12-1996

A Software Architecture for Computer Generated Forces in Complex Distributed Virtual Environments

Vincent B. Zurita

AFIT/GCS/ENG/96D-32

A Software Architecture For Computer Generated Forces In Complex Distributed Virtual Environments

THESIS

Vincent Brian Zurita, B.S.
Captain, USAF

AFIT/GCS/ENG/96D-32

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

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

AFIT/GCS/ENG/96D-32

A SOFTWARE ARCHITECTURE FOR COMPUTER GENERATED FORCES IN COMPLEX DISTRIBUTED VIRTUAL ENVIRONMENTS

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Systems

Vincent Brian Zurita, B.S.

Captain, USAF

December 1996

real soon. I'll let you sleep in, I'll help do the laundry and the shopping, you name it. Amy, in another week and a half I won't have to go to school anymore. We can play Lego on Saturday mornings and I promise I won't get up in the middle of it, grab my briefcase, and kiss you goodbye. Anthony, some day maybe you'll read this and understand why you went so many days in a row without seeing me at all. You've been a great sport through all of this, though, and I promise to hold you and hold you when we get to Virginia. I love you guys... like ca-ca-ca-crazy! Now let's go and have a great time in our new home.

Vincent Brian Zurita

Table of Contents

List of Figures

List of Tables

Abstract

Complex Distributed Virtual Environments (DVEs) present an outstanding opportunity for the Department of Defense to train geographically separated units within a single realistic threat environment with minimal logistical considerations or safety concerns. To increase the fidelity of these simulations, minimize cost, and thereby maximize the training potential, DVEs must be populated with a realistic number of Computer Generated Forces (CGFs). These are currently expensive to design and build due to a lack of standard CGF architectures. A solution to this problem is presented in the form of a CGF Architecture that is applicable to CGFs that model any weapon system. Mapping techniques are discussed that take the architecture from generic templates to weapon system-specific templates ready for implementation. An application based on this architecture, the Fuzzy Wingman, is discussed and its results are presented.

A Software Architecture for Computer Generated Forces in Complex Distributed Virtual Environments

1. Introduction

1.1. Motivation

The Department of Defense (DoD) possesses an extremely valuable training tool in distributed virtual environments (DVEs). DVEs allow armored units to support infantry in simulated tactical warfare, even though the two may be separated by thousands of miles. They enable pilots to fly simulated combined missions with warplanes of allied forces, without the cost or logistics involved in bringing the two together. DVEs permit surface ships to engage simulated enemy vessels in potentially deadly confrontations, without loss of life. Proximity without physical presence, formation without logistics, warfare without carnage; these are but some of the military training benefits being reaped today by distributed virtual environments.

Yet in spite of these capabilities, the DoD is far from realizing the full potential of DVEs. It is handicapped by a limited number of simulators suitable for interaction within these environments. If the intent of a simulation is to model a small-scale engagement, then the existing quantity of simulators may suffice. But the greatest benefits of DVEs are to be gained from large-scale, highly complex simulations in which a diverse range of many thousands of entities interact. Such complex simulations more accurately model the battles our forces are likely to encounter. Increasing the fidelity of a simulation enhances the level of training, which in turn yields a better-prepared force. How then, does one model such battles with a fraction of the number of required simulators?

The answer lies in computer generated forces (CGFs). These computer representations of military forces attempt to model sufficient human behavior to automatically execute a finite set of actions in response to their environment [IST96]. These actions may be as simple as following a road or flying to a waypoint, or as complex as cooperating with another friendly entity to defeat an enemy threat. CGFs of various complexity have been created for many different platforms: dismounted infantry, tanks, and other ground vehicles; rotary and fixed wing aircraft to include fighters and attack helicopters; and various types of surface ships and submarines. CGFs can serve to augment friendly forces or populate enemy formations. In either role, they bring to the virtual battlespace an added degree of fidelity by increasing the number of entities participating in a simulation.

Unfortunately, CGFs have yet to realize *their* full potential because they're plagued with two problems that are all too common with software today--they're expensive to develop and difficult to maintain. Herein lies the irony: We cannot afford an adequate supply of manned simulators, so we turn to CGFs to fill the void--yet CGFs turn out to be expensive as well.

Why is this so and how can the problem be remedied? CGFs are typically expensive and unmaintainable because they suffer from a lack of standardized architecture. There's no accepted, predefined way to go about creating one. This leaves the task of formulating an architecture to the developers of each new CGF. In essence, they're reinventing the wheel with each new entity they design. What is needed is a standardized architecture applicable to the development of all CGFs, regardless of their platform.

1.2. Purpose

It is the intent of this thesis to propose just such an architecture--one that is applicable to the development of *any* computer generated force to be operated in complex distributed virtual environments, independent of the platform being modeled.

1.3. Summary of Chapters

This proposed architecture is presented by first introducing, in Chapter Two, the topics that are relevant to the development of CGFs. Chapter Three identifies the requirements levied against this thesis project. Chapter Four reviews an existing CGF architecture and offers suggestions for improvement upon it. Chapter Five provides a mapping from this improved CGF architecture to the design of an actual CGF. Chapter Six describes the embodiment of this architecture in the Fuzzy Wingman F-15E CGF and relates the results of this implementation, while Chapter Seven concludes the thesis.

Additionally, several appendices provide further information on a range of topics. Appendix A contains a set of related Generic, Applied, and Complete CGF Module Design Templates. Appendix B contains source code for the Common Object Database. Appendix C presents the fuzzy logic term sets and rules used to fly point to point, while Appendix D contains output data from point to point runs.

2. Background

This chapter provides a brief overview of CGF architectures and directs the reader who is unfamiliar with these topics to key sources of information that provide coverage in much more detail than is possible in the scope of this work.

2.1. Distributed Virtual Environments

In the early 1980's, the Army realized the need to network its individual tank simulators to facilitate training on the platoon and company scale. The Defense Advanced Research Project Agency (DARPA) responded by developing a distributed architecture to support virtual simulations involving multiple manned simulators. The resulting system was SIMNET [THOR88], which still enjoys limited use today. SIMNET was successful in meeting its goal of aggregate training in a distributed virtual environment, but it has been limited primarily to the domain of armor and other ground vehicles.

As the Army was gaining its benefits from SIMNET, the DoD began anticipating the need for a more robust architecture to meet the demands of much more complex distributed simulations. These environments would be populated with warfighting entities of all domains--land, sea, air, even space. These entities would be based on manned simulators, CGFs, and instrumented live participants. To ensure interoperability among this diverse collection of highly interactive entities, a standardized simulation infrastructure was required. In 1989, the DoD addressed this concern with the initiation of the Distributed Interactive Simulation (DIS) program [IST94, IST96a]. DIS-based protocol data units (PDUs) have enabled the development of interoperable entities that communicate over the common Defense Simulation Internet (DSI) [IEEE93, KOUP96, STYT96].

In October 1995, the DoD issued its Modeling and Simulation Master Plan, which calls for the establishment of a common technical framework to, among other things, facilitate the interoperability of all types of models and simulations, and provide for the reuse of modeling and simulation components [DOD95]. Central to this framework is the High Level Architecture (HLA), which provides a functional

definition of the simulations and services provided by HLA federations, an interface specification, and object model templates [DMSO96]. HLA is intended as the successor to DIS.

2.2. Computer Generated Forces (CGFs)

CGFs are computer representation of forces to be employed in simulations. They attempt to model either human cognition or human behavior to the point that the forces being represented will engage in some actions automatically in response to their environment. Cognitive models, such as SOAR, attempt to model the human decision-making process as a means of effecting behavior [ROSE91]. On the other hand, behavioral models, such as ModSAF, seek to represent appropriate behavior within a domain without attempting to emulate the human decision-making process [CERA95].

An unfortunate property of many rapidly-developed CGFs is their relative lack of intelligent behavior. This property allows them to be easily discerned in the DVE from entities under the control of manned simulators, giving an otherwise unattainable advantage to the participants being trained. A solution to this problem is proposed by Santos et al that involves the use of fuzzy logic to deal with difficult situations and give a CGF a more human behavior [SANT96]. This is the approach used in this research project. Benslay [BENS96] gives a much deeper treatment of the role of fuzzy logic in this project.

2.3. Collaborative Research

This research was part of a collaborative effort with 1Lt James L. Benslay, Jr., who presents an in-depth treatment of the role of fuzzy logic in a CGF architecture in *A Domain Independent Knowledge Based Architecture for Computer Generated Forces* [BENS96]. Benslay's thesis should be considered an essential companion volume to this work. Indicative of the close degree of cooperative effort, Chapter Four of this thesis has been co-authored and is included in both theses for completeness.

3. Requirements

Every verifiable project must begin with a set of requirements. Those levied against this project are listed herein. Each requirement is accompanied by a description of its purpose and an indication of how it was satisfied.

Requirement	Purpose	Method of Satisfaction
1. Develop a CGF Architecture	To facilitate the design of CGFs of any weapon system	Modular Design Decomposition
2. Develop a Wingman CGF consisting of an SDE, TDE, CDE, DISManager, AeroModel, and FlightController	To validate Modular Design Decomposition of Control Module Types	Set of six concurrent C++ programs
3. Wingman must fly formation	To validate Modular Design Decomposition of Decision and Entity Propagation Module Types	FltCntrl class, FWAeroModel class
4. Wingman must fly point to point	To validate Modular Design Decomposition of Decision, Knowledge Base, and Entity Propagation Module Types	FltCntrl class, Basic Fighter Maneuver data files, FWAeroModel class
5. Wingman must perform DCA mission	To validate Modular Design Decomposition of Mission Module Type	DCAMission class
6. Wingman must employ BVR DCA tactics and weapons	To validate Modular Design Decomposition of Decision Process and Entity Propagation Module Types	fuzzy_tree class, Weapons functions

Table 3.1. Project requirements, their purposes, and means of satisfaction.

More information on the implementation of Requirement 1 can be found in Chapters Four and Five. The remaining requirements are elaborated upon in Chapter Six.

4. A General CGF Architecture

In order to effectively seek the solution of a problem, one must first thoroughly understand the problem. The purpose of this chapter is to explore the domain of the CGF architecture problem, examine a currently existing architecture, and to present a concurrent processing perspective of this architecture. This will be done by breaking the architecture down into its subcomponents, and further defining the functions of the essential subcomponents via semantic Areas of Concern. The concurrent processing perspective is useful for understanding the dynamics of the CGF architecture.

4.1. Define the Problem Domain

When considering the virtual environment, it is certainly plausible to envision a CGF filling the role of any of the entities within that environment. Regardless of the role of the entity, given that we have sufficient understanding of the entity's knowledge and its behavior in a specified environment, we should be able to create a CGF to fill that entity's role. If we assume for the moment that we do have an understanding of an entity's knowledge base and its intrinsic behaviors, the question then arises "How do we assemble this entity"? How would we combine our understanding of the entity's knowledge and behaviors into a software architecture? But more than this, does an architecture already exist that could be used so we wouldn't have to "reinvent the wheel"? If so, can it be adapted to suit this new entity, or has it been so tightly coupled to its current implementation that it won't meet our needs?

This is the essence of the architecture problem domain. Given that a need exists to build a CGF, is there some design architecture or methodology that can be applied to take the CGF from concept to implementation, regardless of the type of CGF that needs to be implemented? This is a more complex problem than may be at first realized. Consider the elements of Figure 4.1. Any CGF we consider will have a combination of these elements, but to identify these general types certainly doesn't appear complex. The complexity comes in when all the specific instances of these elements and their resulting data and control flows are considered.

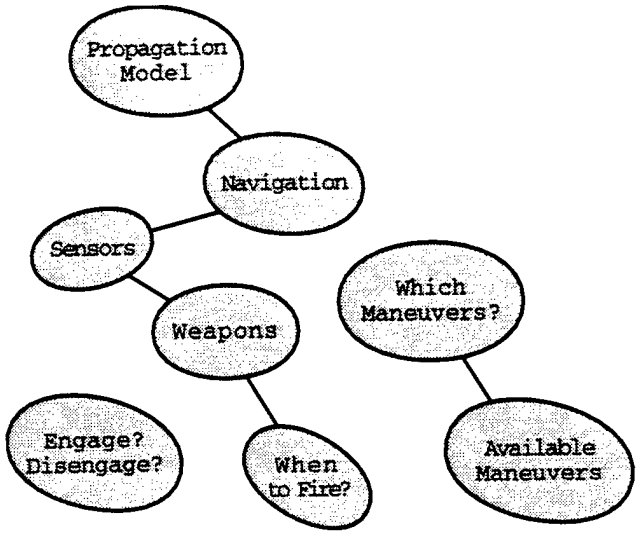


Figure 4.1. Generic elements of a CGF's domain.

Currently, it seems that whenever a simulation center implements a CGF, they do so from the ground up. What is needed is a single, domain independent solution to solve this problem.

4.2. An Existing Architecture

There is an existing model that takes the first step in mapping this problem domain. Santos, et al have devised a general architecture of CGF components [SANT96]. Figure 4.2 is their model for an adaptable CGF architecture. To summarize their work, a CGF is essentially comprised of two types of components, a Physical Dynamics Component (PDC) and an Active Decisions Component (ADC).

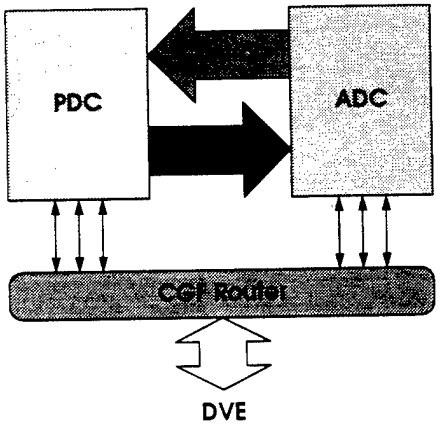


Figure 4.2. CGF architecture showing the Physical Dynamics Component, Active Decisions Component, and CGF Router.

The PDC is made up of the components necessary to model the CGF's physical makeup, such as entity propagation models, sensor models, weapons models, defensive elements models, etc. The PDC also contains those initialization parameters that are necessary to give the CGF a specific identity, such as performance specifications and limitations, as well as more human traits such as operator capabilities and constraints.

The ADC is composed of the components that use the information from the PDC to make decisions, and is broken down into three subcomponents: a Strategic Decision Engine (SDE), a Tactical Decision Engine (TDE), a Critical Decision Engine (CDE), and a Basic Control Module (BCM). Each of the decision engines comprises a different level of decision making process.

