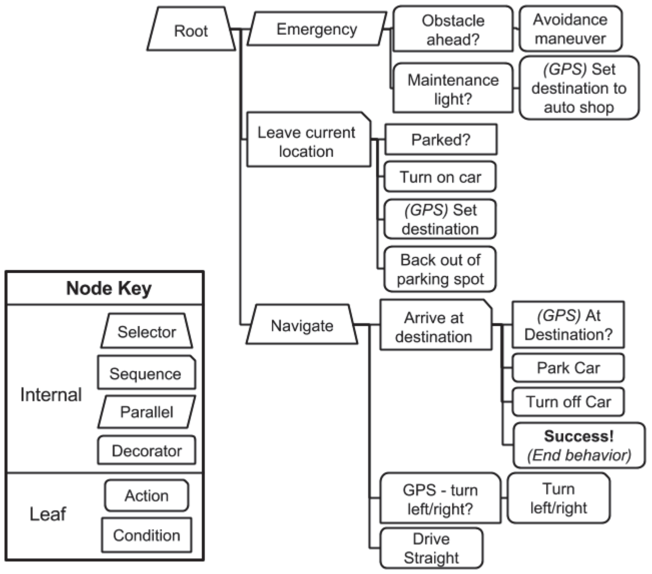

Fig. 1. An example behavior tree implementing autonomous driving behavior (with the aid of a GPS). Note that the nodes in the tree will be "ticked" from top to bottom, implying that behaviors higher in the tree have higher priority.

At the leaf level, action nodes are the only nodes that actually implement control steps upon the agent. When an action node is ticked, it will execute the control step and return running until the control step is complete. Once completed, success or failure return values indicate whether the control step achieved the desired state.

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

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

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

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

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

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

B. Unified Behavior Framework

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

The UBF was initially implemented on robot platforms, as a way to accomplish real-time, reactive robot control [10], [1]. In robot control implementations, a driving factor is the speed with which the robot reacts to the changing environment. Again, the current methodology for ensuring quick response time in reactive control is to tightly couple sensors to actuators through the use of a controller. Figure 2 contains a UML diagram of the Unified Behavior Framework.

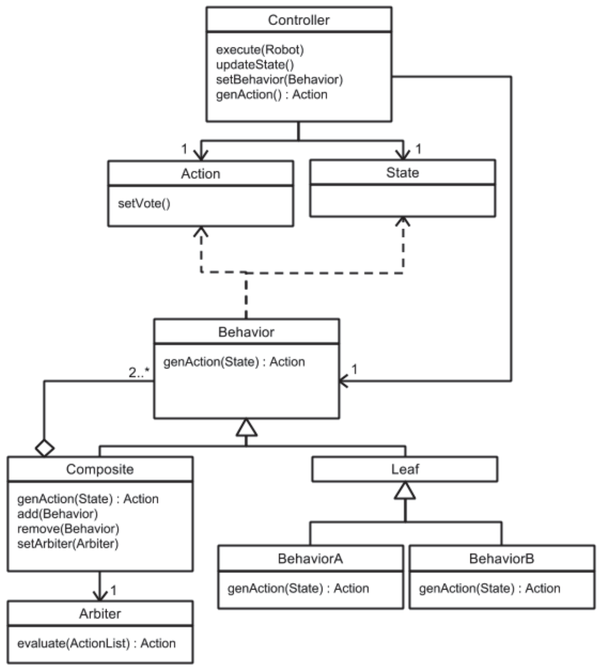

Fig. 2. A UML diagram of the Unified Behavior Framework (UBF) [10].

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

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

As is the case with any robot control architecture, some representation of the real world, or the world model, is present in UBF. This is referred to as the state. Through the updateState() method, the controller interprets the sensor data for the robot. Because of the possible inaccuracies and failures of sensors in robot control applications, the state is described more accurately as the "perceived state," as the actual world state cannot be known, but can only be interpreted based on input from the sensors.

