2007

An Initial Methodology For The Definition And Implementation Of Unmanned Aerial Vehicle Agent Behaviors

William Eric Marsh

Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Engineering Commons

Repository Citation

https://corescholar.libraries.wright.edu/etd_all/85

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact corescholar@www.libraries.wright.edu, library-corescholar@wright.edu.
AN INITIAL METHODOLOGY FOR THE DEFINITION AND IMPLEMENTATION OF UNMANNED AERIAL VEHICLE AGENT BEHAVIORS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering

by

WILLIAM ERIC MARSH
B.S., Arizona State University, 2004

2007
Wright State University
March 28, 2007

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY WILLIAM E. MARSH ENTITLED "AN INITIAL METHODOLOGY FOR THE DEFINITION AND IMPLEMENTATION OF UNMANNED AERIAL VEHICLE AGENT BEHAVIORS" BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN ENGINEERING.

____________________________________
Raymond R. Hill, Ph.D.
Thesis Director

____________________________________
S. Narayanan, Ph.D., P.E.
Department Chair

Committee on
Final Examination

____________________________________
Raymond R. Hill, Ph.D.

____________________________________
S. Narayanan, Ph.D., P.E.

____________________________________
Frank W. Ciarallo, Ph.D.

____________________________________
Joseph F. Thomas, Jr., Ph.D.
Dean, School of Graduate Studies
ABSTRACT

Marsh, William . M.S.E., Department of Biomedical, Industrial, and Human Factors Engineering, Wright State University, 2007. An Initial Methodology For The Definition And Implementation Of Unmanned Aerial Vehicle Agent Behaviors.

In many current agent-based modeling systems, it is difficult for a domain-expert user to define and implement agent behaviors without possessing extensive programming knowledge. MUAVES is an existing simulation environment that serves as a research testbed for examining command and control issues with unmanned aerial vehicle (UAV) systems containing many vehicle agents. In its previous form, defining agent behaviors required knowledge of the C# programming language that some MUAVES users did not have. This thesis presents a new methodology for the definition and implementation of UAV agent behaviors in MUAVES. The new methodology is based on diagramming an agent’s controller state. No programming knowledge is required to reuse modular behaviors and trigger conditions specified by previous researchers. The definition of novel behaviors has also been improved by placing behavioral code in external library files, away from the main simulation code. These novel behaviors can be implemented at any desired level of abstraction. After describing the methodology, some sample scenarios are presented as proofs-of-concept.
# Contents

1 Introduction 1

1.1 Role Of Humans In Agent Modeling 1

1.2 Objectives Of This Research 1

1.3 Multiple UAV Virtual Environment System 2

1.4 Outline Of Thesis 3

2 Background 5

2.1 Agent-Based Systems 5

2.1.1 Agents 5

2.2 Agent Autonomy 6

2.2.1 Observe-Orient- Decide-Act (OODA) Loop 7

2.2.2 Reactive vs. Deliberative Agents 7

2.2.3 Rationality And Optimal Outcomes 9

2.2.4 Artificial Intelligence Planning Research 9

2.2.5 Simulation-Based Planning 10

2.2.6 Rule Bases 10

2.2.7 Self-Adaptation 11

2.2.8 Hierarchical Nature of Behaviors 12

2.3 Multi-Agent Systems and Implications 12
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Some categorized trigger conditions</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>A pyramid showing sample behaviors and how they fit into the hierarchy</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>A hierarchy showing a possible decomposition of behaviors</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>The new entity run loop state logic</td>
<td>36</td>
</tr>
<tr>
<td>4.5</td>
<td>A generic state diagram</td>
<td>39</td>
</tr>
<tr>
<td>4.6</td>
<td>A generic state diagram using method names</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>A state diagram for Scenario 1</td>
<td>42</td>
</tr>
<tr>
<td>5.2</td>
<td>The state diagram from Figure 5.1 after conversion to method names</td>
<td>43</td>
</tr>
<tr>
<td>5.3</td>
<td>A screenshot of Scenario 1</td>
<td>44</td>
</tr>
<tr>
<td>5.4</td>
<td>A state diagram for Scenario 2</td>
<td>45</td>
</tr>
<tr>
<td>5.5</td>
<td>The state diagram from Figure 5.4 after conversion to method names</td>
<td>47</td>
</tr>
<tr>
<td>5.6</td>
<td>A state diagram for Scenario 3</td>
<td>49</td>
</tr>
<tr>
<td>5.7</td>
<td>The state diagram from Figure 5.6 after conversion to method names</td>
<td>52</td>
</tr>
<tr>
<td>5.8</td>
<td>A screenshot of Scenario 3</td>
<td>53</td>
</tr>
<tr>
<td>5.9</td>
<td>A state diagram for Scenario 4, UAV1</td>
<td>55</td>
</tr>
<tr>
<td>5.10</td>
<td>A state diagram for Scenario 4, UAV2</td>
<td>55</td>
</tr>
<tr>
<td>5.11</td>
<td>The state diagram from Figure 5.9 after conversion to method names</td>
<td>57</td>
</tr>
<tr>
<td>5.12</td>
<td>The state diagram from Figure 5.10 after conversion to method names</td>
<td>57</td>
</tr>
</tbody>
</table>
5.13 A screenshot of Scenario 4 .......................................................... 58
5.14 A state diagram for Scenario 5 ..................................................... 60
5.15 The state diagram from Figure 5.14 after conversion to method names .... 61
5.16 A screenshot of Scenario 5 .......................................................... 61
5.17 A state diagram for Scenario 6, UAV1 ......................................... 63
5.18 A state diagram for Scenario 6, UAV2 ......................................... 64
5.19 The state diagram from Figure 5.17 after conversion to method names .... 66
5.20 The state diagram from Figure 5.18 after conversion to method names .... 67
5.21 A screenshot of Scenario 6 .......................................................... 68
Listings

4.1 Generic text input file .................................................. 40
5.1 UAV1.uav - Text input file for UAV1 .................................. 43
5.2 UAV1.uav - Text input file for UAV1 .................................. 48
5.3 UAV1.uav - Text input file for UAV1 .................................. 54
5.4 UAV1.uav - Text input file for UAV1 .................................. 59
5.5 UAV2.uav - Text input file for UAV2 .................................. 59
5.6 UAV1.uav - Text input file for UAV1 .................................. 61
5.7 UAV1.uav - Text input file for UAV1 .................................. 66
5.8 UAV2.uav - Text input file for UAV2 .................................. 67
A.1 Trigger method to test for closeness to a waypoint .................. 71
A.2 Trigger method to test for closeness to the base ..................... 71
A.3 Trigger method to test for low fuel level .............................. 72
A.4 Trigger method to test for nearby targets .............................. 72
B.1 Method to initialize action library ..................................... 74
B.2 Method to get the current vehicle location ............................ 74
B.3 Method to get the current fuel level ................................... 74
B.4 Method for vehicle movement to a particular waypoint ............ 75
B.5 Method for vehicle movement to its base .............................. 75
B.6 Method for vehicle movement to a target .............................. 75
B.7 Method for waypoint traversal
Introduction

1.1 Role Of Humans In Agent Modeling

Agents can serve many impressive purposes, particularly in simulation environments. Although an agent, once constructed, is capable of operating without direct human control, its initial characteristics and behaviors must be defined by its human designers. This involves specifying an agent’s cognitive capabilities and allowable actions as well as more physical characteristics such as sensory capability and attributes of the environment. Once the environment and all agents are defined, the user can run the simulation and make research-relevant observations based on model-generated output (Gilbert and Terna, 1999).

The most obvious method of defining agent behaviors and characteristics is for the designer to directly build them into the software. This requires an in-depth knowledge of the required behavior characteristics and (often extensive) programming knowledge that may not be held by a domain expert. The driving goal of this research is to create a user-friendly methodology for defining agent behaviors, that reflect the designer’s intentions, in a generic, yet expressive, fashion, thereby reducing the need for programming expertise by the domain expert. Such a methodology can significantly enhance human domain expert efforts implementing and exploiting agent-based models.

1.2 Objectives Of This Research
Mert and Jilson (2001) state that a simulation is needed that provides users with the ability to flexibly control agent behavior. This research presents a methodology bridging the gap between the definition and implementation of unmanned aerial vehicle (UAV) agent behaviors as realized in MUAVES, a simulation environment built at Wright State University for UAV modeling and research purposes. An initial methodology coupled with a rudimentary set of behaviors implemented in MUAVES serves as a proof-of-concept. Functional testing was performed to ensure proper operation of these behaviors. Since this is an initial research effort, focused on a functional proof-of-concept, no user testing has been employed but is left for future research.

1.3 Multiple UAV Virtual Environment System

The MUAVES architecture is a multi-UAV virtual testbed environment. It is a flexible, portable, and extensive framework implemented in .Net using the C# programming language. The objective of MUAVES is to facilitate basic research on UAV modeling and control by supporting various control paradigms ranging from full human control to fully autonomous behavior. MUAVES is intended to be easily extendable for research in various UAV domains.

Within MUAVES the characteristics of various entity types and terrain characteristics are defined using a configuration interface. The MUAVES simulation component uses the output of the configuration interface to get its initial scenario. The simulation component connects to a run interface via a communication infrastructure. The run interface allows the user to view and make changes to the underlying simulation in a dynamic, interactive fashion. The communication infrastructure handles all communication between the simulation and run interface components.

MUAVES is a low-resolution combat model in which each entity is conceptually viewed as an agent. Currently MUAVES supports the inclusion of UAVs, GBots (ground-bots), and targets as entities. These agents’ behaviors are partially defined by creating custom dynamic link library (.DLL) files (Hill, 2006). An advantage of dynamic link libraries is the ability to select libraries at runtime. Also, a program using dynamic linking is automatically updated, without re-compilation,
when a linked library is updated. A disadvantage of dynamic linking is that it is more difficult to implement than static linking.