The SDE is concerned with high level functions such as understanding and implementing mission level goals, communicating with other players, interpreting the surrounding environment, and revising mission goals and subgoals.

The TDE's role is to manage the moment to moment operations of the entity. This entails receiving specific taskings from the SDE (subgoals) and implementing these according to the knowledge base of the particular entity. This will include activities such as determining which maneuvers to make in a given situation, determining when to employ ordnance, and when to implement other application specific elements such as electromagnetic counter measures.

The CDE is the survival instinct element of the virtual entity whose responsibility includes taking over control of the entity in emergency situations where termination of the entity is eminent. This is analogous to a person's instinctive reflexes transferred to the domain of the specific entity.

The BCM's role is to determine the proper propagation model inputs to perform the movements dictated by either the TDE or CDE. The BCM is considered a subcomponent of the ADC because it uses modeled behaviors to determine control inputs.

Interactions between the ADC subcomponents were defined as a number of finite state spaces and were maintained in a central communication structure.

Although not considered a component equal to the ADC or PDC, the CGF architecture has another important structure. This is the CGF Router (see Figure 4.2), and represents the interface between the Distributed Virtual Environment (DVE) and the two components, ADC and PDC.

The essence of this model lies in applying the concept of *separation of concerns* to the domain of CGFs. As defined in the practice of software engineering, applying a separation of concerns in a system design means separating the "how" from the "what" in the system. This is especially useful in creating an architecture for modular CGFs where we want to separate out an entity's decision making ability from its physical ability. It is also useful for scoping the entire decision making process down into manageable subcomponents. This concept of applying a separation of concerns is the foundation on which this research is based.

4.3 Essential Subcomponents

The CGF architecture was applied to the Intelligent Wingman as an integral part of this research. The idea was to build the Wingman completely around the ADC/PDC concept so that modules such as the aero model, sensor models, and behavioral components could be interchanged without disturbing the rest of the system. The first step in this process was to identify the principle subcomponents of the architecture that would be needed to begin constructing the Wingman. The following components were directly derived from the architecture model:

ADC Subcomponents
1. SDE
2. TDE
3. CDE
4. BCM

PDC Subcomponents
1. Propagation Model

CGF Router Subcomponents
1. DIS Interface

4.4 Semantic Areas of Concern

One of the primary purposes of Santos et al's architecture was to lay a common foundation from which a software engineer and a knowledge engineer could successfully model both the design and behavior of a CGF. Pursuant to this, the roles of the decision engines were described [SANT96]. However, as the decision engines were being developed for the Wingman, it was necessary to more clearly define the abstracted roles of the decision engines. To accomplish this, semantic Areas Of Concern (AOC) were used. An AOC is simply a question that must be answered in order to understand the role of that decision engine. The purpose of the AOCs is two fold: first, they help the knowledge engineer identify the kinds of knowledge and behaviors that need to be modeled at the different levels of decision making, and second, they help the software engineer organize the various processes into encapsulated modules that can be replaced with like modules. In the end, the AOCs help both types of engineers make a smooth transition from the generic architecture to an applied design.

The AOCs were established as a result of modeling the real world considerations that confront a present day war fighter. We placed ourselves in the role of a "generalized" war fighter, and used our expertise as military officers to determine the knowledge that a decision engine would need in order to accomplish its generalized purpose as described by Santos et al [SANT96]. For example, perhaps the single most fundamental question to any war fighter, regardless of service or rank, is determining one's mission. If a person or unit wants to know their function, the question is asked, "What is my mission?" Likewise, there are intuitive follow-on questions, such as "How do I go about accomplishing my mission?", and "What is my current task within the mission"? These questions are absolutely germane to both the decision engines and the role of a war fighter. In the process of enumerating the questions, they were then tailored to be domain independent, and were focused a bit more to be in line with the thinking of a CGF.

To make the AOCs as simple and direct as possible, they are written in a first person perspective from the CGF's point of view. They are designed so that by answering these questions in the context of a specific type of CGF within a specified domain, the bulk of the work for identifying the CGF's behaviors will be accomplished.

Note that these AOCs are not arranged in any intended chronological order, and that they may be tailored for a CGF type by adding new ones or by eliminating those that do not apply. The key questions are as follows:

1. SDE
- What's my mission?
- How do I go about accomplishing my mission?
- Am I able to accomplish my mission?
- What's my current task within the mission?
- How do I communicate with other entities?
- How do I perceive the outside world?
- How do I deal with a lack of appropriate information?
- What are the target types that I'm designed for?
- What are my responsibilities?

2. TDE
- What kinds of maneuvers am I capable of?
- What kinds of weapons do I know how to use?
- How do I choose which maneuver to apply in a given situation?
- How do I choose which type of weapon to employ in a given situation?
- How do I know when to employ ordnance?
- How do I know that the weapon was effective?
- How am I aware of my environment?
- How do I take the environment into account in decision making?
- What is my tasking within the mission?

3. CDE
- Am I in any immediate danger?
- If I were in danger, what maneuvers do I know how to do to keep safe?
- How would I decide on which maneuver to make?
- What kinds of countermeasures do I know how to use?
- How do I decide on when to use that device?

Defining the roles of the decision engines in this fashion reinforces two points made by Santos et al in their architecture. One, the decision engines are engaged in distinctly different levels of decision making. This distinction implies different cyclic requirements for each decision engine. Two, the decision engines rely on certain PDC components to make decisions.

A conclusion can be drawn from these two points in regards to program organization within the CGF architecture. The conclusion is that concurrent program processes should be centered around the ADC subcomponents. As for PDC subcomponents, they should also be concurrent program processes when cyclic or control requirements demand, or they can be included in the program process they are most closely tied to. An argument could be made to the contrary however, and state that each PDC subcomponent

should be a separate program process and not be included in the program processes primarily devoted to ADC subcomponents. This argument goes against the opinion of established software professionals such as Gomaa [GOMA93] who state that having too many tasks increases the system overhead and complexity unnecessarily. Hence, program process consolidation should be done whenever feasible.

4.5. Concurrent Process Perspective of CGF Architecture

As a result of the above conclusion, Figure 4.3 shows the ADC subcomponents embedded within program processes of the same name. Parallelograms are used to denote program processes as established by Gomaa [GOMA93]. The program processes are depicted as interlocking ADC/PDC components to indicate the close relationship between those subcomponents within that program process. The separate graphs within the PDC portions of the program processes represent separate PDC subcomponents. Note that in keeping with the architecture design, control and state information flow are indicated by the arrows going between the interlocking components.

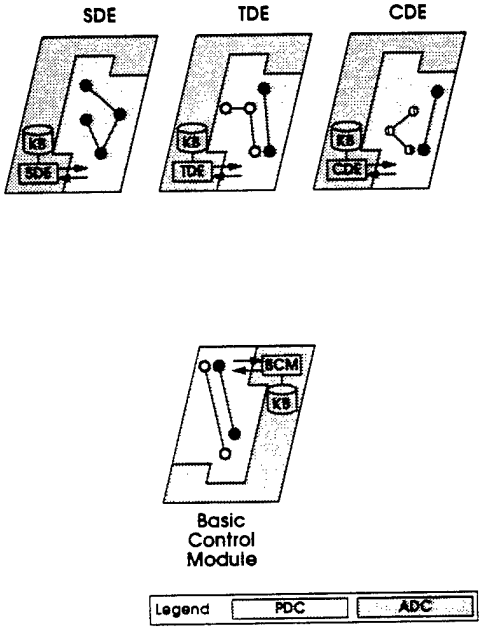


Figure 4.3. ADC and PDC subcomponents embedded within program processes.

Because the PDC and CGF Router subcomponents identified in Section 4.3 have a high importance in the CGF architecture, they are included with the program processes of Figure 4.3 to make Figure 4.4.

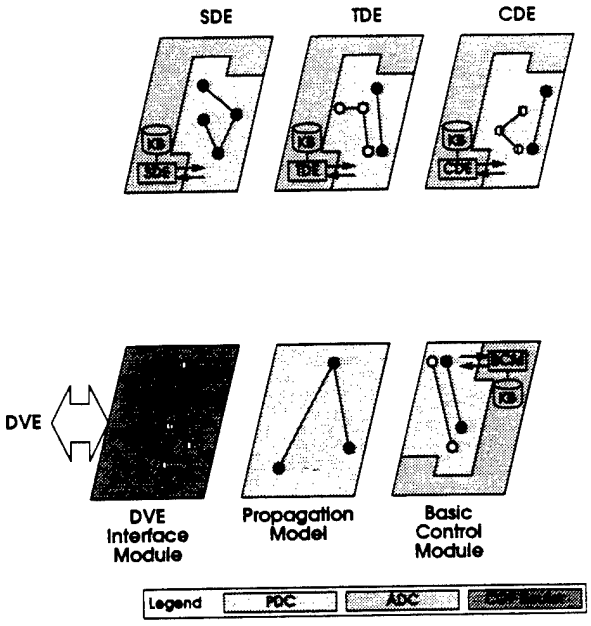


Figure 4.4. Essential PDC program processes in addition to the preceding ADC program processes.

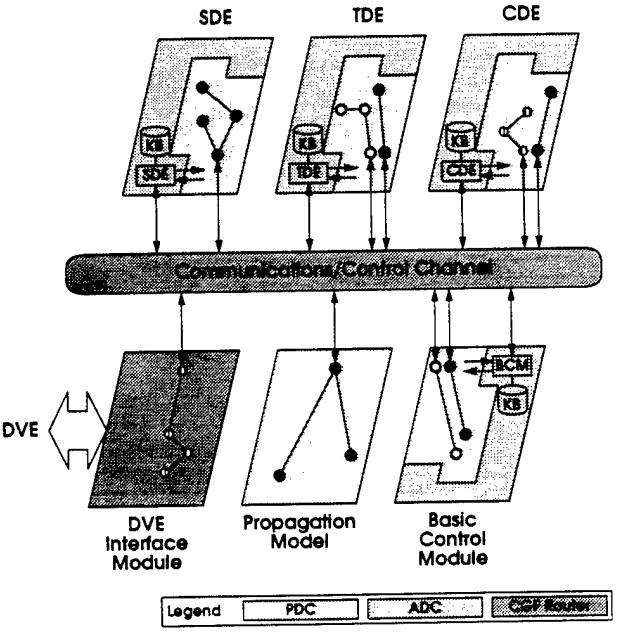


Figure 4.5. The completed concurrent process perspective of the CGF Architecture.

Consequently, the CGF architecture can be broken down into a core of six concurrent process. These are certainly not the only program processes that could be included here, they are simply the only ones identified at this time. Identifying additional program processes would be the result of deciding a particular PDC subcomponent needed to have its own process, or perhaps determining a need for a different level of decision making.

Figure 4.4 does not show any inter-process communication occurring. In order to allow asynchronous communications between the program processes, it is necessary to expand the definition of the CGF Router to include a "Communication/Control Channel". Figure 4.5 shows how this Communication/Control Channel would interconnect all subcomponents between all the program processes. The purpose of this construct is to give any subcomponent access to the information provided by any other subcomponent, regardless of program process or cyclic requirements.

4.6. Conclusion

By defining the decision engines according to semantic AOCs, a common foundation is established from which both software and knowledge engineers can begin the process of designing the software and behaviors of a CGF. By grouping ADC subcomponents with the PDC subcomponents they need most into the same program processes, a natural grouping of concurrent processes is established. By including the essential PDC and CGF Router subcomponents as separate program processes and combining these with the ADC program process, a concurrent process perspective of the CGF architecture is established that can be useful for understanding the dynamics of the architecture.

5. Modular Design Architecture

With the framework of this domain-independent CGF architecture laid out, a mapping methodology to a specific design is in order. This chapter details the steps of such a mapping that result in complete templates that have been applied to a CGF of a specific weapon system. Figure 5.1 captures this overall mapping process of the CGF Architecture, starting by mapping Areas of Concern to specific modules. These modules are then categorized according to similar functionality, yielding a generic template that describes at a high level the requirements that must be satisfied for this module--but it does so without referring to any specific weapon system. This template is then applied to a specific domain of weapon system, for example, fighter aircraft, or bombers. Finally, this applied template is tailored to address concerns specific to a particular weapon system--an F-15E or a B-2; this results in a complete template that is ready for coding.

5.1. Mapping Areas of Concern to Modules

The Areas of Concern presented in the previous chapter represent generic questions to be asked of each decision engine, regardless of the domain of the intended CGF. For instance, the SDE AOC "How do I perceive the outside world?" will be answered much differently for an M-1A1 Abrams than for an F-16D Fighting Falcon. While the M-1 might perceive its world through various types of optical sensors, the F-16 would likely rely on a combination of radar, visual, and infrared sensors. Regardless of the domain, however, each AOC may be mapped to a specific module. A CGF may contain such modules as a Weapon Propagation Module, a Terrain Information Module, and a Maneuver Decision Module. A module may be thought of as an abstract data type (ADT) as defined by Booch [BOOC83]. In this sense, modules contribute greatly to the various principles of software engineering. Particularly, they provide a solid level of abstraction by separating the functionality from the underlying implementation. Figure 5.2 depicts an appropriate mapping from each decision engine's AOCs to a core set of modules.

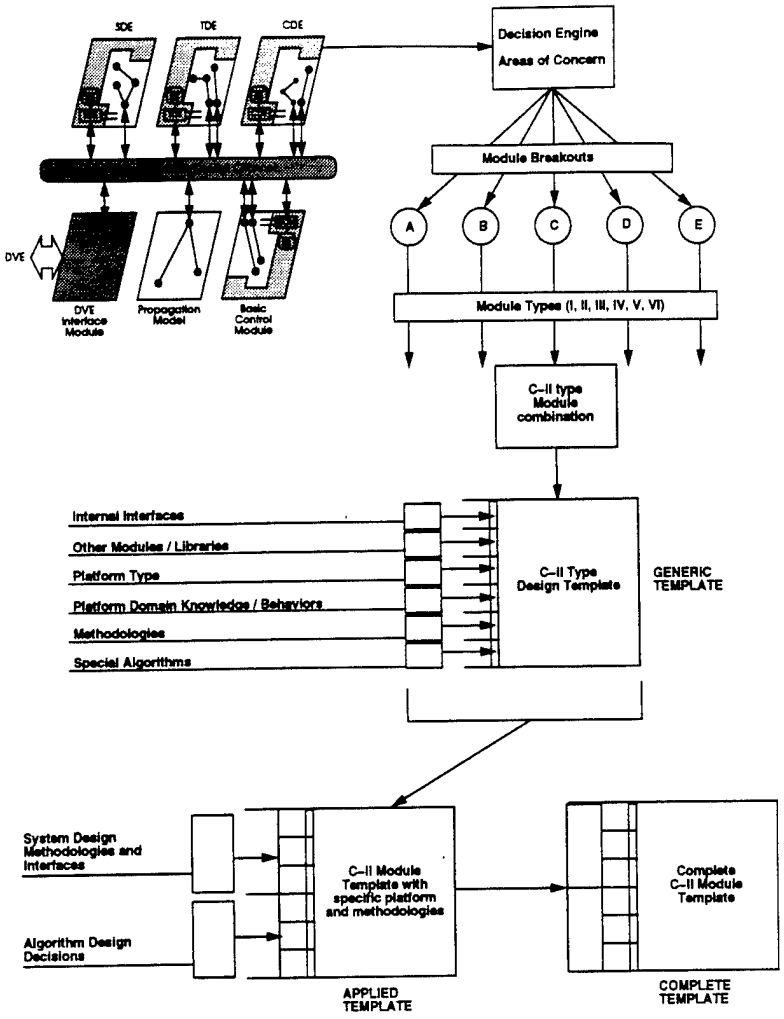


Figure 5.1. Overall visualization of the mapping of Areas of Concern to Complete CGF Design Modules.