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

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

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

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

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

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

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

C. OpenEaagles Simulation Framework

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

OpenEaagles is an open-source simulation framework that defines the design pattern shown in Figure 3 for constructing a wide variety of simulation applications. The framework itself is written in C++ and leverages modern object-oriented software design principles while incorporating fundamental real-time system design techniques to build time sensitive, low latency, fast response time applications, if needed. By providing abstract representations of many different system components (that the object-oriented design philosophy promotes), multiple levels of fidelity can be easily intermixed and selected for optimal runtime performance. Abstract representations of systems allow a developer to tune the application to run efficiently so, for example, interaction latency deadlines for human-in-the-loop simulations can be met. On the flip side, constructive-only simulation applications that do not need to meet time-critical deadlines can use models with even higher levels of fidelity.

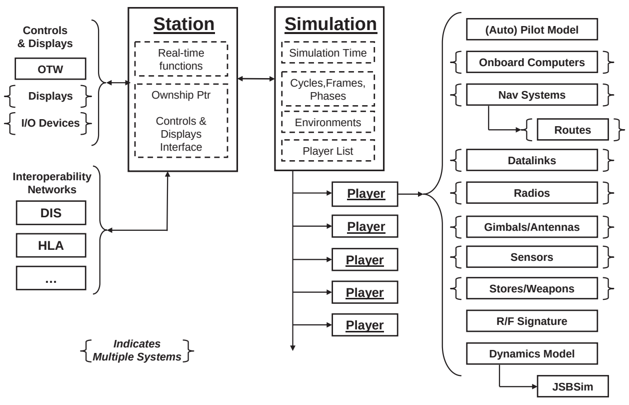

Fig. 3. A graphical depiction of the structure of the OpenEaagles simulation framework.

The framework embraces the Model-View-Controller (MVC) software design pattern by partitioning functional components into packages. As shown, the Station class serves as a view-controller or central connection point that associates simulation of systems (M) with specific views (V) which include graphics, I/O and networks in the case of a distributed simulation.

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

Currently, OpenEaagles has a sophisticated autopilot system, but that is the extent of built-in mechanisms for player or entity autonomy. Other than that, no AI exists in the framework for making simulation entities autonomous. Due to UBF's abstract design structure, it was implemented within OpenEaagles as a set of cooperating classes to define agents which can be attached to Players (i.e., entities) to provide more intelligent features than currently available. Within this structure, UBF agents have access to Player state (world model) and all Player systems which are attached as components such as antennas, sensors, weapons, etc. The Players themselves also include a sophisticated autopilot system which can be used to augment and provide low level control functionality.

III. Methods & results

To demonstrate and test our implementation of UBF within OpenEaagles, we defined a sweep mission scenario. Due to differences between a robot platform and a simulation environment, appropriate adjustments had to be made before implementing UBF. After a discussion concerning revision made to UBF, we revisit the sweep mission to discuss details of bringing it to life. Finally, we will discuss the specific UBF behaviors built and utilized by our agents to successfully navigate this defined scenario.

A. Scenario Description

A simple military mission known as a "sweep" was defined to demonstrate and test UBF-based agents. In this mission, a flight of friendly aircraft navigate towards enemy-controlled or contested airspace. The friendly aircraft search for and engage any enemy aircraft encountered, leaving the area upon destruction of the enemies or an emergency condition being met. The mission is split into four phases: Ingress, Beyond Visual Range (BVR) Engagement, Within Visual Range (WVR) Engagement, and Egress. Figure 4 and Figure 5 depict graphically the progression of the typical sweep mission.

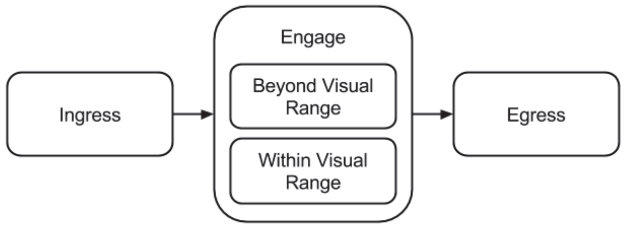

Fig. 4. A graphical depiction of the sweep mission phases.