MUAVES is a distributed simulation architecture with four basic component types:

**Configuration Interface** – This is where the model parameters are defined prior to running the simulation.

**Server** – This handles the communication between clients.

**Simulation Client** – This encompasses all of the computation. Each entity runs as a separate thread of execution. All vehicle behaviors are implemented in this component.

**Runtime Interface** – This is where the current state of the simulation model is displayed to any users. Each user employs this interface to make run-time changes, such as modifying waypoints, to the simulation model.

### 1.4 Outline Of Thesis

The remainder of this thesis is organized as follows.

Chapter 2 provides a literature review on agent behaviors and their definition. This begins with the definition of an agent and proceeds to describe relevant issues such as autonomy, agent interaction in multi-agent systems, imperfect information, and rule bases. Finally, existing methodologies for the definition and implementation of agent behaviors are presented. Chapter 3 presents the problem statement, issues that were addressed (including constraints on the solution) and how the literature review applies to the research. Chapter 4 presents a proposed solution to the agent behavior definition problem. Two taxonomies are formulated for the combat unmanned aerial vehicle domain: one for trigger conditions and another for behaviors. Each taxonomy is described. Finally, the new methodology is presented. Chapter 5 details the proof-of-concept scenarios created to test
the functionality of the new methodology. Chapter 6 summarizes the research, draws conclusions, and outlines possible future research directions.
Background

2.1 Agent-Based Systems

Agent-based modeling is a natural means of describing a system via a collection of autonomous, decision-making entities known as agents. Agents can be added or modified with ease, making agent-based modeling a flexible solution and ideal approach for describing a system of interacting behavioral entities (Bonabeau, 2002a). A frequent challenge for users of an agent model is building and implementing the set of agent behaviors.

2.1.1 Agents

The word “agent” has its origins in the Latin word “agens”, meaning something that acts. There are a variety of opinions regarding what exactly is required to be an agent. Most authors agree that an agent is an active software component that must be situated in an environment, able to collect input from that environment, and able to cause a change to that environment through its actions. Typically agents are also flexible and autonomous. For an agent to be flexible, it must be pro-active, responsive, and social. For an agent to be autonomous, it must be able to control it’s behavior, to some extent, without external intervention (Wooldridge and Jennings, 1995; Jennings et al., 1998; Huhns and Stephens, 1999; Erlenbruch, 2002).

Mert and Jilson (2001) describe tangible and intangible characteristics that an agent may possess. In the UAV domain, tangible characteristics might include attributes such as sensor and
weapon ranges, fuel capacity, and speed. Intangible characteristics might include attitude, discipline, obedience, and motivation.

2.2 Agent Autonomy

There are varying degrees of autonomy present in agent systems. Xu and Shatz (2003) compare autonomy to human free-will. Having autonomy gives an agent the ability to perform tasks to meet design objectives without direct external intervention (Erlenbruch, 2002). Some agents have little autonomy in that their behavior is governed by a set of rules. Other, more autonomous, agents are capable of logical or pseudo-logical reasoning about possible behavioral alternatives and their consequences (Bhargava and Branley, 1995; Wooldridge and Jennings, 1995). Some of these reasoning agents are even capable of learning based on past experience to improve outcomes over time.

Some domains are better suited to the use of autonomous agents than others. In order to accept high levels of autonomy, domain users must trust the agent and expect it to act effectively on their behalf (Luck et al., 2004). In the UAV domain, particularly the military domain, high levels of autonomy may never be appropriate due to safety and accountability concerns. For example, it may be possible for a UAV to use some means of automatic target recognition to identify a UAV as an enemy target and then attack it. In most circumstances, this is undesirable behavior. Before attacking a target, the military needs absolute certainty the identification is correct and must be able to assign accountability in the case of a mistake. Of course, in the case of a simulation environment, these concerns are less relevant, but form an important research agenda.

In some domains it may be possible, and desirable, to place constraints on an agent’s allowable behaviors. For example, it may be appropriate to constrain behaviors in order to obey social laws or physics laws (Gordon, 2000). Also, in some situations, the choice of actions may be stochastic (Graves et al., 2000), such as found in various complex adaptive simulation models of combat.
2.2.1 Observe-Orient-Decide-Act (OODA) Loop

An agent must continually monitor its environment and act based on its observations. A common way of describing this process is the Observe-Orient-Decide-Act (OODA) loop. Hill et al. (2003) list four steps in the OODA loop as follows:

1. Observe – observe environment stimuli
2. Orient – internalize stimuli
3. Decide – develop set of possible actions
4. Act – implement selected action

The time required for an agent to complete an OODA loop is known as its OODA cycle time. Minimizing this time is important in real-life battles because if an agent can operate ‘within’ an enemy agent’s cycle time, that faster agent has an advantage (Hill et al., 2003).

2.2.2 Reactive vs. Deliberative Agents

As mentioned above, there are varying degrees of autonomy present in agent systems. For instance, reactive agents have little autonomy while deliberative agents are highly autonomous.

Reactive Agents

A reactive agent perceives and responds when its environment changes (Xu and Shatz, 2003). The behavioral decisions made by these agents are similar to reflexes (Remondino and Cappellini, 2006) and result from a direct mapping of situation to action. This approach is simple and robust against failure. The problem with this approach is that knowledge of the past is usually not taken into account because no internal state is maintained. This lack of memory means that decisions are based solely on the current state of the system and environment. One implication of this approach is
that the agent does not learn from experience and as such fails to improve performance (Erlenbruch, 2002).

**Deliberative Agents**

A deliberative (or cognitive) agent also reacts to its environment, but does maintain an internal state and thus has a stronger sense of agency (Remondino and Cappellini, 2006). What separates a deliberative agent from a reactive agent is the ability to reason about its actions (Luck et al., 2004). When speaking of this strong sense of agency, we often speak of concepts that are typically applied to humans. These include the concepts of knowledge, belief, intention, obligation, and even emotion (Wooldridge and Jennings, 1995).

A deliberative agent has unique “goals” that drive its behavior. A goal-driven agent is highly autonomous and proactive. Proactiveness is the degree to which an agent behaves in a goal-directed manner (Wooldridge and Jennings, 1995; Xu and Shatz, 2003). In this case there exists a mapping between the agent inputs (state information and goals) and the agent outputs (actions) (Bates et al., 1991). Goals are desired local states, desired end states, selective rewards to maximize, or internal motivations that are maintained within particular bounds (Ilachinski, 1997).

**Belief-Desire-Intention (BDI) Architectures**

A common group of deliberative architectures involve the use of Belief-Desire-Intention (BDI) architectures. In these architectures, data structures are used to represent beliefs, desires, and intentions of an agent. Beliefs are the agent’s perceptions about the environment. Desires are like goals. Intentions represent the chosen course of action to achieve those goals. They provide stability for decision making but are sometimes reconsidered. Once an agent decides which goals it wants to achieve, it must decide how to achieve them before actually taking action (Erlenbruch, 2002). If an agent intends to act, the likelihood of the action is increased, but usually no guarantee is made for a particular action (Wooldridge and Jennings, 1995).
Hybrid Agents

Hybrid architectures combine aspects of both reactive and deliberative agents. A hybrid agent is composed of at least two subsystems, a deliberative and a reactive subsystem. The deliberative subsystem contains a symbolic world model used by the agent to plan and make decisions. Since reaction-time suffers in deliberative agents, the reactive subsystem typically takes precedence over the deliberative component to provide faster response to events when necessary (Wooldridge and Jennings, 1995).

2.2.3 Rationality And Optimal Outcomes

A rational agent should be self-interested. It should choose behaviors that maximize achievement of its goals while minimizing cost (Castelfranchi and Conte, 1998). A rational agent employs the correct behavior. It is also important to consider what is an “optimal outcome”. Ideally, rational behavior produces optimal outcomes. But what is optimal? A simple approach to rationality is known as “economic rationality”. The logic behind economic rationality is that if everything can be reduced to currency, it is easy to speak of maximizing it. In combat the “currency” might be the amount of damage inflicted. Although this approach is commonly used, a disadvantage is that it can often be difficult to reduce the preferences of an agent to scalars, particularly monetary scalars (Huhns and Stephens, 1999). In practice, implementations assume the scalar preference used captures the behaviors desired.

2.2.4 Artificial Intelligence Planning Research

In the early days of agent systems, decision-making was based on planning systems influenced by the domain of artificial intelligence (AI). Planning research is the sub-field of AI deciding what action to take to reach a particular goal. AI planning systems use first principles to decide how agents act. This requires agents to formulate an entirely new plan to satisfy a given goal. Agents
execute a continuous cycle of picking a goal, generating a plan, and executing the generated plan (Jennings et al., 1998).

The problem with this AI-based planning approach is that it does not scale well to larger systems due to “calculative rationality”. A rational system gives an optimal choice if performed at the time that decision-making began. However, in a realistic situation (with a high level of complexity), a search over the space of all possible choices will take long enough that the decision may no longer be useful (Jennings et al., 1998). Thus for real-time agent systems, the scaling hurts performance. In non-real-time systems the scaling means large computational requirements.

2.2.5 Simulation-Based Planning

One interesting, non-deterministic approach to making decisions is known as “simulation-based planning”. In this approach, simulations are embedded within one-another. The embedded simulations are used to evaluate and compare the outcomes of the various choices. An example might be a simulation whose agent considers three alternatives. Each alternative is “played” out by the simulation with the best alternative implemented in the actual simulation execution (Lee and Fishwick, 1997).

2.2.6 Rule Bases

The nature of combat often allows it to be described, in models, by very simple rules. When using these simple rules, and allowing the agents to interact, interesting behavioral results may emerge (Bjorkman, 2000).

A rule base may be expressed in many ways, including a tree structure or a fixed-length string with a finite alphabet. Its content may resemble that of a military manual that describes proper procedures for conducting an operation under given circumstances. Different unit types might often have different manuals for the completion of an operation. A rule base is consulted at discrete
time steps during which an agent evaluates its status and that of the environment to determine an appropriate action (Graves et al., 2000; Mert and Jilson, 2001).