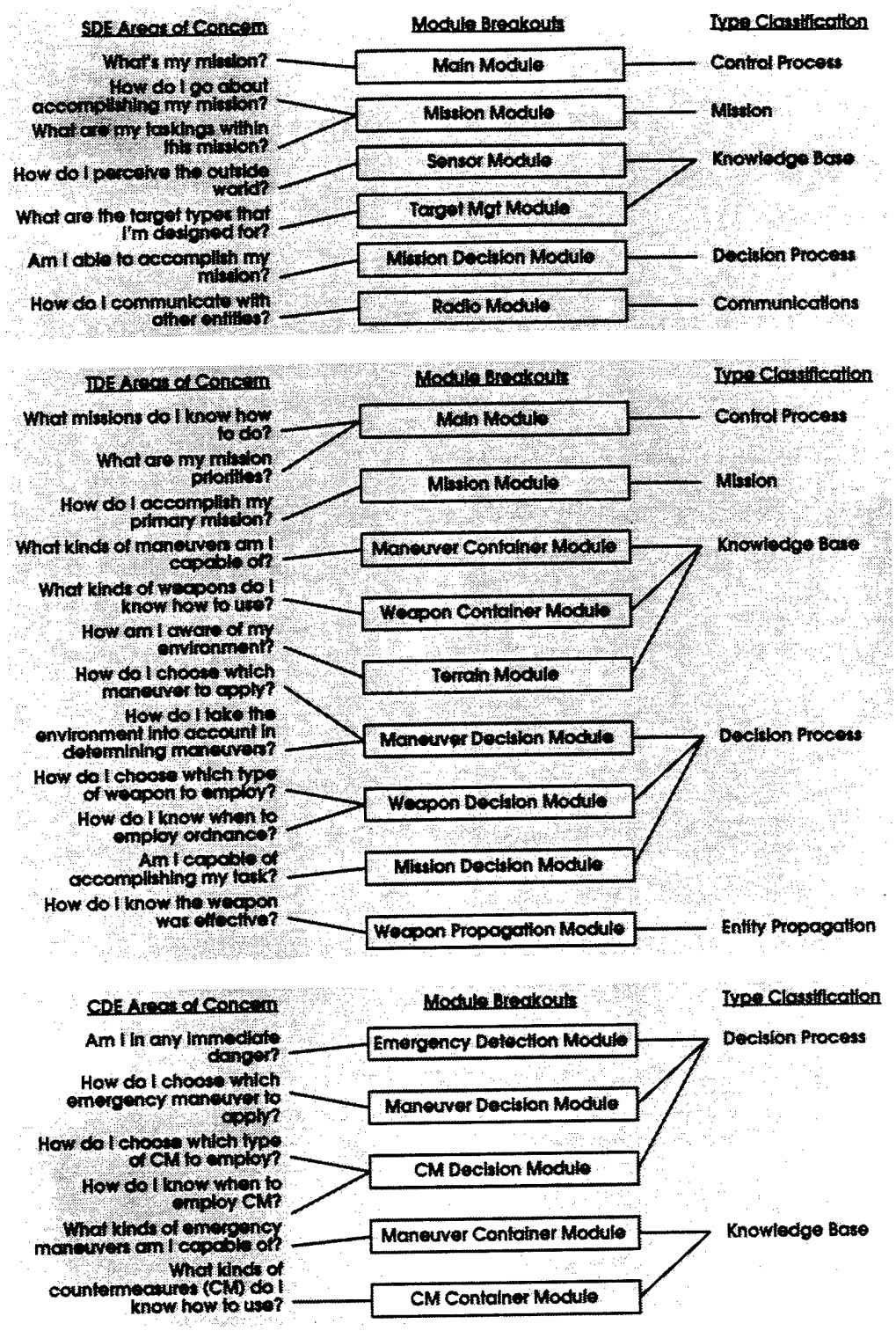


Figure 5.2. Mapping AOCs to modules and type classification of these modules.

5.2. Type Classification of Modules

Once the AOCs have been mapped into appropriate modules, the modules themselves must be categorized. Table 5.1 identifies a core set of module types against which all modules may be categorized. They are grouped as such to capture similar functionality.

Type	Classification	Criteria
I	Control Process	Maintains control over a set of related modules
II	Mission	Provides functionality and maintains flow of control necessary to effect a particular mission
III	Knowledge Base	Provides knowledge of a particular domain that may be used by a Decision Process Module Type
IV	Decision Process	Employs some form of Artificial Intelligence to make some decision
V	Entity Propagation	Provides functionality needed to move an entity through its medium
VI	Communications	Provides a means of transmitting information between entities

Table 5.1. Module types and their associative criteria.

In addition to showing the modules that were mapped to the AOCs of the three decision engines. Figure 5.2 shows the classification of the modules as well. Recall, from Section 3.2, the components comprising the PDC and ADC as described by Santos et al. Note, in Figure 5.2, the natural distribution of components of both the PDC and ADC throughout each of the decision engines. It is this natural blending based on the AOCs of each decision engine that led to the refined architecture proposed herein.

Now that modules have been identified and classified for each of the AOCs, the stage is set for creation of generic templates.

5.3. Generic Templates

Generic templates identify the type and functionality of a given module. As the name implies, they remain domain-independent--that is, they contain no verbiage that is specific to a particular category of weapon system. This means it is acceptable to refer to an entity's offensive weaponry, but it would be a mistake to mention air intercept missiles, since this template is intended to be equally applicable to a bomber as to a fighter. This leads to the goal of a generic template--to enable reuse for any weapon system that possesses the functionality being captured within it. Generic templates consist primarily of two parts.

The first part of a generic template identifies each AOC that has been mapped to the given module and states the requirements for satisfying them as well as a suggested design for the solution. Each requirement must support its parent AOC. The suggestions needn't be detailed; they'll be refined in later templates. For instance, in addressing the AOC *How do I choose which maneuver to apply?*, one of the requirements would be a knowledge of available maneuvers. The suggested design that corresponds might be something as simple as a Maneuver Container Module that contains some representation of the maneuvers this entity is capable of performing.

The second part of a generic template is a generic algorithm that provides one answer for all AOCs embodied within the particular module. Again, care must be taken to ensure the omission of any platform-specific verbiage. This algorithm may be very high-level pseudocode or, preferably, simple English statements. Figure 5.3 illustrates a Generic CGF Module Design Template for the Maneuver Decision Module identified as part of the TDE in Figure 5.2.

```
GENERIC CGF MODULE DESIGN TEMPLATE

Type:
        Decision Process Module Type
Function:
        Maneuver Decision Module
Intended User:  TDE
Specific Areas of Concern and their requirements:
        1.  How do I choose which maneuver to apply in a given situation?
            a) Required: A knowledge of available maneuvers.
               Suggested: A maneuver container module.
            b) Required: A knowledge of to whom the maneuver is to be applied
                         against and where they are located, or an area the
                         maneuver is to be performed in.
               Suggested: Method parameter.
            c) Required: A knowledge of the goal in applying the maneuver.
               Suggested: Assumed, method parameter, lookup, or module.
            d) Required: A knowledge of if we're reevaluating a pervious maneuver
                         decision that we're currently carrying out, or a first
                         time decision.
               Suggested: Method parameter.
            e) Required: A knowledge of my location.
               Suggested: Lookup.
            f) Required: A knowledge of an adversary and his known abilities.
               Suggested: An adversary information module.
            g) Required: A methodology or representation of applying the
                         known maneuvers in relation to another entity or area to
                         accomplish the goal.
               Suggested: An artificial intelligence algorithm/methodology,
                         potentially in a separate module.
            h) Required: A mechanism for relaying the decision to those
                         that need it.
               Suggested: Returned structure, reference pointer in argument
                         list, or TDE intracommunication methodology.
        2.  How do I take the environment into account in decision making?
            a) Required: A knowledge of the terrain.
               Suggested: A terrain container module.
            b) Required: A knowledge of other entities, where they
                         are at, what they are, and what they are doing (when
                         allowable).
               Suggested: An array of sensor modules.
            c) Required: A knowledge of atmospheric conditions.
               Suggested: An atmosphere module.

Generic Algorithm:
        Given the following:
            1) my location,
            2) an entity to which I'm supposed to move with or against and its
               location, OR an area that I'm to move within,
            3) an initial goal;
        Determine if the goal is obtainable;
        Determine how much time is available to make a decision;
        In the given time, choose the best maneuver which comes closest to
        accomplishing the goal that:
            1) doesn't violate terrain knowledge,
            2) takes into account atmospheric conditions,
            3) considers other entities and their potential adverse actions;
        Report the decision.
```

Figure 5.3. Generic CGF Module Design Template for the Maneuver Decision Module.

5.4. Applied Templates

Further refinement of a Module Design Template requires its application to the category of weapon system (platform) being modeled. The result of this is an Applied CGF Module Design Template.

Such a template is constructed by first identifying the intended platform, i.e. fighter aircraft, aerial refueler, tank, frigate, dismounted infantry, etc. Along with this, the domain of the platform is specified, i.e. land, sea, air, or space.

Depending on the *functionality* of the module, specific attributes of the applied design must be identified as well. In the example of a Maneuver Decision Module, it is not enough to specify merely that the module will be determining maneuvers for a fighter aircraft. Indeed, the mission must also be defined as a parameter, for the maneuvers appropriate for a fighter aircraft flying a defensive counter-air mission would differ from those of the same aircraft laden with heavier ordnance for a ground strike mission. Although it is impossible to enumerate all potential attributes that must be specified for all conceivable modules, one can expect these attributes to become apparent as a result of a good domain analysis.

There are attributes that must be enumerated for two of the *types* of modules. The first type of module with a standard attribute is the Knowledge Base Module Type. This has a Data Structure attribute that must be provided. This allows developers to match suitable Knowledge Base Module Types to particular Decision Module Types. Decision Process Module Types are the other type with a standard attribute. Any module that's been classified as a Decision Process Module Type has the sole purpose of inferencing over some knowledge base to come to some type of decision. This implies the use of some form of artificial intelligence--that form must be specified as an attribute here at this level of abstraction.

By making use of attributes in this manner, entire libraries of modules of similar functionality may be specified as variations of a theme. For instance, one could create a number of Applied CGF Module Design Templates for a Maneuver Decision Module for fighter aircraft; one using fuzzy logic, one using case-based reasoning, etc. This would facilitate experiments to measure behavioral differences between CGFs that differ only in the processes they use to implement the same functionality.

An important attribute that is applicable to *all* modules, regardless of functionality or type, is the Software Engineering Methodology. This attribute must be indicated for all Applied CGF Module Design Templates. Doing so allows CGFs to be constructed based on templates utilizing similar software engineering techniques. Figure 5.4 illustrates the attributes of a Maneuver Decision Module.

```
    APPLIED CGF MODULE DESIGN TEMPLATE

    Module Type:   Decision Process Module Type
    Function:      Maneuver Decision Module
    Platform:      Fighter Aircraft
    Intended User: TDE
    (NOTE:  Applied specifics for this type are preceded by an "-->").
--> Target Domain: Air
--> Mission Type:  Defensive Counter Air (DCA)
--> Artificial Intelligence Methodology:
                   Asynchronous, adversarial game tree with fuzzy spatial relationship
                   evaluators.
--> Software Engineering Methodology:
                   Modular Pseudocode Decomposition of Object Oriented Design
    Specific Areas of Concern, their requirements, and platform type specifics:
        1.  How do I choose which maneuver to apply in a given situation?
            a) Required: A knowledge of available maneuvers.
               Suggested: A maneuver container module.
-->                 Knowledge representation of maneuvers should represent movement in 3D
                    space, and should be identified in terms of the actual maneuvers they
                    represent.
            b) Required: A knowledge of to whom the maneuver is to be applied against and
                    where they are located, or an area the maneuver is to be performed in.
               Suggested: Method parameter.
-->                 It would be useful to have some sort of pointer to where this entity
                    resides in order to quickly access the necessary data such as location
                    and type.  Also, as a backup, give some other identifying information
                    about the entity that can verify the entity being pointed to.
    ...
```

Figure 5.4. Portion of Applied CGF Module Design Template for a Maneuver Decision Module showing attributes and enhanced suggested implementations.

Another move toward making Applied CGF Module Design Templates platform-specific is a more detailed description of the suggested implementation of each requirement of an AOC. This serves to further focus the solution on an implementation for a particular category of weapon system. Verbiage must not violate that level of abstraction. For instance, it would be reasonable to refer to a fighter aircraft's air to air missiles, but it would violate the level of abstraction to refer specifically to AIM-54 Phoenix missiles since these are only capable of being carried by F-14 Tomcats.

Finally, this focused solution is incorporated with all of the module's attributes and the previously designed generic algorithm to produce an applied algorithm for a specific platform category. Figure 5.5 illustrates how this applied algorithm lowers the level of abstraction from that of the generic algorithm by addressing computational concerns specific to the intended class of weapon system. Whereas a generic algorithm might have addressed the position of an entity, the applied algorithm for a tank may specify the latitude and longitude, while a fighter's algorithm would undoubtedly go even further and address altitude as well.