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

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

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

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

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

B. The UBF Implementation in OpenEaagles

Using the basic structure of UBF as described in section II-B, the architecture was built on top of the object system defined by OpenEaagles. Then, abstract classes defined by the architecture were extended to provide specific functionality (i.e., behaviors, arbiters) relevant to the sweep mission being implemented. Some changes were made to the original UBF structure to tailor it to the OpenEaagles simulation environment, which are described in detail in the following sections. Figure 6 contains a UML diagram of the UBF including the changes that were made in the OpenEaagles implementation.

1) Agent: As discussed in section II-B, UBF within a robot control application provides flexibility between platforms by utilizing different controllers that interface to hardware. Within a simulation environment, hardware is simulated, and can be accessed through the OpenEaagles object system. Therefore, a controller isn't implemented in the same way as it would be on a robot. Also, within OpenEaagles, player entities are built using the composite design pattern; each entity is a composite of many individual components, which each are composites

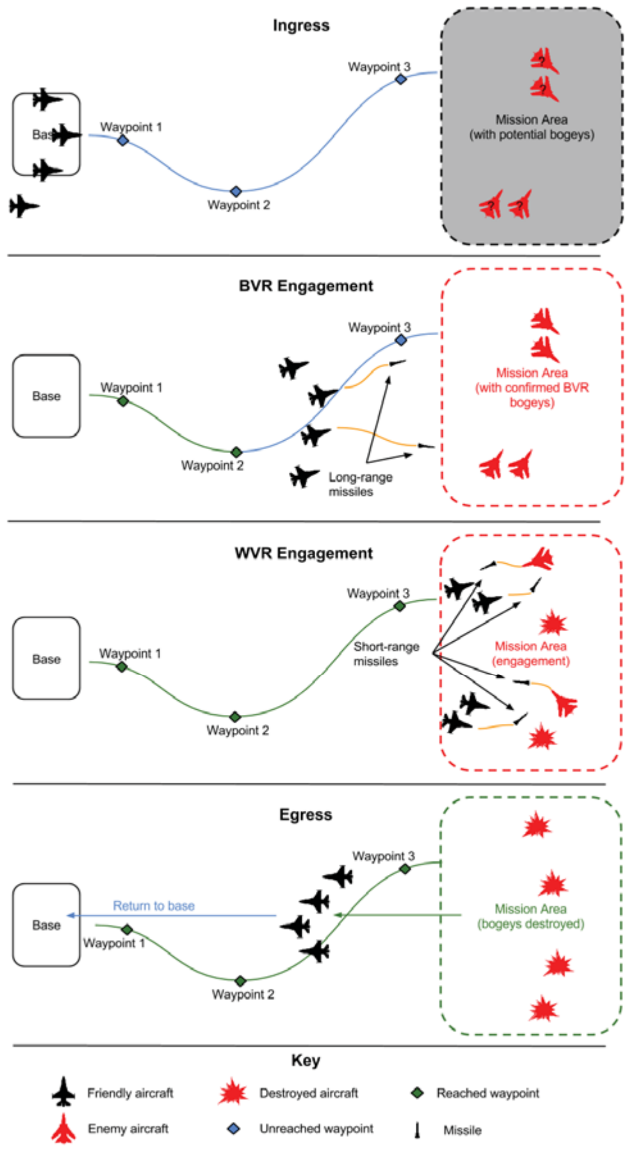

Fig. 5. A birds-eye-view depiction of the sweep mission.

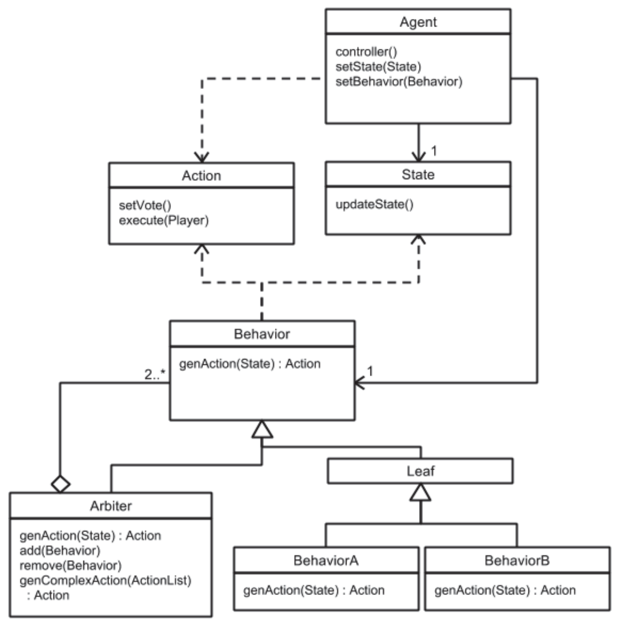

Fig. 6. A UML diagram of the OpenEaagles implementation of the Unified Behavior Framework (UBF).