Gilbert and Terna (1999) say that a rule has two parts:

**Condition** – specifies when the rule should “fire”

**Action** – specifies what should happen when the rule “fires”

In an autonomous agent system, the selection of an appropriate, if not optimal, rule-set is critical. In the combat domain, an ‘optimal’ rule set might give a side its best chance of winning the battle. Commonly, a rule base undergoes a heuristic ‘tweaking’ in which an analyst using the model implementing the rule set systematically alters factors in search for an ideal set of rules (Graves et al., 2000). More ambitious tweaking may involve heuristic optimization methods, a field often referred to as simulation-based optimization.

### 2.2.7 Self-Adaptation

Ideally, the designer of an agent system could pick appropriate rules to optimize behavioral choices and thereby optimize outcomes. In practice, it is not possible to predict all possible environmental conditions an agent may encounter (Luck et al., 2004). An adaptive agent can evolve or learn in order to improve future outcomes or, in the case of combat forces, simply survive in an ever-changing environment (Ilachinski, 1997). Commonly, genetic algorithms are used for agent evolution. An agent may learn by analyzing past outcomes and attempting to optimize future outcomes. Also some agents evolve by simply imitating other agents that have performed well (Cohen et al., 1999).

### Genetic Algorithms

A genetic algorithm allows a rule set to evolve using natural selection. A possible solution to a problem is called a “chromosome”. A chromosome is often defined as a binary bit-string. Through
the use of genetic operators, a new population of chromosomes is created from the most fit chromosomes of the initial population. Typical genetic operators are “reproduction”, “crossover”, and “mutation”. When the reproduction operator is applied, the initial chromosome multiplies as-is. The mutation and crossover operators actually alter the initial chromosome. In the case of crossover, segments of two initial chromosomes are swapped when creating their new counterparts. In mutation, a single bit is flipped when creating a new chromosome (Ilachinski, 1997).

2.2.8 Hierarchical Nature of Behaviors

An individual agent may have several competing goals at a given time. When multiple simple behaviors take place in parallel, their combination can be complex (Jennings et al., 1998).

The subsumption architecture uses a hierarchy to prioritize parallel behaviors. In this architecture, behaviors compete with one-another for control. Primitive behaviors, such as obstacle avoidance, are located at lower layers of the hierarchy and are given precedence over higher layers (Wooldridge and Jennings, 1995). The Brawler air-to-air simulation model uses a value model to prioritize behaviors (Bent, 2003a).

2.3 Multi-Agent Systems and Implications

A Multi-Agent System (MAS) is an environment comprised of more than one agent working together with other agents. A multi-agent system is often referred to as a complex adaptive system. Each agent in a MAS has incomplete information or limited abilities. Therefore, agents have limited, and differing viewpoints. In a MAS, there is no global control system and data is decentralized (Jennings et al., 1998; Remondino and Cappellini, 2006).

This introduction of multiple agents into an environment makes agent adaptation more complex. Once other agents learn, the environment has effectively changed (Jennings et al., 1998). This
causes a cycle in which adaptive agents must continually adapt to the adaptations of other agents (Erlenbruch, 2002).

2.3.1 Interactions Between Agents

Since each agent in a multi-agent system lacks a complete picture of the global environment and has limited capabilities, communication between agents is important for coordination (Xu and Shatz, 2003). Communication may take place directly between agents, or the environment may act as an intermediary (Batty and Jiang, 1999). These interactions not only allow for the communication of plans and goals, but also provide agents with a means to predict or form expectations regarding future actions of other agents and thus adjust their own plans accordingly (Luck et al., 2004). Communication also allows for explicitly modeling teamwork. Sometimes it is beneficial for teams to monitor their collective performance and reorganize efforts accordingly (Jennings et al., 1998).

2.3.2 Emergent Behaviors

As an alternative to top-down behaviors being imposed, each agent in a MAS chooses its behaviors locally (without a centralized planning function) without the help of an abstract world model. When simple, bottom-up rules are used in this manner, unexpected and surprising behaviors on the global level may result from interactions between individual entities. In fact, the structure and pattern of the resulting behaviors, though often counterintuitive, may give the illusion of intelligence and demonstrate self-organization (Frelinger et al., 1998; Berry et al., 2002; Bonabeau, 2002a; Hill et al., 2003).

These interactions between individual entities cause the system’s emergent behavior to arise (Bonabeau, 2002a). If the individual entities in a group make only minor behavioral changes, the group’s emergent behavior may change radically (Bonabeau, 2002b). In addition, global phenomena may often affect the local components and thereby influence local behaviors as well (Gilbert and Terna, 1999).
Many real-world examples of emergent behaviors exist and have been the focus of past research, including:

**The fighting ability of a combat force** – Ilachinski (1997) states that this "cannot be understood as a simple aggregate function of the fighting ability of individual combatants”.

**The flocking of birds** – Reynolds (1987) investigated how the aggregate behavior of a flock of birds is caused by interactions between individual birds with simple behaviors. In his model, each simulated bird perceives its environment and acts independently. The interaction between these birds creates an illusion of central control.

**Traffic jams** – Bonabeau (2002b) describes a traffic jam as a “distinct entity that emerges” from the behaviors of individual drivers “trying to get somewhere” while following or breaking various legal, societal, and personal rules.

**Panic-induced crowd stampedes** – Bonabeau (2002a) claims that when people panic, they are concerned only with short-term personal interests as opposed to social constraints.

**Tornados** – Ilachinski (1997) states that "an air molecule is not a tornado”.

The emergence of complex behaviors is a key benefit when choosing to use an agent-based system for modeling. If emergent phenomena are expected to exist in a system, a designer should consider the use of agent-based modeling as the emergent behavior can become the most beneficial aspect of the system (Bonabeau, 2002a; Bonabeau, 2002b).

**Complicated vs. Complex**

Remondino and Cappellini (2006) make a distinction between a complicated system and a complex system. A complicated system has many distinct parts, but interactions among the parts yield completely predictable behavior. Examples of complicated systems include airplanes and computers. The long-term behavior of a complex system is difficult to model. Examples of complex systems include ecological and economic systems. Complex systems do not need to be complicated systems.
2.3.3 Resolution

In combat modeling, resolution refers to the level of modeling detail. For instance, a high-resolution aircraft model might explicitly model physical forces such as drag and lift while in a low-resolution model the plane merely demonstrates altitude and speed. MUAVES is considered a “low-resolution” combat model.

Combat may be modeled at the level of the individual entity or at a higher level, such as that of a team of UAVs. In MUAVES, each UAV is modeled as a separate entity. This bottom-up approach provides a detailed model of warfare, one in which emergent behavior can be observed.

2.4 Imperfect Information

The simulator component in a software package has access to all information in the simulation scenario. In a combat simulation, this includes knowledge about all agents, their activities, the terrain, and communications. For realism, an individual agent should only have access to a limited subset of this complete set of world knowledge. This means each agent has an incomplete and imperfect view of its environment (Bhargava and Branley, 1995; Gaudiano et al., 2003). Bhargava and Branley (1995) claim that an entity performs the following three basic (error-prone) steps when reasoning:

1. Detection – Stimuli are acquired; these may be faulty or incomplete
2. Measurement – Stimuli are classified, named, identified, etc.; these may be faulty due to incomplete knowledge or incorrect pattern-matching.
3. Interpretation – Conclusions are formed and plans are made; an incomplete rule base may cause interpretation to be faulty.

In many applications, it may be useful to distinguish between global and local information. Global information is available to any agent in the simulation. Local information is available only
to agents within a specific range or in a specific subset. The amount of knowledge represented is application-specific. An implication of this is that agents are more likely to interact with, and influence, nearby agents rather than those that are far away. This makes the cellular automata paradigm a reasonable choice to represent some agent systems (Gilbert and Terna, 1999).

It is important to distinguish between true, static knowledge and dynamic beliefs. Static knowledge might be information such as terrain maps and equipment manuals. Dynamic beliefs might involve enemy positions and intentions. Not only are these beliefs subject to change, but they are influenced by other agents (Bhargava and Branley, 1995).

2.5 Combat Mission Simulations

A simulation of a combat mission is often based on very simple rules that specify agent behaviors. Agents in these scenarios typically navigate to a series of operational points via waypoints along a route. The action taken at a point could vary depending on the operational point, what trigger condition is active, and what agent goals are defined for the point (Mert and Jilson, 2001).

Often operations are thought to have three basic operational points: a starting point, objective points, and an endpoint. Any of these points may share the same location (perhaps when defending a position), there may be more than one objective point, and waypoints may exist between the operational points. Agents may or may not take specific action at a point. A unit may perform different actions depending on its position with respect to these points. For example, if a unit is fired upon en-route to an objective point a different behavior may be chosen than if it is fired upon once it has reached it’s objective point (Mert and Jilson, 2001). This is an example of an agent changing behavior based on environmental stimuli and its internal goals and desires.
2.6 Existing Agent Methodologies

Many multi-agent systems currently exist. Each system provides some means for the agent designer to embed his/her choice of agent behaviors in the system. Methodologies vary in user-friendliness and flexibility.

2.6.1 Methodologies In Other Domains

Discussing agents in the combat domain helps to limit the scope of possible methodologies. However, there is overlap and it may be possible to learn from the techniques used in other domains.

Swarm and MAML

Swarm was designed to facilitate the creation of multi-agent simulations. A Swarm system consists of a hierarchy of agents organized into “swarms”. In this way, a swarm is a group of agents, an agent can be composed of swarms, or an agent can actually be a swarm. Each agent can have a set of reactive rules or, if an agent is a swarm, its behavior emerges from the agents in that swarm. The swarm is a natural means of encapsulation that allows for the flexible representation of multi-level simulations, such as those where agents have beliefs about their environment ([Minar et al., 1996]).

Swarm systems are created using Objective C, an object oriented programming language. Each agent is implemented as an object of a particular class. This is beneficial because it allows for models to be constructed from reusable components. Swarm also contains several class libraries to be reused when building models ([Minar et al., 1996]).

The “activity” library is particularly relevant to behaviors within swarms. Agent and swarm actions are controlled by “schedules”. A schedule is built of instances of activities and defines a sequence of discrete events on agents ([Minar et al., 1996]).