This combination of applied algorithm and attributes yields an Applied CGF Module Design Template that may utilized across an entire category of weapon system, enhancing reuse. A fighter aircraft defensive counter-air Maneuver Decision Module may be adapted just as easily for an F-14 as for an F-15 or F-16 CGF.

```
Applied Algorithm: Fighter Aircraft
1. - Construct the module object with the needed objects.
2. - Determine pilot performance parameters.
3. - Determine locations and orientations of aircraft and bandit.
4. - If target is offensive capable, evaluate geometry for a no-win situation.
5. - As a function of geometry, determine the maximum allowable decision making time.
6. - Determine if this decision process is the first applied to this target.  If not,
        then evaluate whether or not the actual movements are in keeping (by some
        fuzzified amount) with those decided on.  If either the aircraft or the
        bandit are sufficiently off the chosen maneuver, then scrap the current
        maneuver and reevaluate.  As long as the aircraft and bandit are performing
        reasonably as expected, then keep the score of the current maneuver as a
        comparison to consequent evaluations.
7. - Depending on the bandit type and the geometry involved, construct a game tree
        based on the known maneuvers for those posture states.
8. - Determine pruning heuristics.
9. - Based on positions, bandit type, velocities, maneuver selections, weapon
        choices, and terrain info, evaluate as many combinations as possible
        without exceeding the maximum allowable decision making time.  Retain the
        highest scoring combination.
10. -  If this is a follow-on maneuver evaluation for a bandit, then compare the
        results of this decision with the previous one.  Choose the better of the
        two and report that for implementation.
11. - If directed, record necessary data for a decision trace.
```

Figure 5.5. Applied Algorithm Portion of an Applied CGF Module Design Template for the Maneuver Decision Module as Applied to a Fighter Aircraft Domain.

5.5. Complete Templates

The final refinement to the system of templates is to map an Applied CGF Module Design Template from its platform to a specific weapon system to be run on a specific computer system, using a specified language and libraries. The end result is a Complete CGF Module Design Template, ready for coding.

The first step in this final mapping is to identify, as attributes, the intended weapon system (F-16D vs. F-15E, AH-1 vs. AH-64, etc.), the target computer system and operating system, programming language, and any specific software libraries that are to be used. Specifying these attributes allows one to build a repository of many varieties of Complete Templates based on differing combinations of attribute values. Additionally, existing attributes may be further clarified by giving references to source information. Figure

5.6 shows the various attributes of a Complete CGF Module Design Template for an F-15E DCA Maneuver Decision Module.

```
    COMPLETE CGF MODULE DESIGN TEMPLATE

    Module Type:        Decision Process Module Type
    Function:           Maneuver Decision Module
    Platform:           Fighter Aircraft
    Weapon System:      F-15E
    Intended User:      TDE
    Target Computer System:    SGI Onyx
    Target Operating System:   Irix 6.2
    Programming Language:              C++
    (NOTE:  Applied specifics for this type are preceded by an "-->").
    (NOTE:  Complete specifics for this type are preceded by an ">>>").
--> Mission Type:       Defensive Counter Air (DCA)
>>>             Combat Air Patrol as defined by Shaw.
--> Target Domain:      Air
>>>             As defined by IST-CR-95-14.
--> Artificial Intelligence Methodology:
                Asynchronous, adversarial game tree with fuzzy spatial relationship
                evaluators.
>>> Inferencing Library:
                FuzzyCLIPS.
--> Software Engineering Methodology:
                Modular Pseudocode Decomposition of Object Oriented Design
>>> Intracommunication Library:
                Common Object Data Base (CODB)
>>> DVE Interface Library:
                DIS Manager
```

Figure 5.6. Portion of a Complete CGF Module Design Template for an F-15E Defensive Counter-Air Maneuver Decision Module.

In addition to clarifying existing attributes, the suggested implementations of AOC requirements are refined to the point of being specific to the modules new attributes. Figure 5.7 shows how suggestions are tailored to the specific weapon system, while bearing in mind the intended hardware and software constraints.

Finally, all of the constraints identified so far are combined with the previously-created Applied Algorithm to produce a Complete Algorithm. This captures all details necessary to go straight to code. If properly constructed, the algorithm itself will serve well as in-line comments in the resulting code. Figure 5.8 presents the level of detail required in a Complete Algorithm. It is at this level that all restrictions on verbiage are finally removed, allowing the developer to address computational and representational concerns specific to an individual weapon system. It is here that addressing Phoenix missiles is not only acceptable, but expected when creating an F-14 CGF.

```
Specific Areas of Concern, their requirements, and platform type specifics:
    1.  How do I choose which maneuver to apply?
        a) Required: A knowledge of available maneuvers.
           Suggested: A maneuver container module.
-->        Knowledge representation of maneuvers should represent movement in 3D
           space, and should be identified in terms of the actual maneuvers they
           represent.
>>>        Maneuvers are modeled as a series of points in a matrix, sufficient in
           number to allow the maneuver to maintain a recognizable shape when scaled
           in 3D to meet the geometry requirements.  The maneuver container module
           should keep the methods necessary to read in and lookup the maneuvers and
           their points.
           DESIGN CONCERN: It is possible, that with all the maneuvers possible from a
           particular state from a particular task, that the maneuver module could be
           quite large, and the overhead for categorizing the structure could be
           overly complicated if not handled properly.
```

Figure 5.7. Portion of a Complete CGF Module Design Template for an F-15E Defensive Counter-Air Maneuver Decision Module showing a Refinement of Suggested Implementation.

```
Complete Algorithm: Fighter Aircraft: F-15E
Central Algorithm: Make_Decision
     - Pass in ptr to entity_identification_record that identifies specific target.
     - Pass in target index.
1. -  Determine pilot performance parameters.
     - Obtain Pilot Profile (PP) rating.
2. -  Determine locations and orientations of Wingman and target.
     - Get current XYZhpr of Wingman and target.
       Determine if Bandit ID as passed in is accurate, if not, search for
          appropriate.
       ID. If not found, pass back some sort of error flag.
3. -  If target is offensive capable, evaluate geometry for a no-win situation.  If
          Target's score is above a certain threshold, and the PP rating is high
          enough to realize this, limit tree construction to disengaging maneuvers
          and set goal to "disengage".
4. -  As a function of geometry, determine the maximum allowable decision making time.
     -  As a function of relative distance to Wingman and Bandit, determine longest
          allowable decision time for game tree decision making.
5. -  Determine if this decision process is the first applied to this target.  If not,
          then evaluate whether or not the actual movements are in keeping (by some
          fuzzified amount) with those decided on. If either the aircraft or the
          bandit are sufficiently off the chosen maneuver, then scrap the current
          maneuver and reevaluate.  As long as the aircraft and bandit are performing
          reasonably as expected, then keep the score of the current maneuver as a
          comparison to consequent evaluations.
     - Passed in as a parameter - decided by the calling method.
     -  If following up on a previous decision, evaluate geometry based on actual
          movements.  Retrieve previous XYZhpr of both aircraft, the maneuver
          decision made, and resulting score.  Place both aircraft in the Wingman
          centered coordinate system, but offset each aircraft's XYZhpr by the
          changes observed from the previous iterations.  Determine how far off the
          maneuver path both aircraft have drifted.  Fuzzify this value.  As a
          function of PP, determine if the algorithm can detect a discernible
          difference from either aircraft's actual position from the previously
          projected position.  If so - then scrap the current maneuver and reevaluate
          completely.  Otherwise, consider the current maneuver score as a baseline
          to compare all new maneuvers against.
```

Figure 5.8. Portion of a Complete Algorithm for an F-15E Defensive Counter-Air Maneuver Decision Module.

5.6. Conclusion

The methodology presented in this chapter maps the individual Areas of Concern of each Decision Engine into appropriate modules and then incrementally develops those modules into weapon system-specific designs, ready to be coded into solutions. This CGF Architecture provides the framework within which CGFs of any domain may be built, reusing designs and implementations from related CGFs that were previously developed, and in so doing, can help lower the cost of CGF development.

6. Implementation and Results

This chapter presents the embodiment of the CGF architecture as the Fuzzy Wingman, a fighter aircraft CGF employing fuzzy logic for its inferencing processes. The Wingman can currently be configured to fly as either an F-16D Fighting Falcon or an F-15E Eagle by simply initializing the propagation model to mimic the appropriate aircraft. It is also able to carry out F-15E defensive counter-air missions. Details are presented of the three Decision Engines, as well as the other major components that make up the Fuzzy Wingman CGF: the AeroModel Aircraft Propagation Module; the FlightController Module, which provides control inputs to the AeroModel; the DISManager, AFIT's interface to DIS-based DVEs; and the Common Object Database, which serves as the communication/command channel for all of these individual processes. Other support modules will be detailed that play an important role in the Wingman project as well, such as the Round Earth Utilities Module, which provide the coordinate conversion functions necessary to interact in a DIS DVE; the Terrain Module; the Maneuver Container Module; the Weapon Module; and the Sensor Module. For specific information regarding the role of fuzzy logic in this project, refer to Benslay's thesis [BENS96], which covers this topic in great detail.

Unless otherwise noted, all programs were written in C++ and compiled and executed on Silicon Graphics workstations using SGI's own compiler, CC, and running under the IRIX 6.2 operating system. The configurations of the various machines used in the implementation of the Wingman are enumerated in Table 6.1. Makefiles were used extensively to facilitate the tasks of compiling the large number of dependent modules.

Workstation	Model	Number of CPUs	CPU Speed	RAM
leonardo	Onyx	4	100 MHz	192 MB
rembrandt	Onyx	2	100 MHz	192 MB
donatello	Indigo^2	1	250 MHz	192 MB
monet	Indigo^2	1	250 MHz	192 MB
dali	Indigo^2	1	250 MHz	192 MB

Table 6.1. Configuration of hardware used in Fuzzy Wingman implementation.

Figure 6.1 provides an object diagram of the Fuzzy Wingman showing the distinct processes that communicate with each other via the CODB.

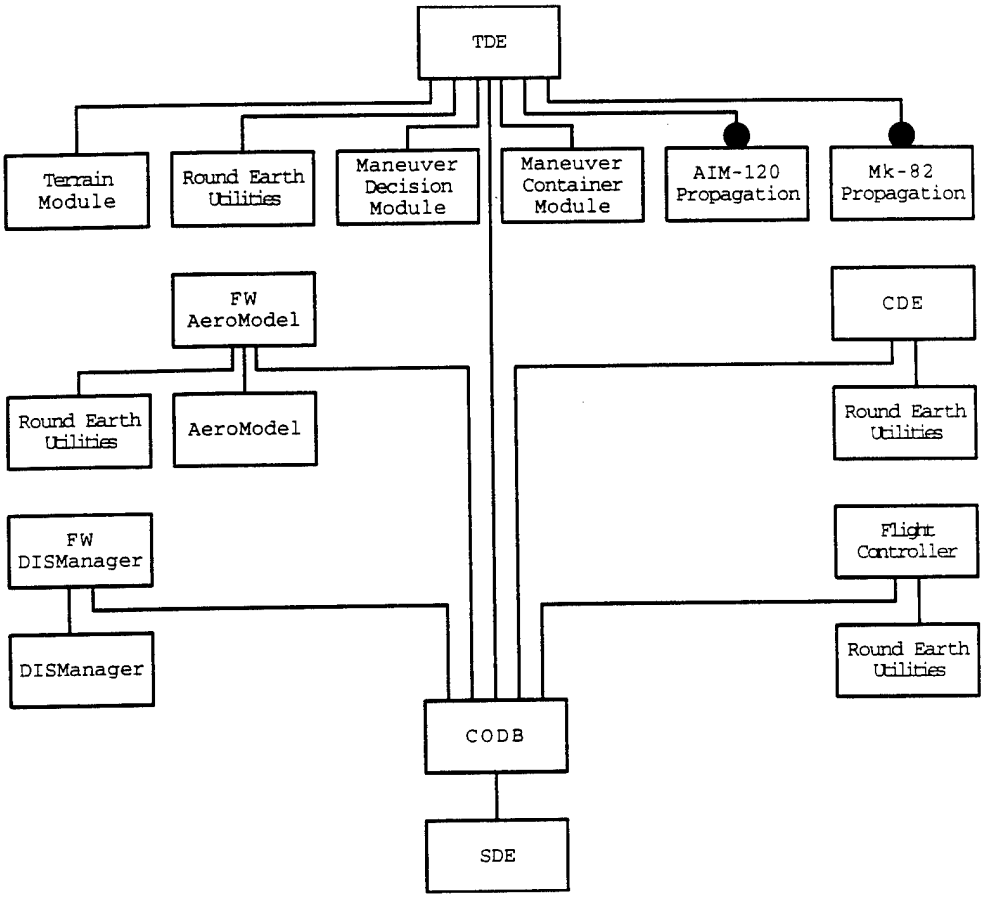


Figure 6.1. Object diagram of the Fuzzy Wingman.

6.1. Common Object Database

The concurrent nature of the CGF Architecture demands a communications/command infrastructure that is able to bridge the gap between separate processes. The Common Object Database (CODB) fulfills this requirement [STYT97]. The CODB is a C++ template class that uses shared memory techniques to pass information between unrelated processes. The CODB employed by the Wingman includes improvements to the standard version. The improvements focus on the relationship of the processes using the CODB. The standard version of CODB enabled communications only between closely related processes; that is, processes that have been spawned by a common parent process. When a process

is spawned using the C++ `sproc()` function with the `PR_SADDR` attribute, it shares all virtual space (shared memory, mapped files, data space) with its parent process [SILI91]. In this manner, the single CODB object was implicitly shared in memory among all related processes. The newer version of the CODB built for and incorporated into the Wingman does not rely on processes being closely related. Instead, it builds a shared memory arena using IRIX inter-process communications (IPC) functions [SILI91]. This enables processes that are completely independent (i.e. launched separately from the command prompt) to access a single CODB by explicitly joining the shared arena in which the CODB resides. This "heavy duty" CODB has been tested on both sproc'd and completely independent processes, and successfully transmits correct information in both cases.

The CODB accomplishes its task of directing information flow between multiple processes through the use of ten public and three private methods. With these methods, it creates a shared memory arena and places in it an array of double buffers. There exists one set of `doublebuffers` and a `readbuffer` and `writebuffer` pointer for each structure type recognized by the CODB (see dbtypes.h in Appendix B). At any given time, one of these buffers (pointed to by `readbuffer`) contains the most recently accessible information and is available to any number of concurrent readers, while the other buffer (pointed to by `writebuffer`) is available to only one process at a time for writing. When the writing process relinquishes control of the write buffer, the pointers to the buffers are swapped, effectively turning the write buffer into the read buffer, and vice versa. This dual buffer swapping is the key to maximizing accessibility to data, even while fresh data is being written. Figure 6.2 illustrates the array that comprises the internal structure of the CODB.