of their own components. To maintain consistency with this design pattern, UBF needed an overarching component object that contains the whole of the UBF structure. The most effective method was to create an Agent class that contains multiple elements of the UBF, namely, the controller, the root behavior, and the state. This UBF agent can be added - as a component - to an (intended) autonomous player entity in order to add UBF functionality. Through the periodic time phase updates of the Simulation, the agent trickles down requests for updates to the state, and requests for execution of the actions on the autonomous player entity.

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

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

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

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

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

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

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

C. Scenario Implementation

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

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

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

2) Scenario Arbiters, State, and Action classes:

a) Arbiters: As mentioned in section II-B3, there are a variety of arbitration schemes available to facilitate decision making in the UBF behavior tree. In our scenario, two separate Arbiters were designed. Unfortunately, due to time constraints, only one was tested and verified with the sweep mission scenario.

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

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

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

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

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

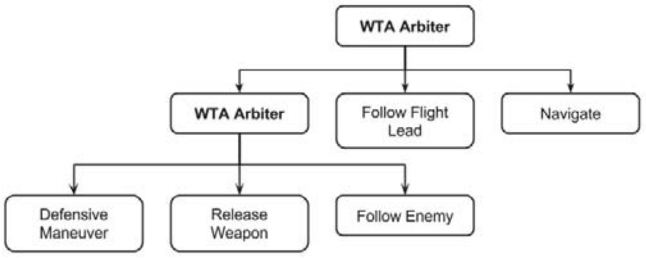

Fig. 7. The UBF behavior tree for the sweep mission scenario.

3) UBF behaviors: In designing the scenario, multiple behaviors were created so that the pilot agents could seek out their sweep mission goal of destroying the enemy aircraft. These behaviors are implemented for the agent's navigation, missile evasion, pursuit of the enemy, and weapons release. They are discussed individually in the sections below. Figure 7 shows the UBF tree structure for the sweep mission scenario.

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

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

Because a particular formation is specified, the wall formation (shown in figure 8), the UBF agents can use their flight ranking to determine their physical position in relation to the flight lead. In this way, the UBF agent can tell its flight ranking based on the players predefined name assigned when constructing the simulation. To make things simpler, a naming convention of "*(flight name)(rank)*" was used to identify which flight the agent is a part of, and their intended rank in the flight. As rank could change if flight leaders were shot down, there was a mechanism built into PlaneStates updateState() method that determines the actual current ranking, rather than just the original predefined ranking.

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

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

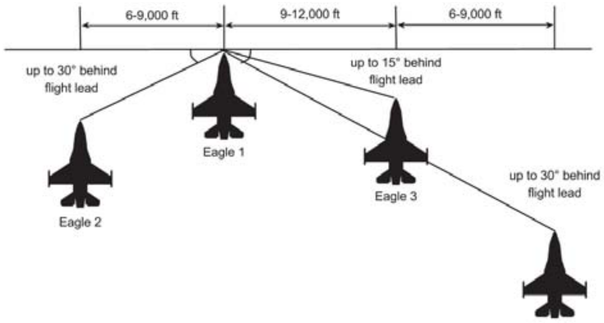
6-9,000 ft
9-12,000 ft
6-9,000 ft
up to 30° behind flight lead
up to 15° behind flight lead
up to 30° behind flight lead
Eagle 1
Eagle 2
Eagle 3
Eagle 4

Fig. 8. A graphical depiction of the "wall" flight formation.

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

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

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

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

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

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

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

IV. Assessment

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

A. Granularity of behaviors

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

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

B. UBF versus Behavior Trees

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

V. Future Work

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

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

VI. Conclusion

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

Acknowledgment

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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