Swarm is difficult for a domain expert without programming knowledge to use. Attempts have been made to bridge this gap. László Gulyás and Corliess (1999) have created a macro language
and design interface for use with Swarm. The Multi-Agent Modeling Language (MAML) is a macro-language designed to make the process of creating a model easier for non-programmers. It is more abstract than pure object-oriented code and is designed to use concepts from the agent-based discrete-event simulation domain. The Model Design Interface (MDI) is a graphical CASE tool, implemented in Java, used to design and implement models (László Gulyás and Corliss, 1999).

**ZEUS Agent Building Tool-Kit**

Collis et al. (1998) created the ZEUS agent building tool-kit to assist in the creation of collaborative agent systems. These collaborative agents are typically autonomous and cooperation with other agents is an important trait. This cooperation is beneficial in situations where each agent has limited resources or abilities. By emphasizing the pooling of resources and abilities, these agents can be used for larger tasks collectively than individually. The ZEUS agent building tool-kit is a library of components coupled with a development methodology to aid in the engineering of these applications.

A generic ZEUS agent includes many components. Following is a list of the ones particularly relevant to choice of behaviors (Collis et al., 1998):

**Co-ordination Engine** – This is where goal-related decisions are made. This is also where coordination with other agents takes place. This involves task delegation and working toward a shared goal.

**Acquaintance Model** – This keeps track of inter-agent relationships as well as beliefs regarding other agents.

**Planner and Scheduler** – This uses the decisions made by the coordination engine and information about available resources to plan actions.

**Task/Plan Database** – This keeps track of known tasks. Also reactive behaviors for certain events are stored here.
The ZEUS agent development methodology can be summarized as follows (Collis et al., 1998):

1. The problem domain is analyzed, looking for autonomous entities that could be modeled as agents.

2. Attributes of each agent are identified. This includes the resources and capabilities of the agents.

3. Costs, duration, preconditions, products, and constraints are determined for each task. Relationships between agents and beliefs that agents have about each other are specified.

4. Based on the societal status and role of each agent, appropriate coordination protocols are specified.

5. Code is automatically generated.

ZEUS agent generator software has been created to assist with the methodology above. This software allows for a user to define agent attributes visually. Agent information can be viewed and altered at run-time, allowing for behavioral changes (Collis et al., 1998).

2.6.2 Methodologies In The Combat Domain

Since land and air combat have many similarities, examples of both are described in this section.

ISAAC (Irreducible Semi-Autonomous Adaptive Combat)

ISAAC is an agent-based simulation system of land combat. It employs a bottom-up approach using heterogeneous agents. ISAAC was designed as an environment where researchers can explore how low-level rules can influence emergent behavior on the global level. It can be used to model emergent behaviors, such as breakthroughs and encirclement, that were not explicitly programmed into the model. Basic combat and movement rules are intended to be low in numbers and as simple as possible (Ilachinski, 1997).
Each agent in ISAAC has an embedded doctrine, mission, situational awareness, and a mechanism that allows for the adaptation of rules. The doctrine is realized as a default local rule set which specifies how an agent is to act in a generic environment. Behavior-directing goals define the mission. Situational awareness is achieved through the use of sensors, which generate an internal map of the situation to be used by the agent (Ilachinski, 1997).

An ISAAC agent moves toward or away from particular goals, and decision-making takes place at the level of the individual agent. Decisions are made based on the goals and motivations of a given agent. Each agent has a personality weight vector with the following fields representing goals (Ilachinski, 1997):

- alive friendly
- alive enemy
- injured friendly
- injured enemy
- friendly flag
- enemy flag

The values in the weight vector indicate how much importance an agent places on being close to each respective goal. A negative value in the weight vector indicates that the agent would rather increase distance from that goal. A penalty function weights the distances from other agents and flags according to the personality vector. The agent moves in the environment to incur the smallest penalty (Ilachinski, 1997).

An agent’s default rules can change at each time step based on the environment. Each agent has a set of meta-rules that dictates how it will adapt its personality at each time-step. These meta-rules specify constraints that must be met. If they are not met, the personality weights are altered for the current time-step (Ilachinski, 1997).

A low-level agent can only access local information within a finite range. The ranges for various types of information are specified by the researcher. The following ranges are set (Ilachinski, 1997):
Sensor range – the range at which an agent can sense other agents

Fire range – the range at which an agent can engage another agent

Threshold size – the size of the boxed area around an agent that can be used in decision-making

Movement range – how far an agent may choose to move in one time step

Communications range – an agent communicates the contents of its sensor field to other agents within this range

ISAAC provides a command and control hierarchy of local and global commanders in a simulation scenario. The user is the “Supreme Commander”. He defines the scenario and all relevant parameters and dispositions. Global commanders operate within the scenario defined by the supreme commander and use global information to control interaction between local commanders. Local commanders then aggregate the information from all of the subordinate low-level agents within their command area. A low-level agent accesses only local information (Ilachinski, 1997).

If a low-level ISAAC agent is under the control of a local commander, additional weights are added to its personality vector (Ilachinski, 1997):

- Stay close to Local Commander
- Obey orders of Local Commander

A user can fully define an ISAAC scenario by creating a lengthy data input file with several sections. It is also possible to set the personality and performance characteristics of an agent using on-screen prompts on the agent definition screen. If the simulation is in “Interactive Mode”, the user can view and adjust an agent’s personality vector and constraints on the fly while the simulation is running (Ilachinski, 1997).
Brawler

Brawler is an engagement-level simulation system for few-on-few air combat (Bent, 2003a). An emphasis is placed on individual pilot situational awareness and cooperative tactics among a flight of aircraft. Brawler is considered a high-resolution simulation model due to its detailed modeling of hardware and physical effects (Buschor, 1998). Brawler is known to produce realistic behavior in its simulated pilots (Bjorkman, 2000).

A pilot is modeled as a fully-functional decision-making entity. The pilots’ decision process is modeled in terms of missions, tactical doctrines, aggressiveness, perceived enemy capability, reaction time, and decision quality. A pilot chooses actions during each “conscious event” based on its perception of the current combat situation, along with programmed doctrine-based tactics. Each alternative set of actions considered by a pilot is projected into a future state and a score is explicitly assigned based on the desirability of this future state (Buschor, 1998).

The selection of “skill levels” is a way of modeling limited pilot capabilities. An assumption is made that greater pilot proficiency translates to greater situational awareness. The pilot skill levels vary in two regards: the number of aircraft that can be tracked simultaneously and the pilot’s memory length. There are three skill levels in Brawler: Rookie, Pilot, and Ace. A rookie can track up to three aircraft, a pilot can track up to five aircraft, and an ace can track an unlimited number of aircraft. The ace skill level is likely not realistic because it gives pilots a global information view (Buschor, 1998).

Buschor (1998) lists the following three phases of a “conscious event”:

1. Situation Assessment – In this phase, a pilot updates its mental model to include any new sensory data.

2. Decision – Based on the updated mental model and alternative evaluation, the pilot makes a decision.
3. Execution – In this phase the decided-upon action is carried out. In some cases this may not be a direct action (such as maneuvering or use of weapons). The action may instead be to set future objectives.

A decision hierarchy is used in Brawler, providing a layered decision-making process. In this way, high-level decisions control low-level decisions. The hierarchy is as follows (Buschor, 1998):

- Flight Posture Decision
- Flight Tactics Decision
- Pilot Posture Decision
- Maneuver Decision and Weapons Employment Decision

The flight posture decision serves to determine the general course of action. The flight lead then makes the flight tactics decision. In this decision, tactics are chosen to implement the flight posture. Value-driven decisions at lower levels are influenced by the flight tactics decision. A pilot then makes a set of pilot posture decisions which involve the immediate objectives for the individual aircraft (Bent, 2003a). The pilot posture decision is responsible for deciding which enemy aircraft are important and selecting a weapon. The pilot is somewhat more inclined to obey an “order” from the flight tactics decision, but may also exhibit common sense. Finally, the pilot makes the maneuver decision and the weapons employment decision (Bent, 2003b).

Three types of decision processes are used in Brawler (Bent, 2003a):

- Value-Driven Decisions
- Traditional Decision Rules
- Production Rules

Value-driven decisions are used for the maneuver decision, pilot posture decision, and flight posture decision. They are robust and flexible but explicit “knowledge” must be present in the evaluation function. This type of decision is useful when there is a large number of objectives that are weighed against one another (Bent, 2003a).
Traditional decision rules involve logic branching based on present environmental conditions. This type of decision process is best for cases involving very simple, well defined choices. Traditional decision rules are used in matters such as whether to fire a weapon or wait, missile avoidance, or ground avoidance. The rules are difficult to modify and lack flexibility as they are often hard-coded into the software. The order of the rules considered implies the relative importance of each factor (Bent, 2003a).

A production rule consists of a condition-action pair whose choice biases the value system. Production rules are useful because they are external to the Brawler program and are therefore easily modified (Bent, 2003a).

An interesting feature of Brawler is its information-oriented simulation architecture used to model pilot perception. The flow of information into each pilot’s mental model is simulated explicitly. The true state of the system is stored in a central status array. Each pilot has its own mental status array that contains its perceived state of the system. This perceived state is not perfect. Some information, such as the location of other aircraft, is uncertain. Some aircraft or missiles may not exist in a pilot’s mental status array, meaning the pilot is unaware of these entities in the system (Bent, 2003a).

Scenario and behavior information is defined in Brawler via text files. These files define the mission, flight tactics, and all required engineering data. Basic flight tactics are considered. The user-defined production rules produce specialized pilot behaviors by modifying the value function choices. Developing these rule sets tend to be knowledge-intensive and quite time-consuming for the Brawler modeler (Bent, 2003a).