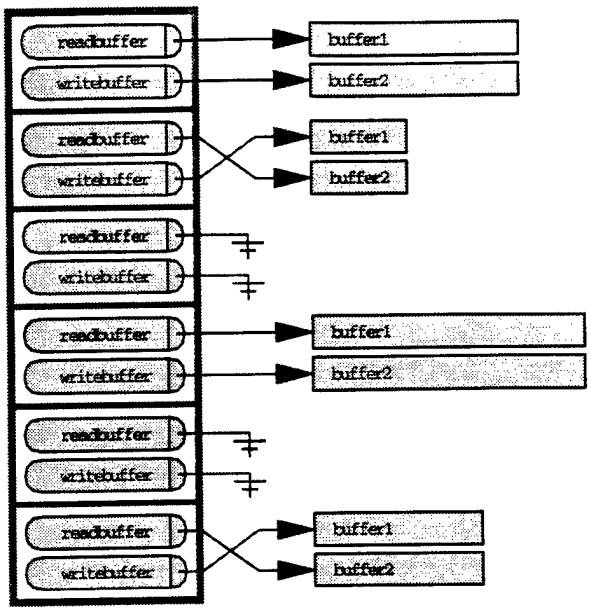


Figure 6.2. The `DoubleBuffer` array--the heart of the CODB, shown referencing four `readbuffer/writebuffer` pairs that have been allocated in shared memory.

The shared memory arena and its contents are initialized by the `CommonObjectDB` constructor and its calls to `initCommonObjectDB` and `initDoubleBuffer`. These three methods allocate 1 MB of RAM for the arena at a base virtual address of `0x20000000` (512MB). It is important that none of the CGF processes attempt any other memory operations in this space to avoid corrupting the CODB data with a memory collision. The initialization process allows 50 sets of `DoubleBuffers` and access by up to 16 distinct processes. All of these values may be modified as needed. They are defined in `dbtypes.h`.

In order to either read or write data to a particular data structure, that structure must first be registered with the CODB. This is done by instantiating a CODB object with the desired structure as the template type and indicating its associated `StructureType` enumeration. The `StructureType` value simply serves as an index into the `DoubleBuffer` array, while the structure that completes the template is used solely to obtain the amount of memory that needs to be allocated for the `DoubleBuffers` themselves. In other words, there is no mandated correlation between the template structure and the `StructureType` value. When registering a new structure, the CODB first checks to see if a `DoubleBuffer` has already been

established at the array position indicated by the new structure's associated `StructureType` value. If so, the new structure is ignored and a warning message is returned to the calling process. If not, sufficient memory is allocated in the shared arena for the new `DoubleBuffer`, and its associated locks and pointers are initialized.

While it is acceptable to attempt to register a structure twice, it is fatal to attempt to read or write a structure that hasn't been registered. If a process attempts this operation, CODB will issue an error message to the offending process and terminate it, leaving other processes unaffected. To avoid this problem, any process can first check to see if another process has registered a structure by calling the `Registered` method before attempting to read or write that structure.

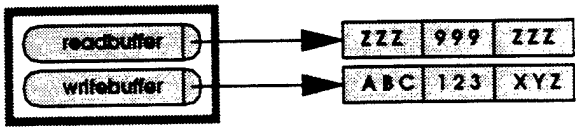
To facilitate the shared nature of the data stored in the CODB, all read and write operations involve a pair of method calls, one to begin the operation, and one to end it. It is absolutely imperative that these operations be performed in pairs. `BeginRead` locks out the aforementioned swapping process until the last reader of the particular structure issues its `EndRead`. If any reader process fails to match up an `EndRead` to each `BeginRead`, the process writing to that same buffer will become deadlocked, and all other processes that read that structure will receive the same stale data over and over again. `BeginWrite`, `BeginReadWrite`, and `BeginMagic` all lock out subsequent writers from accessing the indicated buffer until the corresponding `EndWrite`, `EndReadWrite`, or `EndMagic` method, respectively, is called. If any writer process fails to match up an `EndWrite`, `EndReadWrite`, or `EndMagic` to each `BeginWrite`, `BeginReadWrite`, and `BeginMagic`, not only will that same process become deadlocked, but so too will all other processes attempting to write to that same buffer, and all other processes that read that structure will receive the same stale data over and over again.

The functionality of the `BeginRead/EndRead` pair is straightforward. `BeginRead` indicates to the CODB that a process is attempting to read the `readbuffer` residing in the `DoubleBuffer` array at the slot indicated by the `StructureType` value. `EndRead` indicates to the CODB that the process is finished reading the same `readbuffer`.

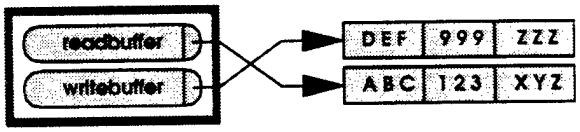
BeginWrite/EndWrite are just as straightforward. BeginWrite indicates to the CODB that a process is attempting to write to the writebuffer residing in the DoubleBuffer array at the slot indicated by the StructureType value. EndWrite indicates to the CODB that the process is finished writing to the same writebuffer. Because BeginWrite assumes the entire structure will be written to, it doesn't bother copying the current contents of the readbuffer into the writebuffer before handing over the stale contents of the writebuffer to the calling process. As long as the entire structure is written to, the stale contents will be completely overwritten by the new contents with no problems. However, if a process calls BeginWrite and then only updates a portion of the data in the structure, the remaining data items in the structure risk being corrupted by previous data.

Figure 6.3 illustrates this phenomenon. Initially, the writer process calls BeginWrite, then writes the entire Simple structure from its local copy into the CODB. After it calls EndWrite, the readbuffer and writebuffer pointers are swapped enabling all reading processes to access the correct data. The writer then calls BeginWrite again, but because only the contents of str1 have changed, it only writes that one value to the CODB before calling EndWrite. It is at this point that the problem of data persistence occurs. Any reader processes that access Simple_Struct in the CODB will unknowingly read stale data, indeed in this case the values of num and str2 have never even been initialized. This problem is remedied by the inclusion of the BeginReadWrite/EndReadWrite pair of methods.

```
struct Simple_Struct{
  char  str1(3) ;
  int   num ;
  char  str2(3) ;
} ; //end struct

Simple_Struct LocalSimple ;
LocalSimple.str1 = "ABC";
LocalSimple.num = 123;
LocalSimple.str2 = "XYZ";

Simple_Struct *GlobalSimple;

GlobalSimple = CODB->BeginWrite(SimpleStruct);
*GlobalStruct = LocalStruct;
```




```
CODB->EndWrite(SimpleStruct);

LocalSimple.str1 = "DEF";
GlobalSimple = CODB->BeginWrite(SimpleStruct);
GlobalStruct->str1 = LocalStruct.str1;
```




```
CODB->EndWrite(SimpleStruct);
```

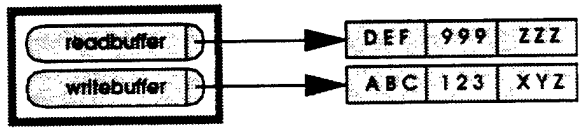


Figure 6.3. The stale data persistence phenomenon. If the contents of readbuffer aren't copied over to writebuffer before writing only portions of the structure, then stale data will persist in the unwritten portions.

The `BeginReadWrite/EndReadWrite` and `BeginMagic/EndMagic` pairs are identical to each other. `BeginMagic/EndMagic` were names given to these methods in early stages of development of the CODB. They've since been replaced by the more appropriately-named `BeginReadWrite/EndReadWrite`, but they remain for backwards-compatibility. Their function is identical to that of the `BeginWrite/EndWrite` pair, with one major exception. `BeginReadWrite` first copies the contents of the current `readbuffer` into the `writebuffer` before handing over the `writebuffer` to the calling process. This enables the writer process to update only portions of the structure and still maintain integrity of the remaining data items within that

structure. The argument can be made of eliminating the `BeginReadWrite/EndReadWrite` and `BeginMagic/EndMagic` pairs and simply having `BeginWrite` perform this initial copy function. If all structures maintained within the CODB were very small, then performance wouldn't suffer as a result of the memcopy operation that performs this task. But many structures registered within the Wingman system consume large amounts of memory; frivolously copying such large memory segments can take a noticeable toll on CPU performance.

6.2. DISManager

The DISManager is the Wingman's implementation of a generic DVE Interface Module. It was created by Mr. Steven Sheasby of Distributed Simulation Technologies, Inc for use in AFIT DVEs. It uses the DIS 2.0.3 protocol to exchange Protocol Data Units (PDUs) between the Wingman and the DVE [IST95, IST96b]. It is designed to use the CODB to distribute incoming PDU information between processes.

The DISManager consists of two closely-related processes--the parent process that is responsible for sending PDUs from the local application (i.e. the Wingman) to the DVE, and its child process that receives PDUs from the DVE and places their information into the CODB for any other process to access. The DISManager provides method calls for sending various PDUs. For most applications, this design is sufficient for a proper interface to a DVE. But because the Wingman consists of several unrelated processes running concurrently, a problem arises. Namely, how does one process make a method call to a completely unrelated process? The solution to this lies in a subclass of the DISManager that uses the CODB as a means of passing control between processes.

This subclass is the FWDISManager. It registers with the CODB a DISManager control structure that contains booleans that serve as switches for the various PDUs that can be sent; the structure can best be thought of as a matrix containing rows of weapons and columns of PDU types as shown in Table 6.2. This control structure is then used by the weapons propagation process that, for instance, launches an AMRAAM, and needs a corresponding Fire PDU to be broadcast to the DVE. This weapon propagation process indicates the need for such a PDU by setting the `FirePDU` field to `TRUE` for that particular

AMRAAM in the CODB. On the other end, the FWDISManager process reads this switch setting from the CODB and makes the appropriate call to the DISManager's `broadcast_fire_pdu` method.

	Fire PDU	Detonate PDU
Mk-82	FALSE	FALSE
Mk-82	FALSE	FALSE
Mk-82	FALSE	FALSE
Mk-82	FALSE	FALSE
Mk-82	FALSE	FALSE
Mk-82	FALSE	FALSE
AIM-120	TRUE	FALSE
AIM-120	FALSE	FALSE
AIM-9	FALSE	FALSE
AIM-9	FALSE	FALSE

Table 6.2. Matrix representation of the DISManager control structure showing an AIM-120 AMRAAM that has been launched and needs a corresponding Fire PDU to be broadcast.

The FWDISManager subclass is one of two instances in the Wingman system of using the CODB as a means of control. The other example occurs in the CDE and is presented in Section 6.11. All other processes use the CODB strictly as a means of communicating data.

6.3. AeroModel Module

The AeroModel Module propagates the Wingman aircraft through the air within the DVE. It was originally written in the C programming language for Wright Laboratory's Flight Simulation Facility (WL/FIGD). Capt Terry A. Adams converted it into a C++ class for use in the Rapidly Reconfigurable Virtual Cockpit [ADAM96]. This same version is used in the Wingman. The AeroModel has the ability to model the flight dynamics of F-15, F-16, F-18, F-5E, and A-10 aircraft. Since this model was used as is, it was obviously not subject to development according to the CGF Architecture described herein.

It is important to note several points of concern with the AeroModel. First, the AeroModel propagates the aircraft entity on a 20 Hz cycle through a North East Down (NED) flat earth coordinate system. This means the positive X-axis corresponds to North, while the positive Y- and Z-axes are associated with East and Down, respectively. This is a right-hand coordinate system. The linear units associated with the NED system are feet. Additionally, the heading, pitch, and roll of the aircraft are

identified by their aeronautical engineering notation, namely psi, theta, and phi, respectively. Table 6.3 describes how these orientation angles are measured.

Orientation	Measurement Range	Description
+ Heading	0° to 180°	Positive rotation around +Z-axis
- Heading	0° to -180°	Negative rotation around +Z-axis
+ Pitch	0° to 90°	Positive rotation around +Y-axis
- Pitch	0° to -90°	Negative rotation around +Y-axis
+ Roll	0° to 180°	Positive rotation around +X-axis
- Roll	0° to -180°	Negative rotation around +X-axis

Table 6.3. Measurement methods for heading, pitch, and roll.

Finally, the unitless measurements of the throttle, stick, rudder, and airbrake controls are outlined in Table 6.4. These are the basic inputs that control the cycle to cycle propagation of the aircraft.

Control Device	Measurement Range	Description
Throttle	0.0 to 1.5	0.0 to 0.99 is mil power, 1.0 to 1.5 is afterburner
Stick Lat	-1.0 to 1.0	0.0 to -1.0 is stick left (-roll) 0.0 to +1.0 is stick right (+roll)
Stick Long	-1.0 to 1.0	0.0 to -1.0 is stick forward (-pitch) 0.0 to +1.0 is stick back (+pitch)
Rudder	-1.0 to 1.0	0.0 to -1.0 is rudder left (-hdg) 0.0 to +1.0 is rudder right (+hdg)
Air Brake	1, 0, or -1	1 = extend, 0 = hold, -1 = retract

Table 6.4. Throttle, stick, rudder, and airbrake measurements.

Figure 6.4 shows the Aircraft Body Coordinate System (ABCS), which has the same configuration as the NED system, but is locked to the airframe such that the positive X-axis projects through the nose of the aircraft, positive Y goes out the right wing, and positive Z drops out the bottom. It is important to be intimately familiar with these measurements in order to construct proper solutions to aircraft control problems (i.e. flying formation or flying point to point).

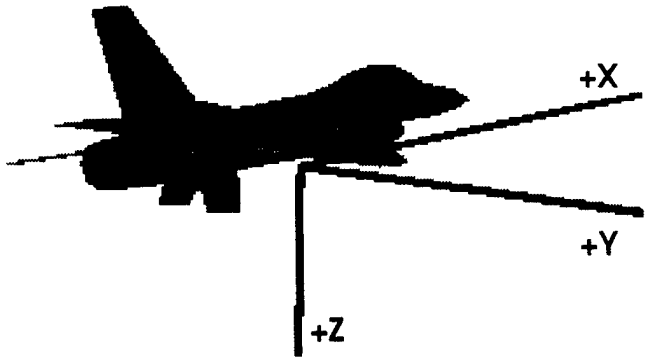


Figure 6.4. The Aircraft Body Coordinate System.

The AeroModel has proven itself to be an excellent aircraft propagation model for the Wingman. Two F-16 pilots flew the Virtual Cockpit and praised the fidelity of the AeroModel [ADAM96].