**Playbook**

In sports, teams have many predefined dynamic “plays” called by name to express intent and cause each player to perform certain functions. This allows for a large number of independent participants to coordinate effectively. In the context of a sports team, the collection of a team’s plays is referred
to as a “playbook”. Calling a “play” allows for a person in a supervisory role to delegate certain tasks to subordinates without necessarily dictating the exact means used to accomplish the task. In sports, as well as other domains of supervisor-subordinate interaction, it is not usually appropriate for a supervisor to give instructions in terms of individual movements (Miller et al., 2004).

Playbook tasking serves as the basis for SIFT’s1 Playbook system. Playbook is designed to coordinate behavior for multiple, heterogeneous unmanned aerial vehicles. The ability to call “plays” keeps a human supervisor in the decision-loop while giving subordinates varying degrees of autonomy (Miller et al., 2004; Miller et al., 2005).

A “play” is a predefined template created by a human user that defines a common task with certain parameterized elements. Plays are comprised of behaviors chosen from a library and may be nested. These templates help to reduce the supervisor workload while allowing for a more detailed description of behaviors if appropriate (Miller et al., 2004; Miller et al., 2005).

A nice aspect of Playbook is that it allows for “Variable Autonomy Control”. The human user can simply define high-level mission goals in standard situations. However, the user may also decide to describe mission plans in a level of detail appropriate for the current situation. This is done by requiring or prohibiting specific actions or resource usage (Miller et al., 2005). Playbook allows for both “hard” and “soft” constraints. A hard constraint must be met by the subordinate, while a soft constraint is merely a goal. While a user is discouraged from requesting that a play be completed using specific equipment (and thereby short-circuiting the reasoning process), it is possible (Miller et al., 2004). An analogy can easily be drawn to the way that a manager might interact with a subordinate in the workplace.

To use Playbook, UAV agents must be capable of interpreting and applying plays based on current mission context. The Playbook execution environment provides certain intelligent default behaviors such as collision avoidance (Miller et al., 2004).

1Smart Information Flow Technologies - http://www.sift.info
Problem

3.1 Problem Statement

In most current real-life UAV control systems, such as the Predator system, each UAV requires one or more fully-trained operators. The flight path is usually manually specified and then the UAV is remotely controlled. There is little automation present in current systems and there is no way to model operator intent (Miller et al., 2005).

The present situation with UAV simulation software is somewhat better, with the ability to model intent and more advanced levels of autonomy. However, little work has been done to allow the definition of UAV behaviors in a non-programmer, user-friendly manner. Ideally, one need not be a software engineer to correctly specify desired behaviors in a system.

MUAVES is an existing simulation environment of the combat UAV domain. It currently lacks a user-friendly methodology for the definition and implementation of UAV agent behaviors. Presently, a user who wishes to modify the behaviors of an agent must use the C# language. The goal of this research is to all but eliminate the need for programming knowledge and make it possible for any researcher possessing adequate domain knowledge to specify desired behaviors and see those behaviors realized in the simulation.
3.2 Issues That Must Be Addressed

Many aspects of the literature review performed above are relevant and impact the solution to this problem. The following is a list of characteristics that a new methodology must possess or allow for:

- Agents should be able to handle unforeseen conditions. An agent may not be able to actually respond intelligently to every situation that it faces, but it should at least be robust enough that the software does not crash.

- The system should be capable of being extended to support adaptation. By providing this ability to adapt, there is a risk of undesirable side-effects. Global behavior may be affected by local change (Gordon, 2000).

- Agent response should be timely. This refers to response-time with respect to the overall speed of the simulation. Satisfying many behavior requirements will increase the computational complexity of a system, thereby increasing the lag of the system as a whole. It is important to ensure an agent does not waste time before deciding on a course of action.

- The user should be confident that the agent will act effectively, robustly, and reliably (Luck et al., 2004).

- Agents should not be strictly controlled at the level of individual movements. This is not typically how a person controls a human subordinate and it is not an efficient management strategy (Miller et al., 2004). The user should be able to specify behaviors at a level of abstraction appropriate to the scenario.

- The system must be able to handle multiple units, each with different equipment and capabilities.

- The methodology should support a level of user modeling and capability reasonable for a researcher and domain expert.
Methodology

4.1 Taxonomy of Trigger Conditions

Proper timing and coordination are important for successful combat operations (Grieger and Gill, 2001). In air combat, the pilot must react to changing conditions in the environment. Therefore care should be taken when choosing trigger conditions in a combat simulation. A trigger condition may be either a single occurrence or a particular set of environmental conditions used to “trigger” a particular desired action. There are many possible triggers in the combat UAV domain. It is possible to categorize the universe of trigger data into three broad categories. Figure 4.1 graphically depicts a simple scenario with some sample trigger conditions and where they would fall in the taxonomy below.

**Raw Sensor Data** – All data comes from a sensor. This category refers to data measured by a sensor and used in its received form. Examples of raw UAV sensor data include: location of other agent, a sound is detected, line-of-sight to an object exists, an obstacle is detected

**Communication Data** – Data in this category is in the form of a communication from another agent. This could be a request from another agent to perform an action. Communicated data could be simply another agent’s raw sensor data. If an agent knows the location of enemy units, this information can be shared with others (Grieger and Gill, 2001). It is possible that some agents may be deemed more trustworthy than others. This is why communicated data is distinguished from raw sensor data.
Interpreted Data: 
I have a 95% success rate when engaging this type of threat.

Figure 4.1: Some categorized trigger conditions
**Interpreted Data** – This category refers to data that is either statistical in nature or requires additional analysis before triggering a behavior. Examples of interpreted data include: predictions have a certain degree of success, boundary or threshold reached, attack failed, attack succeeded.

Note that there is room for debate on appropriate categorizations. For example, depending on implementation, reaching a boundary can be thought of as either raw sensor data or interpreted data. Here it is assumed that position data is compared to other data (perhaps a terrain map) before making this determination. However, in some scenarios, a boundary may be marked by a beacon. In this case, the trigger may be categorized as raw sensor data.

### 4.1.1 Applicability to the Problem

The new methodology allows for trigger conditions of all categories. Each type of trigger is handled differently from a software standpoint. However, the user of the new methodology will treat each type of trigger with similar ease. If a user wishes to use trigger conditions that have been used previously by other researchers, no programming is required and trigger conditions of all categories are treated exactly the same. The user does not need to understand that the conditions may be tested differently by the software.

### 4.2 Taxonomy of Behaviors

Behaviors are categorized in much the same way as triggers. However there is another dimension to this classification. It is useful to distinguish basic behaviors such as movement in a particular direction from more complex ones such as coordination with another vehicle. This hierarchy can be seen in Figure 4.2.

It is illustrative to examine Figure 4.3. This figure shows how combinations of basic (lower-level) behaviors combine to cause the emergence of higher-level behaviors.
High Level

Middle Level

Movement
Change Altitude

Payload Action

Communication Action

Movement Action

Payload Action

Communication Action

Movement Action

High Level
Suppress

Middle Level
Avoid Obstacles

Movement
Change Altitude

Middle Level
Attack

Movement
Move In Direction

Payload Action

Fire Weapon

Figure 4.2: A pyramid showing sample behaviors and how they fit into the hierarchy

Figure 4.3: A hierarchy showing a possible decomposition of behaviors
At the lowest level, it is useful to categorize combat UAV behaviors into three broad categories:

**Movement Action** – This category includes any action that involves movement. Examples of movement action include: change altitude, change speed, move in a particular direction

**Communication Action** – This category includes any action that involves communication with other agents. Examples of communication action include: accept task, leave ‘pheromone’ signal, call for reinforcement

**Payload Action** – This category includes any action that involves using the UAV payload. Examples of payload action include: fire weapon at target, fire weapon in air, take photograph of target area

Combinations of these lower-level behaviors can cause more intelligent, higher-level behavior to emerge. Although many levels are possible, it is illustrative to speak in terms of a three-level hierarchy. The proposed levels are as follows:

**High Level** – These behaviors are very abstract concepts. Examples of high level behaviors include: cooperate, coordinate, negotiate, learn model of other agents, suppress

**Middle Level** – These behaviors are more concrete than those at the high level, but are still comprised of simpler, lower-level behaviors. Examples of middle level behaviors include: avoid obstacles, collect information, find other agents, assess bomb damage

**Low Level** – These behaviors are atomic; they are not comprised of behaviors from any lower level. Examples of low level behaviors include all behaviors identified in the above categories of agent actions. For example, an agent changing speed is a low level behavior. If it increases speed and changes direction, it may be exhibiting a higher level avoidance behavior.
4.2.1 Applicability to the Problem

As with trigger conditions, the new methodology allows the user to define and implement all categories of UAV behaviors. Some of these behaviors will be more complex, but it is important that a similar approach is taken for all behaviors. The user can control agent behavior at all levels described above. When implementing a novel behavior, the higher its level, the more programming knowledge the user must possess. All behaviors that have been used by previous researchers can be treated in a modular manner such that the current researcher can treat behaviors of all categories the same.

4.3 Designing A Methodology

In this section, the problem solution is presented. The creation of a new methodology was approached by first looking at what the user needs to do and then working towards what the system currently needs as input in order to adequately fulfill the user’s desires.

4.3.1 The Perspective Of The User

The user can benefit from an easy-to-understand methodology to communicate his/her agent-behavior desires in a non-ambiguous fashion. There are particular trigger conditions and behaviors that a user may wish to model in MUAVES. For the methodology to be accessible to non-programmer users, selection of an appropriate input method was critical.

Triggers and Behaviors

A first step is to define the types of triggers and behaviors a user of MUAVES may need to use in their simulations. In sections 4.1 and 4.2, triggers and behaviors for the combat domain were categorized, respectively. Some of these categories are more relevant in MUAVES than others.
MUAVES can model any of them, although some are accommodated better than others. To date, MUAVES has been mainly used to model movement actions using sensor data and interpreted data. This knowledge was valuable when designing and testing the new methodology.

**Potential Input Methods**

Many existing agent modeling systems use plain-text input files to configure agent behaviors. This is perhaps initially the best way to provide users with the ability to specify behaviors in a non-ambiguous manner. This file could either be passed off to an experienced programmer for implementation or the system could parse it (eliminating the need for a “middle-man”). For this methodology, the latter approach was chosen. Of course, if the system is capable of parsing a file, creating an interface allowing the user to define the behaviors is fairly easy. However, designing a usable interface may be more difficult and is an area for future research.