6.4. Round Earth Utilities Module

The Round Earth Utilities (REU) Module provides functions that translate an entity's position and orientation between the AeroModel's NED Coordinate System and the World Coordinate System (WCS) used in DIS applications. The NED Coordinate System is described in Section 6.3. The WCS is a right-handed geocentric Cartesian coordinate system that has its origin at the center of the earth. The positive X-axis passes through the Prime Meridian at the Equator, while the positive Y-axis passes through the Equator at 90° East longitude and the positive Z-axis passes through the North Pole [IST94a]. Its unit of linear measurement is the meter; it measures angles in radians. Figure 6.5 illustrates this coordinate system.

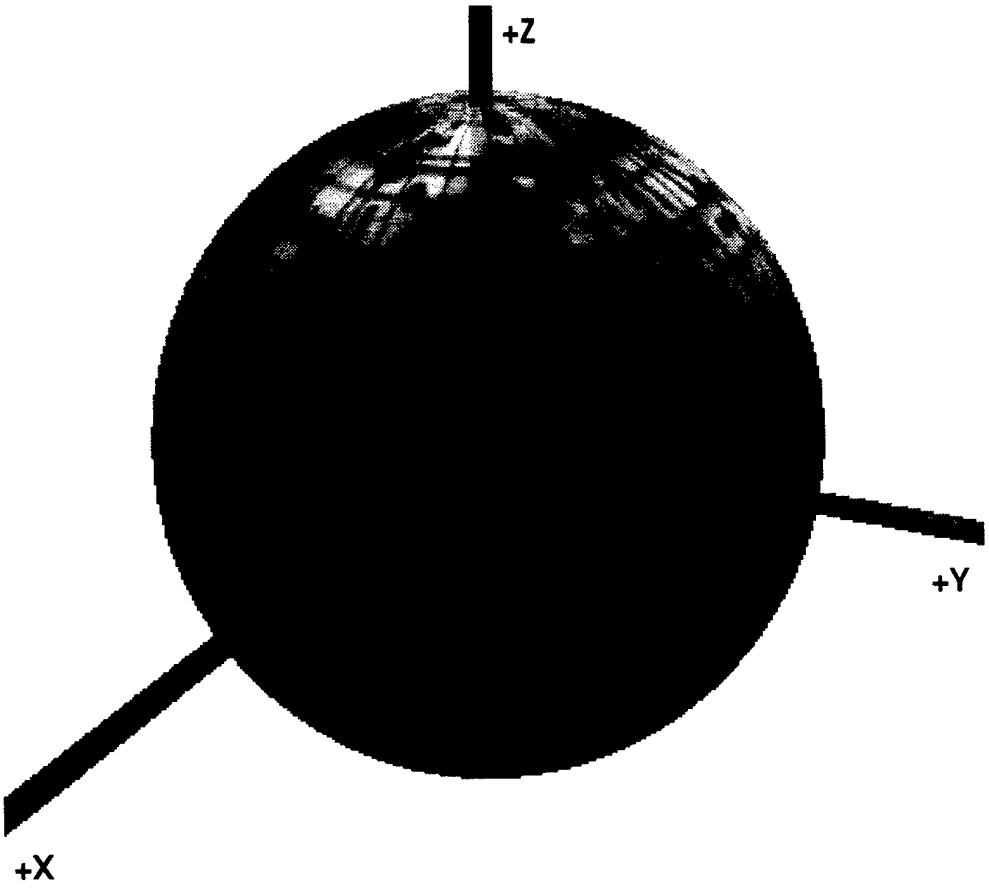


Figure 6.5. The World Coordinate System used in DIS applications.

Before position/orientation information can be converted from one system to the other, REU must first be initialized. This entails mapping the origin of the flat earth terrain to its corresponding point on the surface of the earth in WCS coordinates. This flat earth origin in WCS coordinates is referred to as FEO_W. Using the Fort Knox, Kentucky terrain supplied with ModSAF as an example, if the point on the surface of the flat earth terrain (which also falls within the boundaries of the Knox terrain) at 37°40'N 86°26'W is used as the origin of that flat earth terrain model, then its corresponding point FEO_W in WCS xyz coordinates would be (314487.7674, 5045478.7908, 3876337.4122). This is calculated using data from the WGS 84 standard [DMA1]. From this, REU calculates the angle of rotation around the WCS Z-axis necessary to align FEO_W with the 90° West line of longitude. This angle is theta. It then calculates beta, the angle of

rotation around the WCS X-axis necessary to align FEO_W with the North Pole. These conversion of points between the two coordinate systems is straightforward as seen in Figure 6.6.

NED to WCS	WCS to NED
• NED Coordinate System is mapped to WCS such that NED +X lies on WCS +Y, NED +Y lies on WCS +X, and NED +Z lies on WCS -Z • Coordinate is raised up by the earth's radius • Coordinate is rotated around WCS +X by beta • Coordinate is rotated around WCS +Z by theta	• NED Coordinate System is mapped to WCS such that NED +X lies on WCS +Y, NED +Y lies on WCS +X, and NED +Z lies on WCS -Z • Coordinate is rotated around WCS +Z by -theta • Coordinate is rotated around WCS +X by -beta • Coordinate is lowered by the earth's radius

Figure 6.6. Algorithms for converting between WCS and NED Coordinate Systems.

REU makes extensive use of the Performer version 2.0 library matrix and vector functions. This limits the Wingman to operation on SGI machines configured with Performer 2.0. This is unfortunate since Performer is primarily a graphics library and the Wingman produces no graphical output. Another point of concern is that Performer automatically allocates memory to be used as shared arenas. Care must be taken to ensure these arenas don't conflict with those of the CODB.

6.5. Terrain Module

The Terrain Module developed for the Fuzzy Wingman is a C++ class that makes use of existing ModSAF compact terrain database (CTDB) files and their accompanying functions written in the C programming language. The ModSAF CTDBs were chosen to maintain consistency with ModSAF, the primary application whose CGFs the Wingman would be tested against. The ModSAF Knox terrain is shown in Figure 6.7.

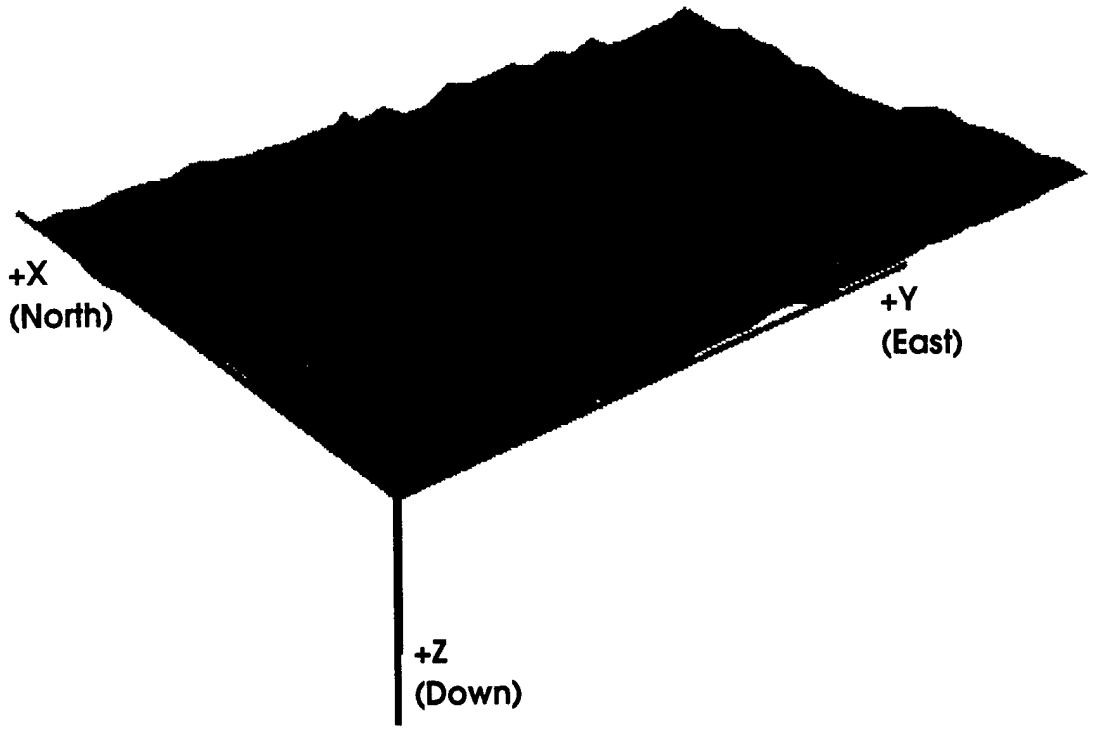


Figure 6.7. The ModSAF Knox terrain used in with the Fuzzy Wingman. Elevation is exaggerated by a factor of 15 to highlight the peaks and valleys.

The primary purpose of the Terrain Module is to provide elevation information about the terrain over which the Wingman is flying. Using the ModSAF Knox CTDB proved to be an effective way to ensure consistency of terrain elevation data between applications.

6.6. Maneuver Container Module

In order to determine the best maneuver to fly in a particular situation, the Wingman must have a repository of known maneuvers that the aircraft is able to perform. The Maneuver Container Module provides this repository.

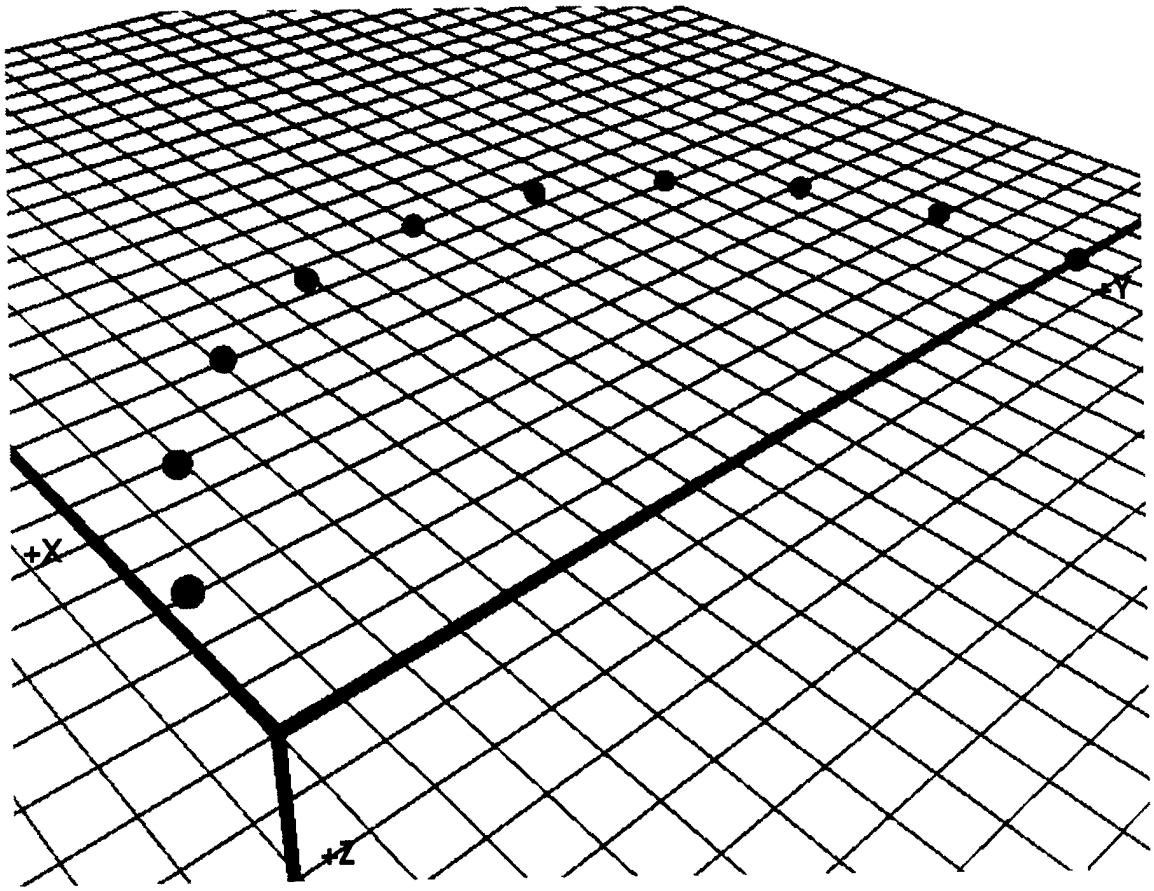


Figure 6.8. The 3D points that comprise a simple lead right turn.

Individual maneuvers are represented as sets of 3D points in the Maneuver Coordinate System (MCS). The MCS is identical to the NED Coordinate System, with the exception of the location of the origin. The MCS origin semantically marks the beginning of the maneuver and is laid on top of the Aircraft Body Coordinate System when the maneuver is initiated as in Figure 6.8. A variable number of subsequent points follow to define the particular maneuver. Table 6.5 shows how each point in the set has an x, y, and z value, and an associated heading, pitch, and roll. The Maneuver Container Module stores these variably-sized point sets as linked lists.

NED					
X	Y	Z	Heading	Pitch	Roll
154.5	24.5	0.0	18.0	0.0	30.0
293.9	95.5	0.0	36.0	0.0	30.0
404.5	206.1	0.0	54.0	0.0	30.0
475.5	345.5	0.0	72.0	0.0	30.0
500.0	500.0	0.0	90.0	0.0	30.0
475.5	654.5	0.0	108.0	0.0	30.0
404.5	793.9	0.0	126.0	0.0	30.0
293.9	904.5	0.0	144.0	0.0	30.0
154.5	975.5	0.0	162.0	0.0	30.0
0.0	1000.0	0.0	180.0	0.0	30.0

Table 6.5. Numerical representation of a lead right turn. Each row defines a subsequent point in the maneuver.

Maneuvers are grouped together as either offensive, defensive, or high aspect. Each group of maneuvers is then placed in the module as a higher level linked list, as illustrated in Figure 6.9. Data dumps of the contents of the Maneuver Decision Module proved it to be a reliable storage and retrieval facility, providing abstract, rapid access to the maneuvers employed by the Wingman.