Many engineers, particularly those with computer science backgrounds, are familiar with the use of state diagrams. This seems to be an appropriate starting point for a new methodology. A limit of the use of state diagrams is that it may be difficult to model competing behaviors. In the new methodology, the user uses a plain-text input file to unambiguously describe a state diagram.

**4.3.2 The Perspective Of MUAVES**

MUAVES is the simulation environment where the user’s desired agent behaviors are realized. Since MUAVES is an existing system, the new methodology had to fit within MUAVES’ functionality. The new methodology serves to augment the functionality of MUAVES by making the existing functionality more usable. The creation of a new methodology for the definition of behaviors did not require major architectural changes.
Entity Run Loop

Vehicles in MUAVES execute simultaneously as separate threads in the simulation client. Each UAV executes its respective “run-loop” within the running simulation. All behavioral choices are made within this “run-loop”.

Dynamic Link Library Files

MUAVES was originally designed to handle agent movement using dynamic link libraries (.dll files). This provides a degree of flexibility to change the implementation of agent behaviors at runtime. Previously, behavior code was split between the entity run loop and a library file.

Two observations with respect to MUAVES were immediately relevant. First, for ease of use and flexibility, it is desirable to place all segments of code that a user may normally wish to edit within the .dll files. A user of the methodology should not need to make changes to the entity run-loop. Second, the initial system could be extended to support the use of two separate .dll files at a time: one containing methods that test for trigger conditions, and one that implements the possible behaviors. Both of these changes were made to MUAVES. The trigger condition library is comprised of methods that return a boolean for use in the run-loop decision logic. The behavior library contains code to update the position of a unit, and the position can be queried by the run-loop when needed.

Necessary Changes to MUAVES

A software-based implementation of a state machine was created. This state code is generic and allows for run-time creation of states and state transitions. Several changes were made to the existing MUAVES system. This ability to change the state machine at run-time means functionality can be added allowing the user to change agent behaviors while the simulation is running. Also, the system could be extended in the future to support adaptation by morphing the state machine at run-time. Figure 4.4 shows the flow of the new run-loop. In particular, four specific changes were made:
Initialize Starting State

Send Data to Run Interface

For Each Rule in Current State:

If Trigger Condition for Rule is Satisfied

Switch State Accordingly

Take Action Based on State
Generic State Class – This new class includes instance variables to store the action to be taken while in the state, and a list of trigger/transition pairs.

Rule Class – Each object of this new class defines a transition on a state machine. As is typical for a state machine, a transition condition and a new state are defined.

Run-Loop Decision Logic – The entity run-loop was modified to incorporate decision logic to travel through the new virtual state machine. All code for the actual implementation of behaviors was moved to the .dll files.

DLL Files – A .dll file was created to implement trigger conditions. Each method in this library is static and returns a boolean value to indicate whether the condition was satisfied. Each trigger method takes the calling Entity and an Array of Strings as input. All possible behaviors are implemented in another .dll file. The behavior library file was modified such that all behavior methods now have void return types. For input, they now accept an Array of Strings and a double value used when calculating how far to move. It is important for ease of implementation and future expansion that all general behavior methods have the same signature and all trigger methods have the same signature. For this reason, Arrays of Strings are used exclusively to specify parameters. There are three special methods, with different signatures, included in the new behavior library file:

void init(Entity thisEnt) – This method is called to initialize the behavior library before calling any other methods.

Location GetLocation() – This method is used to retrieve the location that the vehicle should be at based on any past movement actions. This should be called at the end of the entity run-loop to get the result of the behaviors chosen during an iteration of the loop.

double GetFuelRemaining – This method is used to determine the amount of fuel that the vehicle has remaining. It is called at the end of the entity run-loop to get the result of the behaviors chosen during an iteration of the loop.
4.3.3 Diagramming the Behaviors

At an aggregate level UAV behavior can be viewed as a sequence of lower-level behaviors. For instance, a very simple reconnaissance mission can be defined as the lower-level behaviors:

- travel to target,
- loiter at target, and
- return to base.

MUAVES provides the facility to define and incorporate libraries of such lower-level behaviors. The domain expert defining UAV behaviors will likely want their envisioned high-level behaviors defined as a sequence of lower-level behaviors. The methodology supports this view of behavior definition.

In the methodology, a UAV ‘Behavior State’ is defined. Each behavior state is associated with a lower-level behavior from the library of actions. Changing behavior states represents a transition between states caused by some trigger condition. Thus, a domain expert defines UAV behavior as a sequence of transitions between lower-level behaviors. A state-transition diagramming method is employed to capture a domain expert’s desired UAV behavior.

Figure 4.5 depicts an example node-arc state-transition diagram. The nodes represent the lower-level behaviors such as those found in a library of behaviors. The arcs represent conditions causing the transitions among the behaviors states.

4.4 The Methodology

The recommended methodology for the definition of UAV agent behaviors within MUAVES is an ordered sequence of steps.

1. Draw a state diagram depicting the desired flow of agent behaviors. A generic state diagram is shown in Figure 4.5. Each state should have a name associated with it as well as the desired
UAV action while in the state. Each arc connecting the states specifies a transition condition. An engineer or domain expert with no programming knowledge should be able to create a state diagram.

2. Using the state diagram, choose appropriate .dll files containing the methods required to perform the actions and test the scenario trigger conditions. Place these files in the simulation libraries folder. For highly specialized behaviors, new library methods may need to be created. This is obviously more difficult than using predefined libraries, but it is much easier than altering code within the entity run-loop as the implementation of such behaviors previously required. The files TriggerSet.dll and ActionSet.dll should be the only .dll files residing in the working libraries folder at any time.

3. Convert the state diagram generated previously so that it uses method names included in the chosen trigger and behavior libraries. Also include any other parameters required by the method, as shown in Figure 4.6 using the ‘:’ to begin the parameter list and to separate parameters. It is important that any method referred to in the state diagram actually exists in the library files. A generic example can be seen in figure 4.6. Note that parameters are
optional and there may be any number of them. The parameters must match what is expected by the chosen method.

4. Create a text file from the state diagram using the format shown in Listing 4.1. Every state should be specified with a ‘#’ and every transition condition should be specified using ‘*’. The trigger conditions and behaviors specified in this file must exactly match the names of methods in the .dll files that will be used. Save the text file in the folder containing the behavior and trigger libraries following the naming convention “UAV_Name.uav”. Future extensions of this methodology will automate the text file generation step.
Proof-Of-Concept Testing

This chapter recounts the proof-of-concept testing performed on the methodology prototype.

5.1 Scenario Revisions

5.1.1 Scenario 1 - Basic Movement Functionality

The purpose of the first proof-of-concept scenario was to test whether changes to MUAVES functioned as expected and to demonstrate the new methodology.

This very simple scenario involved a single unmanned aerial vehicle repeatedly flying on a mission from its base to a waypoint and then returning to base. A simple scenario with only one UAV and one waypoint was created in the Configuration Interface.

Step 1 - Drawing a State Diagram

The state diagram in Figure 5.1 shows a simple scenario to test movement functionality. While in State 0, the UAV moves toward Waypoint 1. Once Waypoint 1 is reached, the UAV enters State 1. While in State 1, the UAV moves toward its base. Once its base is reached, the UAV enters State 0 and the process repeats.

Step 2 - Choosing Libraries

From the state diagram, the following functionality is needed:
Figure 5.1: A state diagram for Scenario 1

- **Triggers:**
  - Waypoint Reached
  - Base Reached

- **Actions:**
  - Move Toward Waypoint
  - Move Toward Base

The chosen trigger library has the following methods:

- `waypointReached`
- `baseReached`

The method `baseReached` needs no parameters. The method `waypointReached` needs a waypoint number.

The chosen action library has the following methods:

- `moveToWP`
- `moveToBase`
Figure 5.2: The state diagram from Figure 5.1 after conversion to method names

Listing 5.1: UAV1.uav - Text input file for UAV1

#0 moveToWP:1
#1 moveToBase
*0>1 waypointReached:1
*1>0 baseReached

The method moveToBase requires no parameters. The method moveToWP requires a parameter specifying the number of the waypoint to move toward.

These .dll files were placed in the simulation libraries folder.

Step 3 - Converting the State Diagram to use Method Names

The state diagram in Figure 5.2 is identical to the diagram in Figure 5.1, except the text has been altered so that it now exactly matches method names found in the action and trigger libraries. Also, parameters have been added.
Figure 5.3: A screenshot of Scenario 1

Step 4 - Create Text File

The text file shown in Listing 5.1 was created from the state diagram in Figure 5.2.

Results

The scenario worked as expected. The vehicle repeatedly flew back and forth between the base and the waypoint. A screenshot of this scenario is in Figure 5.3.
5.1.2 Scenario 2 - Return to Base for Refueling

Another aspect of basic movement functionality is the ability for a vehicle to return to base when a low-fuel condition is encountered. This example illustrates the use of vehicle status information in trigger conditions.

In this scenario, a single unmanned aerial vehicle flies between its base and a single waypoint, as in the first scenario. The difference from the first scenario is that the vehicle will now realize when it becomes low on fuel and fly back to base. For the purposes of this test, the waypoint is placed far enough away from the base that the vehicle cannot possibly make it to the waypoint and return to base without running out of fuel. The expected behavior is for the vehicle to fly part-way to the waypoint and then return to base once it realizes that its fuel level is insufficient for a safe return to base.

Step 1 - Drawing a State Diagram

The state diagram in Figure 5.4 shows an extended version of scenario 1. No new states were needed for this scenario. All that was added is a new transition from State 0 to State 1. The trigger for the
new transition is that the vehicle becomes low on fuel. While in State 0, the UAV moves toward Waypoint 1. As in scenario 1, once Waypoint 1 is reached, the UAV enters State 1. While in State 1, the UAV moves toward its base. Once its base is reached, the UAV enters State 0 and the process repeats. If at any point on its way to the waypoint the vehicle determines that it is low on fuel, the vehicle will enter State 1. Once in State 1, the vehicle will move toward the base as usual.

**Step 2 - Choosing Libraries**

From the state diagram, the following functionality is needed:

- **Triggers:**
  - Waypoint Reached
  - Base Reached
  - Low Fuel
- **Actions:**
  - Move Toward Waypoint
  - Move Toward Base

The chosen trigger library had the following methods:

- waypointReached
- baseReached

For this scenario, a new method was added to the trigger library; a method named *lowFuel*. This method tests whether the vehicle has enough fuel to return to base if traveling further. The addition of the new method demonstrates the user’s ability to add new trigger conditions. With the low fuel trigger condition now implemented (assuming that such a simple implementation meets their needs), future researchers can use it instead of creating their own.

No parameters are needed for the new trigger method. Only information on the vehicle’s current location, the location of the base, and the current fuel level is needed. This information is already available to the trigger libraries.
Figure 5.5: The state diagram from Figure 5.4 after conversion to method names

As in scenario 1, the chosen action library has the following methods:

- moveToWP
- moveToBase

These .dll files were placed in the simulation libraries folder.

Step 3 - Converting the State Diagram to use Method Names

The state diagram in Figure 5.5 was created in the same manner described in scenario 1. The only difference between it and the state diagram shown in Figure 5.2 is the addition of a lowFuel transition.

Step 4 - Create Text File

The text file shown in Listing 5.2 was created from the state diagram in Figure 5.5.
Listing 5.2: UAV1.uav - Text input file for UAV1

#0 moveToWP:1
#1 moveToBase
*0>1 waypointReached:1
*0>1 lowFuel
*1>0 baseReached

Results

The scenario worked as expected. The waypoint was placed far enough away from the base that the UAV would have no way of making the entire trip without running out of fuel. The vehicle never made it to the waypoint, as it always needed to refuel before it got there.

5.1.3 Scenario 3 - Circle a Target

In addition to knowing how to move from Point A to Point B, it is also important that a vehicle can notice a nearby target and move in for a closer look. The purpose of the third scenario was to prove that the methodology supported the ability to detect a target and then move toward it.

In this scenario, the vehicle is instructed to fly to a waypoint. If a target is detected en-route to the waypoint, the vehicle will go to the target and circle the area (perhaps to perform reconnaissance). When the vehicle gets low on fuel, it returns to base. It then moves between the base and the waypoint repeatedly.

Step 1 - Drawing a State Diagram

The state diagram for scenario 3 is shown in Figure 5.6. This diagram is more elaborate than the previous ones, but the actions and triggers are very similar. Note that there is a subtle difference between State 0 and State 4. In State 0, the UAV is searching for a target while moving to Waypoint 1. In State 4, the UAV is simply moving to the waypoint.
Figure 5.6: A state diagram for Scenario 3
State 2 has the action *Move to Target* and the only way out of that state is if the vehicle runs low on fuel. The reason for this is that since MUAVES operates at discrete time steps and the vehicle “jumps” from location to location, the vehicle will never truly reach the target location exactly. This means that the vehicle will move back and forth, emulating a “circling” behavior until it runs low on fuel.

**Step 2 - Choosing Libraries**

From the state diagram, the following functionality is needed:

- **Triggers:**
  - Waypoint Reached
  - Base Reached
  - Low Fuel
  - Target Detected
- **Actions:**
  - Move Toward Waypoint
  - Move Toward Base
  - Move Toward Target

The chosen trigger library had the following methods:

- `waypointReached`
- `baseReached`
- `lowFuel`

The chosen action library has the following methods:

- `moveToWP`
- `moveToBase`
New methods were added to the action and trigger libraries. A method named `moveToTarget` was added to the action library. A similar method named `targetSensed` was added to the trigger library. This trigger method detects when a target is within the vehicle’s sensor range. For this functionality to work, the MUAVES vehicle Sensor class was extended to include a method to return an ArrayList of all detected targets. The trigger method checks whether that list is empty. The action method moves to the first target in that list. This method is not particularly intelligent. In a more realistic scenario, target ranking may be appropriate. Such a ranking behavior could be added in future MUAVES versions.

No parameters are needed for either of the new methods. The detection range was already defined in the Sensor class.

These .dll files were placed in the simulation libraries folder.

**Step 3 - Converting the State Diagram to use Method Names**

The new state diagram in Figure 5.7 was created as in the previous scenarios.

**Step 4 - Create Text File**

The text file shown in Listing 5.3 was created from the state diagram in Figure 5.7.

**Results**

The scenario worked as expected. A screenshot can be seen in Figure 5.8.

**5.1.4 Scenario 4 - Multiple Vehicles**

Thus far, the merits of this methodology were demonstrated with respect to single-vehicle behavioral control. The methodology must also support the definition of behaviors for multiple UAVs. In this scenario, basic movement of two UAVs is demonstrated.
Figure 5.7: The state diagram from Figure 5.6 after conversion to method names
Figure 5.8: A screenshot of Scenario 3
Listing 5.3: UAV1.uav - Text input file for UAV1

#0 moveToWP: 1
#1 moveToBase
#2 moveToTarget
#3 moveToBase
#4 moveToWP: 1
*0>1 waypointReached : 1
*0>1 lowFuel
*1>0 baseReached
*0>2 targetSensed
*2>3 lowFuel
*3>4 baseReached
*4>3 lowFuel
*4>3 waypointReached : 1

This scenario consists of two vehicles, both departing from the same base but following two different paths. Past scenarios only involved one waypoint. In this scenario, one vehicle must traverse two waypoints (and the base) while the other vehicle has a single but different waypoint.

Step 1 - Drawing State Diagrams

This scenario requires two state diagrams, one for each vehicle. If vehicle behaviors are the same, a single state diagram will suffice but a separate text file must be created for each vehicle.

The state diagram shown in Figure 5.9 is for UAV1. It has a state for the movement to each waypoint. The state diagram for UAV2 is shown in Figure 5.10. It has only two states because the vehicle simply moves between the base and a single waypoint.
Figure 5.9: A state diagram for Scenario 4, UAV1

Figure 5.10: A state diagram for Scenario 4, UAV2
Step 2 - Choosing Libraries

From the state diagram, the following functionality is needed:

- **Triggers:**
  - Waypoint Reached
  - Base Reached

- **Actions:**
  - Move Toward Waypoint
  - Move Toward Base

The chosen trigger library had the following methods:

- waypointReached
- baseReached

The chosen action library has the following methods:

- moveToWP
- moveToBase

No new methods are needed; this functionality is similar to previous scenarios. These .dll files were placed in the simulation libraries folder.

Step 3 - Converting the State Diagram to use Method Names

The new state diagrams are seen in Figures 5.11 and 5.12.

Step 4 - Create Text File

The text files for UAV1 and UAV2 are shown in Listings 5.4 and 5.5, respectively. These were derived from the state diagrams in Figures 5.11 and 5.12, respectively.
Figure 5.11: The state diagram from Figure 5.9 after conversion to method names

Figure 5.12: The state diagram from Figure 5.10 after conversion to method names
Figure 5.13: A screenshot of Scenario 4
Listing 5.4: UAV1.uav - Text input file for UAV1

#0 moveToWP: 1
#1 moveToWP: 2
#2 moveToBase
*0>1 waypointReached: 1
*1>2 waypointReached: 2
*2>0 baseReached

Listing 5.5: UAV2.uav - Text input file for UAV2

#0 moveToWP: 1
#1 moveToBase
*0>1 waypointReached: 1
*1>0 baseReached

Results

The scenario worked as expected. A screenshot can be seen in Figure 5.13.

5.1.5 Scenario 5 - Unlimited Possibilities

Sometimes users do not need complex behaviors. For example, they might just want a vehicle to follow a path of waypoints. This scenario demonstrates how this is possible. The scenario also shows that the methodology does not restrict flexibility for the inventive researcher.

Step 1 - Drawing State Diagram

This scenario requires one state diagram. It has one state and no transitions.

The state diagram in Figure 5.14 shows the single state for this scenario. Since the waypoints are unknown at design-time, all we know is that the vehicle must traverse them. This behavior is
at a higher level than those previously illustrated in which the behaviors explicitly included which waypoints to move toward.

**Step 2 - Choosing Libraries**

Only one action is required for this scenario.

- Actions:
  - Traverse Waypoints

  The trigger library needed no particular methods.

  The method `traverseWaypoints` was created. It requires no parameters and it ensures that the vehicle is always moving to the next waypoint.

**Step 3 - Converting the State Diagram to use Method Names**

The new state diagram can be seen in Figure 5.15.

**Step 4 - Create Text File**

The text file for UAV1 is shown in Listing 5.6. It was derived from the state diagram in Figure 5.15.
State 0 - traverseWaypoints

Figure 5.15: The state diagram from Figure 5.14 after conversion to method names

Listing 5.6: UAV1.uav - Text input file for UAV1

#0 traverseWaypoints

Figure 5.16: A screenshot of Scenario 5
Results

The scenario worked as expected. A screenshot can be seen in Figure 5.16. Other implementations are possible but this one was chosen because it allows for the functionality to be reused in the least complex manner.

Note that it would have been possible to make this scenario more complex by adding other behaviors. For example, it would be possible to use the target detected trigger as seen previously. In this way, a vehicle could traverse the waypoints indefinitely until it spotted a target.

5.1.6 Scenario 6 - An Aggregate Of Everything

A final scenario was created to demonstrate most of the currently implemented behaviors working together. Also the ability to detect a nearby vehicle is demonstrated. This scenario has two vehicles. The first UAV is tasked with traversing its waypoints while looking for targets. If a target is detected, the vehicle should move to the target and loiter until reinforcements arrive. If reinforcements arrive, it should return to base to refuel and then continue traversing waypoints where it left off. If at any time it runs low on fuel, it should return to base before continuing. The second vehicle should move back and forth between its base and one waypoint. If it spots a target while en-route to its waypoint, it should move to the target and loiter until it runs low on fuel. If it runs low on fuel at any time, it should return to base before continuing its mission.