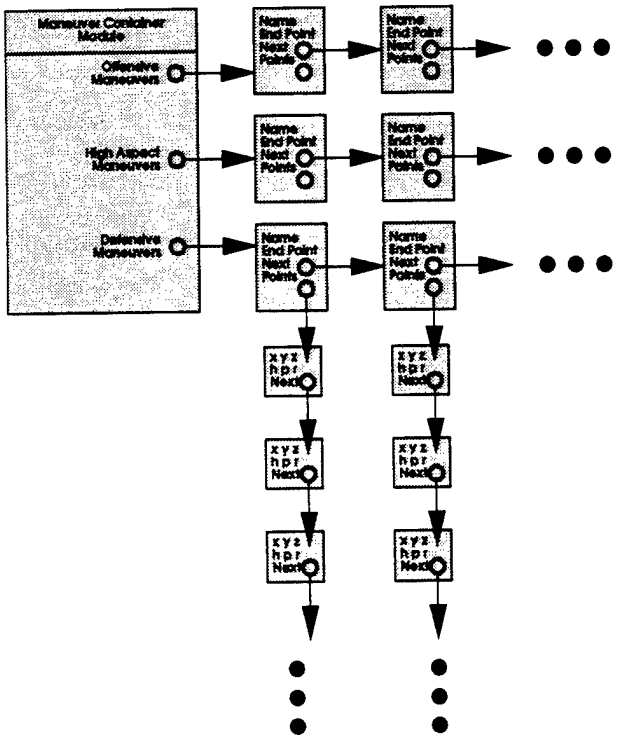


Figure 6.9. Organization of maneuvers within the Maneuver Container Module.

6.7. Weapon Module

The Wingman employed a simplistic Weapon Module to provide the ability to fire, propagate, and detonate weapons. This module contains NED propagation models of Mk-82 bombs and AIM-120 AMRAAMs. It can easily be expanded to model any other type of dumb bomb or air intercept missiles. This is due to the fact that weapon parameters such as acceleration, maximum speed, maximum range, and lethal blast radius are maintained in separate data files. This also facilitates the use of either classified or unclassified values based on the security of the system. No classified weapons data was used in this Weapon Module or any of its weapons data files.

The Mk-82 propagation model drops the bomb according to simple Newtonian acceleration models. It performs computationally inexpensive estimations of aerodynamic considerations. Each bomb is initially given the same x, y, and z velocities and accelerations as the Wingman aircraft as reported by the AeroModel at bomb release time. The initial bomb velocity vectors are then multiplied by a random value very near to 1.0 to provide a slight dispersion factor for multiple bombs as they drop. When observed in ModSAF, the bombs appeared to fall in a realistic manner, dispersing slightly and drifting behind the Wingman as the aircraft continued its flight. The bombs detonated properly upon impact with the ground. However, ModSAF only allows direct-fire weapons to destroy vehicles. Since ModSAF does not recognize Mk-82s as direct-fire weapons, the lethality of the Mk-82s could not be tested.

The AIM-120 propagation model is very lethal due to its simplicity. When launched, it simply accelerates along the Wingman's velocity vector for two seconds (target acquisition phase) and then immediately orients itself and its velocity and acceleration vectors toward the target aircraft (target lock phase). With every subsequent iteration, it repeats this reorientation, until it reaches its intended target or runs out of propellant and crashes to the ground. In every test in which the target was within range, the AMRAAM scored a kill. This closely approximates the probability of kill that has been quoted by domain experts. Although most aspects of the missile model suffice for their low level of complexity, the transition from target acquisition to target lock would benefit by being smoothed out, possibly with a splining technique.

6.8. Sensor Module

No explicit Sensor Module was built for the Wingman. Instead, the Fuzzy Wingman was simply endowed with perfect knowledge of its environment by having access to information about all entities in its DVE. This was done in order to facilitate testing of formation flying and weapon employment. To properly model an F-15 or F-16 aircraft, appropriate Sensor Modules must be incorporated.

6.9. Strategic Decision Engine

A generic Strategic Decision Engine is charged with the high level responsibilities of understanding and implementing mission level goals, communicating with other players, interpreting the surrounding environment, and revising mission goals and subgoals. In the case of the Fuzzy Wingman, the SDE has limited responsibilities in these areas. It does not communicate with any other entities; the intended means of inter-entity communication was to have been the Command and Control Simulation Interface Language (CCSIL), however, this software was unavailable during the implementation phase of this project. Nor does the Wingman revise mission goals; the scope of this project precludes implementation of this responsibility.

6.9.1. SDE Control Module The Wingman's SDE Control Module exists as the method `decision_logic` in the SDE class. It currently embodies the strategic-level requirements of a DCA mission in an F-15E aircraft, such as communicating with the lead aircraft, setting target priorities, determining mission state, checking radar, interrogating threat aircraft, performing visual checks, determining intercept flow state, determining strategies, and assessing the mission. Because the SDE is not currently responsible for revising mission goals, it is able to simply remain in the `decision_logic` loop until the mission has been completed.

6.9.2. SDE F-15E Defensive Counter-Air Mission Module The SDE F-15E DCA Mission Module is embodied in the SDE class as a series of methods. These methods perform the strategic-level requirements of a DCA mission in an F-15E aircraft as outlined in Air Combat Command's F-15E Fighter Fundamentals Manual [ACC92].

6.10. Tactical Decision Engine

A TDE's role is to manage the moment to moment operations of the entity. In the Wingman, this entails receiving the specific mission from the SDE (in this case DCA only) and implementing it according to the mission employment tactics of an F-15E Eagle. This includes activities such as flying formation, determining which basic fighter maneuvers (BFMs) to perform in a threat environment in which one or more bandit aircraft exist, and determining when to employ air intercept missiles (AIMs).

6.10.1. TDE Control Module The Wingman's TDE Control Module exists as the method `decision_logic` in the TDE class. It currently embodies the tactical-level requirements of a DCA mission in an F-15E aircraft, such as determining BFM state, determining the best maneuver to perform, getting the status of the bandit aircraft, and determining whether to release weapons.

6.10.2. TDE F-15E Defensive Counter-Air Mission Module The TDE F-15E DCA Mission Module is embodied in the TDE class as a series of methods. These methods perform the tactical-level requirements of a DCA mission in an F-15E aircraft as outlined in Air Combat Command's F-15E Fighter Fundamentals Manual [ACC92].

6.11. Critical Decision Engine

The CDE is ultimately to embody the survival instinct reflexes of the Wingman whose responsibility includes taking over control of the aircraft in emergency situations where a collision or other immediate and dire threat is eminent. As implemented, the CDE monitors the aircraft for ground collisions. When a collision is detected, the CDE clears the `Alive` flag in the CODB's `CDEStruct` to signal other processes to cease processing.

6.12. Flight Control Module

A large portion of the implementation effort focused on enabling the Wingman to fly formation and fly point to point. These functions are both manifested in the Flight Control Module.

6.12.1. Fly Point To Point The Wingman's task of flying from point to point brings together a vast majority of the system components. The TDE obtains the set of maneuver points to be flown from the Maneuver Decision Module. It then passes the points to the Flight Control Module via the CODB. The Flight Controller obtains the current aircraft positional data from the AeroModel through the CODB, uses fuzzy logic to decide upon the appropriate aircraft controls, and then passes these controls through the CODB to the AeroModel to alter the dynamics of the aircraft. This section will focus on the use of fuzzy logic in determining the aircraft controls. Creating a fuzzy logic controller entails the creation of many smaller items, such as rule chaining diagrams, reasoning schemas, and term sets. Benslay provides a thorough treatment of these [BENS96]. The products of interest here are the term sets and rule sets. But before any numbers can be processed through a controller, the identity of those numbers must be known. It is therefore important to gain an insight into how a human pilot goes about the task of flying a high-performance fighter aircraft through a point in space. This is essentially a three step process:

- First, the aircraft is rolled until the destination point is aligned on the negative Z half of the XZ plane of the aircraft body coordinate system (ABCS).
- Back pressure is applied to the stick until the destination point lies directly ahead of the aircraft.
- The aircraft is rolled again until its wings are level with the ground. The aircraft is then maintained in this attitude until it passes through the point.

Several term sets are necessary to model this behavior. These are listed in Table 6.6. The terms of each set are named such that they complete the phrase *Target is* _____ *Wingman*. The complete contents of each term set are included in Appendix C.

Term Set	Functionality
RelativelyX	Measures relative X in NED coordinate system in angular fashion.
RelativelyY	Measures relative Y in NED coordinate system in angular fashion.
RelativelyZ	Measures relative Z in NED coordinate system in angular fashion.
RelativelyDist	Measures relative distance in NED coordinate system.

Table 6.6. Term sets required to fly point to point.

In addition to the terms sets, a set of 81 rules were devised to model the three steps of flying to a point in 3D space. Eighty-one rules were needed to provide coverage for all combinations of RelativelyX, RelativelyY, RelativelyZ, and RelativelyDist.

Figure 6.10 shows the results of using these term sets and rules to control the aircraft. Figure 6.10(a) shows the four points comprising the zigzag route to be flown with the Wingman just under way at the starting point initially flying straight East. Figures 6.10(b)-(e) show the Wingman as it approaches each of the points. Data was collected from five runs over this course. These complete data can be found in Appendix D. A summary of the minimum proximity of the Wingman to the points is presented in Table 6.7.

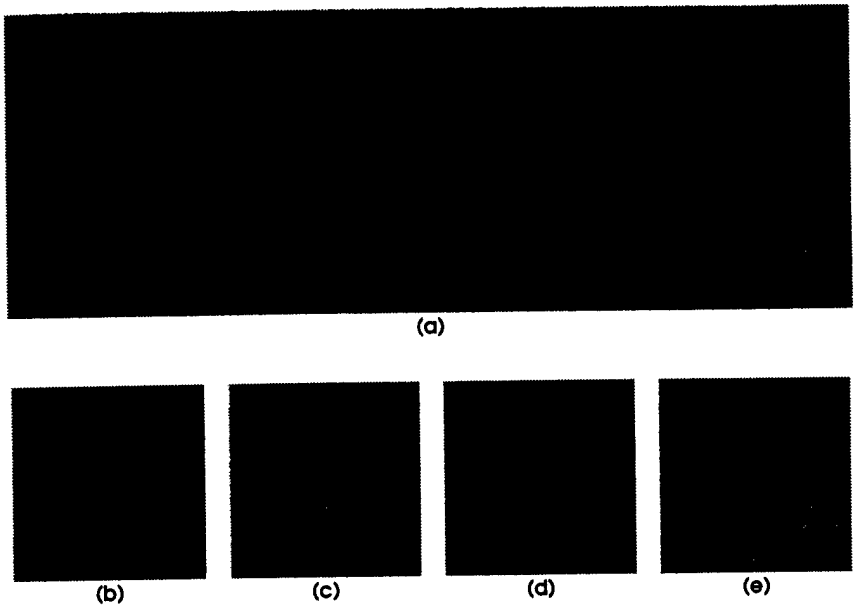


Figure 6.10. ModSAF representation of the Wingman flying a zigzag course over Fort Knox.

Minimum Proximity to Point	Run 1 (feet)	Run 2 (feet)	Run 3 (feet)	Run 4 (feet)	Run 5 (feet)
1	299	291	287	287	285
2	294	299	295	294	296
3	304	291	292	294	294
4	303	299	298	302	311
Average Minimum Proximity	300	295	293	294.25	296.5
Overall Average Minimum Proximity	295.75				

Table 6.7. Minimum proximity between the Wingman and the zigzag points across five runs.

One may ask why the Wingman can't seem to get much closer than 300 feet to its intended route points. A suggested solution might be to tighten up the values used in the fuzzy term sets, but doing so

would defeat the original intent of using fuzzy logic. The purpose was to introduce a measure of variability in the Wingman's actions to provide a better emulation of a human operator. Indeed, the very fact that the Wingman only ever gets to within about 300 feet of its points indicates that the fuzzy logic is working exactly as planned.

7. Conclusions

In this thesis, a new architecture was presented that facilitates the design and construction of CGFs of any domain and addresses the important issue of concurrent processing. The use of such an architecture can have profound effects on the way CGFs are constructed. No longer would developers need to start from the ground floor with every new CGF built. Instead, repositories of CGF Module Design Templates could be maintained for all to either reuse or modify for their own particular needs.

Widespread use of this new architecture could enable a greater variety of CGFs to be built, so Distributed Virtual Environments could be populated with a more realistic set of entities. This equates to more realistic training exercises, resulting in better prepared military forces.

The Fuzzy Wingman implementation of this architecture shows the good deal of extensibility afforded by the CGF Architecture. Although it proved the architecture viable, the Wingman would benefit greatly from a more robust set of modules. Weapon, sensor, and communication modules all need to be enhanced. On the other hand, many of the critical infrastructure components proved themselves to be reliable. The CODB provided an easy method of inter-process communication and control. The DISManager efficiently passed PDUs between the application and the DVE.

An excellent opportunity for future research lies in the modification of the CODB to enable communication and control between *distributed* processes, as opposed to merely multiple processes running concurrently on one machine. This would allow for a much more powerful CGF that could take advantage of numerous modules developed under the CGF Architecture.

Appendix A
CGF Module Design Templates

A.1. Generic CGF Module Design Templates

GENERIC CGF MODULE DESIGN TEMPLATE

Type and Function:

Decision Process Module Type: Maneuver Decision Module

Intended User: TDE

Specific Areas of Concern and their requirements:

1. How do I choose which maneuver to apply in a given situation?

   a) Required: A knowledge of available maneuvers.

      Suggested: A maneuver container module.

   b) Required: A knowlege of to whom the maneuver is to be applied against and where they are located, or an area the maneuver is to be performed in.

      Suggested: Method parameter.

   c) Required: A knowledge of the goal in applying the maneuver.

      Suggested: Assumed, method parameter, lookup, or module.

   d) Required: A knowledge of if we're reevaluating a pervious maneuver decision that we're curently carrying out, or a first time decision.

      Suggested: Method parameter.

   e) Required: A knowledge of my location.

      Suggested: Lookup.

   f) Required: A knowledge of an adversary and his known abilities.

      Suggested: An adversary information module.

   g) Required: A methodology or representation of applying the known maneuvers in relation to another entity or area to accomplish the goal.

      Suggested: An artificial intelligence algorithm/methodology, potentially in a separate module.

   h) Required: A mechanism for relaying the decision to those that need it.

      Suggested: Returned structure, reference pointer in argument list, or TDE intracommunication methodology.

2. How do I take the environment into account in decision making?

   a) Required: A knowledge of the terrain.

      Suggested: A terrain container module.

   b) Required: A knowlege of other entities, where they are at, what they are, and what they are doing (when allowable).

      Suggested: An array of sensor modules.

   c) Required: A knowledge of atmospheric conditions.

Suggested: An atmosphere module.

Generic Algorithm:

Given the following:

1) my location,

2) an entity to which I'm supposed to move with or against and its location, OR an area that I'm to move within,

3) an initial goal;

Determine if the goal is obtainable;

Determine how much time is available to make a decision;

In the given time, choose the best maneuver which comes closest to accomplishing the goal that:

1) doesn't violate terrain knowlege,

2) takes into account atmospheric conditions,

3) considers other entities and their potential adverse actions;

Report the decision.