Step 1 - Drawing State Diagrams

This scenario requires two state diagrams.

The state diagram in Figure 5.17 shows the states for UAV1. As in the previous scenario, the waypoints are unknown at design-time. All that is known is that the vehicle must traverse them. Figure 5.18 shows the state diagram for UAV2.
Figure 5.17: A state diagram for Scenario 6, UAV1
Figure 5.18: A state diagram for Scenario 6, UAV2
Step 2 - Choosing Libraries

The following triggers and actions were needed for this scenario.

- **Triggers**
  - Target Detected
  - Friendly UAV Detected
  - Low Fuel
  - Base Reached
  - Waypoint Reached

- **Actions:**
  - Traverse Waypoints
  - Move to Target and Loiter
  - Move to Base
  - Move to Waypoint

A new method named `vehicleSensed` had to be created to check for friendly vehicles within sensor range. This method requires no parameters. All other methods had been used previously, so there was no need to create them.

Step 3 - Converting the State Diagrams to use Method Names

The new state diagrams can be seen in Figures 5.19 and 5.20.

Step 4 - Create Text File

The text files for UAV1 and UAV2 are shown in Listings 5.7 and 5.8, respectively. They were derived from the state diagrams in Figures 5.19 and 5.20, respectively.

Results

The scenario worked as expected. A screenshot can be seen in Figure 5.21.
Figure 5.19: The state diagram from Figure 5.17 after conversion to method names

Listing 5.7: UAV1.uav - Text input file for UAV1

#0 traverseWaypoints
#1 moveToTarget
#2 moveToBase
*0>1 targetSensed
*1>2 vehicleSensed
*1>2 lowFuel
*2>0 baseReached
*0>2 lowFuel
Figure 5.20: The state diagram from Figure 5.18 after conversion to method names

Listing 5.8: UAV2.uav - Text input file for UAV2

```
#0 moveToWP:1
#1 moveToBase
#2 moveToTarget
*0>1 waypointReached:1
*1>0 baseReached
*0>1 lowFuel
*2>1 lowFuel
*0>2 targetSensed
```
Figure 5.21: A screenshot of Scenario 6
Contributions and Future Work

6.1 Summary

This research created a methodology for defining and implementing unmanned aerial vehicle agent behaviors. This methodology targets use by domain-experts with limited programming knowledge. The methodology lessens the burden of programming previously required for the implementation of agent behaviors. The methodology was described in a general form and then demonstrated in several proof-of-concept scenarios.

6.2 Contributions

In the past, little research had been done on how to define and implement UAV agent behaviors in a user-friendly, yet flexible manner. Behaviors often had to be defined by programmers who then had to alter model code. For MUAVES, this code was buried in the actual MUAVES simulation code. This research provides an initial methodology for domain experts. The methodology tries to shield the MUAVES user from programming so the user can instead focus on behavior definition and model results.
6.3 Future Work

There are several aspects of this research that could benefit from further investigation. Some of these are:

1. Define and develop a graphical model editor to ease the user’s modeling task. The graphical editor should manipulate the text input files. The research challenge is to represent the information in a logical manner for the user.

2. More methods should be added to trigger and action libraries to provide a more robust menu to the MUAVES user.

3. MUAVES is an interactive simulation so further research work could examine extensions of the methodology to facilitate dynamic behavioral changes.

4. Some of the behaviors in models like Brawler are derived from human-subject testing. Further extensions of this methodology should accommodate the definition of behaviors found on human-subject test results.
Listing A.1: Trigger method to test for closeness to a waypoint

/// <summary>
/// This method tests if the vehicle is within the vicinity of a particular waypoint.
/// </summary>
/// <param name="ent">the calling vehicle</param>
/// <param name="command">Index 1 should contain the waypoint number.</param>
/// <returns>the truth of the condition</returns>

public static bool waypointReached(Entity ent, string[] command)
{
    if (ent.vehicles.PresentLocation.getVicinity((Location)ent.vehicles.MyWaypoints[Convert.ToInt16(command[1])], 15))
    {
        return true;
    }
    else
    {
        return false;
    }
}

Listing A.2: Trigger method to test for closeness to the base

/// <summary>
/// This method tests if the vehicle is within the vicinity of its base.
/// </summary>
/// <param name="ent">the calling entity</param>
/// <param name="command">not used in current implementation</param>
/// <returns>the truth of the condition</returns>

public static bool baseReached(Entity ent, string[] command)
{
    if (ent.vehicles.PresentLocation.getVicinity(ent.baseLocation, 40))
    {
        return true;
    }
else
{
    return false;
}
}

Listing A.3: Trigger method to test for low fuel level

/// <summary>
/// This method tests if the vehicle is low on fuel.
/// </summary>
/// <param name="ent">the calling entity</param>
/// <param name="command">not used in the current implementation</param>
/// <returns>the truth of the condition</returns>
public static bool lowFuel(Entity ent, string[] command)
{
    double distBaseX = ((ent.baseLocation.X - ent.vehicles.PresentLocation.X) *
                        (ent.baseLocation.X - ent.vehicles.PresentLocation.X));
    double distBaseY = ((ent.baseLocation.Y - ent.vehicles.PresentLocation.Y) *
                        (ent.baseLocation.Y - ent.vehicles.PresentLocation.Y));
    int distBase = (int)Math.Sqrt(distBaseX + distBaseY);
    double fuelToBase = distBase * ent.vehicles.FuelConsumption;

    // very simple way of checking
    if (ent.vehicles.FuelRemaining < (1.1 * fuelToBase))
    {
        return true;
    }
    else
    {
        return false;
    }
}

Listing A.4: Trigger method to test for nearby targets

/// <summary>
/// This method tests if any targets are sensed.
/// </summary>
/// <param name="ent">the calling entity</param>
/// <param name="command">not used in the current implementation</param>
/// <returns>the truth of the condition</returns>
public static bool targetSensed(Entity ent, string[] command)
{
    ArrayList closeTargets = ent.vehicles.SensorValue.senseTargets();
if (closeTargets.Count > 0)
{
    return true;
}
else
{
    return false;
}
Action Method Code

Listing B.1: Method to initialize action library

```java
    /// <summary>
    /// The vehicle is assigned initial values such as waypoint list, base location, fuel consumption, starting fuel.
    /// must be included in any move control class
    /// </summary>
    /// <param name="thisEnt">the calling entity</param>
    public void init(Entity thisEnt)
    {
        ent = thisEnt;
        myWayPoints = ent.myPathOfWaypoints;
        fuelConsump = ent.vehicles.FuelConsumption;
        initFuel = ent.vehicles.FuelCapacity;
        baseLocation = ent.baseLocation;
        fuelRemaining = initFuel;
        currentLocation = baseLocation;
        wpnum = 1;
    }
```

Listing B.2: Method to get the current vehicle location

```java
    /// <summary>
    /// Gets the vehicle location.
    /// must be included in any move control class
    /// </summary>
    /// <returns>the current vehicle location</returns>
    public Location getLocation()
    {
        return currentLocation;
    }
```

Listing B.3: Method to get the current fuel level
### Listing B.4: Method for vehicle movement to a particular waypoint

```java
/// <summary>
/// Controls vehicle movement to a particular waypoint
/// </summary>
/// <param name="command">Index 1 should contain the waypoint number.</param>
/// <param name="spdel">a value calculated by the Entity class relating to
/// the speed of the simulation</param>
public void moveTo WP(String[] command, double spdel) {
    Location toMoveTo = (Location)myWayPoints(Convert.ToInt16(command[1]));
    global.logger.log("spdel:" + spdel);
    double delR = spdel / (3600000d);
    global.logger.log("delR:" + delR);
    moveToLocation(toMoveTo, delR);
}
```

### Listing B.5: Method for vehicle movement to its base

```java
/// <summary>
/// Controls vehicle movement to its base
/// </summary>
/// <param name="command">not used in this implementation</param>
/// <param name="spdel">a value calculated by the Entity class relating to
/// the speed of the simulation</param>
public void moveToBase(String[] command, double spdel) {
    double delR = spdel / (3600000d);
    global.logger.log("delR:" + delR);
    global.logger.log("BaseLocation:" + baseLocation);
    moveToLocation(baseLocation, delR);
}
```

### Listing B.6: Method for vehicle movement to a target

```java
/// <summary>
/// Gets the fuel remaining of the vehicle.
/// must be included in any move control class
/// </summary>
/// <returns>the current fuel level</returns>
public double getFuelRemaining() {
    return fuelRemaining;
}
```
### Method for waypoint traversal

```java
public void moveToTarget(String[] command, double spdEl)
{
    ArrayList closeTargets = ent.vehicles.SensorValue.senseTargets();
    double delR = spdEl / (3600000d);
    moveToLocation(((Target) closeTargets[0]).StartLocation, delR);
}
```

### Method for traversal of all waypoints and its base

```java
public void traverseWaypoints(String[] command, double spdEl)
{
    if (wpnum == ent.vehicles.MyWaypoints.Count)
    {
        if (!ent.vehicles.PresentLocation.getVicinity((Location)ent.vehicles.BaseLocation, 15))
        {
            moveToBase(command, spdEl);
        }
        else
        {
            wpnum = 1;
        }
    }
    else if (!ent.vehicles.PresentLocation.getVicinity((Location)ent.vehicles.MyWaypoints[wpnum], 15)
    {
        String[] com = new String[2];
        com[0] = "traverseWaypoints";
        com[1] = wpnum.ToString();
        moveToWP(com, spdEl);
    }
    else
    {
        wpnum++;
    }
}
```
Bibliography


Bent, N. E. (2003a, December). *The Brawler Air Combat Simulation Analyst Manual (Rev. 6.5.1)*. Vienna, VA, USA: L-3 Communications, GSI.

Bent, N. E. (2003b, December). *The Brawler Air Combat Simulation Management Summary Manual (Rev. 6.5.1)*. Vienna, VA, USA: L-3 Communications, GSI.