A.2. Applied CGF Module Design Templates

APPLIED CGF MODULE DESIGN TEMPLATE

Module Type: Function: Platform Type:

Decision Process Module Type: Maneuver Decision Module: JetFighter

Intended User: TDE

(NOTE: Applied specifics for this type are preceeded by an "-->").

--> Mission Type:

Defensive Counter Air (DCA)

--> Target Domain:

Air

--> Artificial Intelligence Methodology:

Asynchronous, adversarial game tree with fuzzy spatial relationship evaluators.

--> Software Engineering Methodology:

Modular Pseudocode Decomposition of Object Oriented Design

Specific Areas of Concern, their requirements, and platform type specifics:

1. How do I choose which maneuver to apply in a given situation?

a) Required: A knowledge of available maneuvers.

Suggested: A maneuver container module.

--> Knowledge representation of maneuvers should represent movement in 3D space, and should be identified in terms of the actual maneuvers they represent.

b) Required: A knowlege of to whom the maneuver is to be applied against and where they are located, or an area the maneuver is to be performed in.

Suggested: Method parameter.

--> It would be useful to have some sort of pointer to where this entity resides in order to quickly access the necessary data such as location and type. Also, as a backup, give some other identifying information about the entity that can verify the entity being pointed to.

c) Required: A knowledge of the goal in applying the maneuver.

Suggested: Assumed, method parameter, lookup, or module.

--> Highly dependent on mission type. Because of the complexities involved with making a single maneuver decision module consider all the possible missions, their individual states, and the resulting possible maneuver combinations, this applied module is built specifically for a DCA mission. Hence, the goal of applying the maneuver is implied: maneuver this aircraft as to put it in an optimum position for firing missiles or guns at a bandit aircraft while denying the bandit any superior positional advantage.

Consequently, this module will need to have knowledge of the available weapons that can be employed and their performance characteristics so that these parameters can be used in the maneuver evaluations. Hence, this module should have a

reference to a weapon data container module.

d) Required: A knowledge of if we're reevaluating a previous maneuver decision that we're currently carrying out, or a first time decision.

Suggested: Method parameter.

--> The TDE should decide and pass as a parameter if this is the first decision for this target.

e) Required: A knowledge of my location.

Suggested: Lookup.

--> This module should have a simple way of ascertaining the aircraft's exact location and orientation.

f) Required: A knowledge of an adversary and his known abilities.

Suggested: An adversary information module.

--> In order to determine the correctness of a particular maneuver combination, possible adversarial responses to established maneuvers should be known. For simplicity, and when engaged against another like acft, these could be a duplication of the known maneuvers for this aircraft type. For higher fidelity reasoning, an adversary information module should be developed, especially if the target is dissimilar.

g) Required: A methodology or representation of applying the known maneuvers in relation to another entity or area to accomplish the goal.

Suggested: An artificial intelligence algorithm/methodology, potentially in a separate module.

--> For this application, modify an adversarial game tree for asynchronous play which uses a fuzzy inferencing engine to score positions by evaluating the geometry between this aircraft and the bandit. The game tree should compare possible maneuvers and their relative strengths/weaknesses and choose an appropriate maneuver.

Naturally, depending on the number of resulting maneuver combinations, the number of node evaluations could grow quite large, so some heuristic must be applied to trim the game tree, as well as establishing a time limit in which a decision must be reached. Depending on system level design methodologies, choose a heuristic which reflects the pilot's abilities to make the same choice.

The game tree should take into account the types of maneuvers used for a specific DCA state, as well as disgarding any maneuver that violates the terrain knowledge.

Measure and fuzzify the following Battle Space Domain measurement types:

Range to Designated Target (RDT)
Relative Target Altitude (RAL)
Target Aspect (TAS)
Approach Quarter Nose (AQN)
Approach Quarter Tail (AQT)

Create an evaluation term set which describes the "goodness" of the resulting geometry configuration.

h) Required: A mechanism for relaying the decision to those that need it.

Suggested: Returned structure, reference pointer in argument list, or TDE intracommunication methodology.

--> Pass by reference parameter that points to a modified maneuver point list.

2. How do I take the environment into account in decision making?

a) Required: A knowledge of the terrain.

Suggested: A terrain container module.

--> This module should have access to a terrain container module, and not necessarily own one itself. Perhaps the best technique would be to pass a reference as an argument parameter, or through an intracommunictaion mechanism like the CODB, as long as immediate use can be made of it.

b) Required: A knowlege other entities, where they are at, what they are, and what they are doing (when allowable).

Suggested: An array of sensor modules.

--> Similar to the terrain container module, this module should have access to the sensor modules that are owned by other modules in the system. Likewise, references to these modules could be made through argument parameters or through the intracommunictaion mechanism.

c) Required: A knowledge of atmospheric conditions.

Suggested: An atmosphere module.

--> This module should have access to an atmospheric module, and not necessarily own one itself. Perhaps the best technique would be to pass a reference as an argument parameter, or through an intracommunictaion mechanism like the CODB, as long as immediate use can be made of it.

Generic Algorithm:

Given the following:

1) my location,

2) an entity to which I'm supposed to move with or against and its location, OR an area that I'm to move within,

3) an initial goal;

Determine if the goal is obtainable;

Determine how much time is available to make a decision;

In the given time, choose the best maneuver which comes closest to accomplishing the goal that:

1) doesn't violate terrain knowlege,

2) takes into account atmospheric conditions,

3) considers other entities and their potential adverse actions;

Report the decision.

Applied Algorithm: Jet Fighter

--> Mission Type:

Defensive Counter Air (DCA)

--> Artificial Intelligence Methodology:

Asynchronous, adversarial game tree with fuzzy spatial relationship evaluators.

--> Software Engineering Methodology:

Modular Pseudocode Decomposition of Object Oriented Design

1. - Construct the module object with the needed objects.

2. - Determine pilot performance parameters.

3. - Determine locations and orientations of aircraft and bandit.

4. - If target is offensive capable, evaluate geometry for a no-win situation.

5. - As a function of geometry, determine the maximum allowable decision making time.

6. - Determine if this decision process is the first applied to this target. If not, then evaluate whether or not the actual movements are in keeping (by some fuzzified amount) with those decided on. If either the aircraft or the bandit are sufficiently off the chosen maneuver, then scrap the current maneuver and reevaluate. As long as the aircraft and bandit are performing reasonably as expected, then keep the score of the current maneuver as a comparrison to consequent evaluations.

7. - Depending on the bandit type and the geometry involved, construct a game tree based on the known maneuvers for those posture states.

8. - Determine pruning heuristics.

9. - Based on positions, bandit type, velocities, maneuver selections, weapon choices, and terrain info, evaluate as many combinations as possible without exceeding the maximum allowable decision making time. Retain the highest scoring combination.

10. - If this is a follow-on maneuver evaluation for a bandit, then compare the results of this decision with the previous one. Choose the better of the two and report that for implementation.

11. - If directed, record necessary data for a decision trace.

A.3. Complete CGF Module Design Templates

COMPLETE ICGF MODULE DESIGN TEMPLATE

Module Type: Function: Platform Type: Role:

Decision Process Module Type: Maneuver Decision Module: JetFighter: Wingman

Intended User: TDE

(NOTE: Applied specifics for this type are preceeded by an "-->").

(NOTE: Complete specifics for this type are preceeded by an ">>>").

--> Mission Type:

Defensive Counter Air (DCA)

>>> Combat Air Patrol as defined by Shaw.

--> Target Domain:

Air

>>> As defined by IST-CR-95-14.

--> Artificial Intelligence Methodology:

Asynchronous, adversarial game tree with fuzzy spatial relationship evaluators.

>>> Use FuzzyCLIPS as the fuzzy inferencing logic.

--> Software Engineering Methodology:

Modular Pseudocode Decomposition of Object Oriented Design

>>> Intracommunication Structure:

Common Object Data Base (CODB)

>>> Virtual Entity Managers

DIS Manager + World State Manager

Specific Areas of Concern, their requirements, and platform type specifics:

1. How do I choose which maneuver to apply in a given situation?

a) Required: A knowledge of available maneuvers.

Suggested: A maneuver container module.

--> Knowledge representation of maneuvers should represent movement in 3D space, and should be identified in terms of the actual maneuvers they represent.

>>> Maneuvers are modeled as a series of points in a matrix, sufficient in number to allow the maneuver to maintain a recognizable shape when scaled in 3D to meet the geometry requirements. The maneuver container module should keep the methods necessary to read in and lookup the maneuvers and their points.

DESIGN CONCERN: It is possible, that with all the maneuvers possible from a particular state from a particular task, that the maneuver module could be quite large, and the overhead for categorizing the structure could be overly complicated if not handled properly.

b) Required: A knowlege of to whom the maneuver is to be applied against and where they are located, or an area the maneuver

is to be performed in.

Suggested: Method parameter.

--> It would be useful to have some sort of pointer to where this entity resides in order to quickly access the necessary data such as location and type. Also, as a backup, give some other identifying information about the entity that can verify the entity being pointed to.

>>> The module will need two things: 1) a pointer to the entity_identification_record that has the detailed specs of the target, and 2) an index into the entity array within the World State Manager for quick lookups.

c) Required: A knowledge of the goal in applying the maneuver.

Suggested: Assumed, method parameter, lookup, or module.

--> Highly dependent on mission type. Because of the complexities involved with making a single maneuver decision module consider all the possible missions, their individual states, and the resulting possible maneuver combinations, this applied module is built specifically for a DCA mission. Hence, the goal of applying the maneuver is implied: maneuver this aircraft as to put it in an optimum position for firing missiles or guns at a bandit aircraft while denying the bandit any superior positional advantage.

Consequently, this module will need to have knowledge of the available weapons that can be employed and their performance characteristics so that these parameters can be used in the maneuver evaluations. Hence, this module should have a reference to a weapon data container module.

d) Required: A knowledge of my location.

Suggested: Lookup.

--> This module should have a simple way of ascertaining the aircraft's exact location and orientation.

>>> Access to the aircraft's entity identification record through the CODB.

e) Required: A knowledge of an adversary and his known abilities.

Suggested: An adversary information module.

--> In order to determine the correctness of a particular maneuver combination, possible adversarial responses to established maneuvers should be known. For simplicity, and when engaged against another like acft, these could be a duplication of the known maneuvers for this aircraft type. For higher fidelity reasoning, an adversary information module should be developed, especially if the target is dissimilar.

>>> To maintain simplicity on this first module, consider that the bandit aircraft has the same maneuver database as we do, hence use the appropriate BFM State category of the Maneuver Decision Module.

f) Required: A methodology or representation of applying the known maneuvers in relation to another entity or area to accomplish the goal.

Suggested: An artificial intelligence algorithm/methodology, potentially in a separate module.

--> For this application, modify an adversarial game tree for asynchronous play which uses a fuzzy inferencing engine to score positions by evaluating the geometry between this aircraft and the target. The game tree should compare possible maneuvers and their relative

strengths/weaknesses and choose an appropriate maneuver.

Naturally, depending on the number of resulting maneuver combinations, the number of node evaluations could grow quite large, so some heuristic must be applied to trim the game tree, as well as establishing a time limit in which a decision must be reached. Depending on system level design methodologies, choose a heuristic which reflects the pilot's abilities to make the same choice.

The game tree should take into account the types of maneuvers used for a specific DCA state, as well as disgarding any maneuver that violates the terrain knowledge.

Measure and fuzzify the following Battle Space Domain measurement types:

Range to Designated Target (RDT)
Relative Target Altitude (RAL)
Target Aspect (TAS)
Approach Quarter Nose (AQN)
Approach Quarter Tail (AQT)

Create an evaluation term set which describes the "goodness" of the resulting geometry configuration.

>>> Use FuzzyCLIPS as the inferencing logic.

DESIGN CRITERION 1: FuzzyCLIPS is a regular C product and not a C++ product. Hence, when the library files are used within an object, it is important that only one process be allowed to use that library at any given time. Otherwise, if multiple processes attempt to access and use the library, FuzzyCLIPS has no mechanism to keep separate rule firing stacks or state spaces for the individual processes. Hence when considering where this module is to be used within the TDE (or other DE), it must exist in an object that uses a single process approach.

DESIGN CRITERION 2: Because this Maneuver Decision Module will more than likely co-exist with other Maneuver Decision Modules within the DE, it will be important to introduce the rule sets as separate modules within the FuzzyCLIPS corporate memory. This will allow this Maneuver Decision Module to specify the subset of rules with which it is concerned. This is important in that if each of the Maneuver Decision Modules have 300+ rules, you do not want to incur the performance hit of evaluating rules that do not pertain to your mission perspective.

Game Tree Construction: Note that this will not be a traditional (ie static & synchronous) game tree, but one modified for dynamic & asynchronous input. Instead of separate minimizing and maximizing levels, there will be only two major levels below the root node - one to describe the Wingman's possible moves, then one to describe the Bandit's possible moves. Nodes beyond this point will describe fuzzily determined evaluations of the resulting spatial orientations based on a yet to be defined time interval.

Pruning Heuristics: Determine by Pilot Profile (PP) if the algorithm will receive any additional pruning heuristics in the way of hints to which combination of maneuvers to look first (could be given as a .dat file, easilly modifiable, that is read in from program start).

Evaluations: The first evaluations should be a "snapshot" of what the resulting geometry will look like should the chosen maneuvers be employed.

Tree Traversal: Should begin with those maneuver combinations that appear in the "hints" file as discerned by the PP heuristic, and which also have a snapshot evaluation above a certain threshold. If this combination does not

exist, then simply start with the combination that has the best evaluation.

"Look Ahead" Time: As a function of PP, determine the intervals of time that will be used to evaluate the intermediate spatial orientations of the Wingman and Bandit as they proceed through the maneuver combinations. For instance, the h8igher the PP rating, the farther ahead the pilot will be able to accurately discern the potential outcome of his chosen maneuver.

The following Battle Space Domain measurement types have the following term sets:

RDT: nil, close, near, far
RAL: low, same, high
TAS: cold, nil, hot
AQN: left_beam, nose, right_beam
AQT: right_beam, tail, left_beam

Go through the process of illiciting the knowledge neccessary from a domain expert (fighter pilot) as to the terms to use in the evaluation term set, the membership functions for this and the measurement sets, and the rules that model the pilot situational evaluation behaviors.

g) Required: A mechanism for relaying the decision to those that need it.

Suggested: Returned structure, reference pointer in argument list, or TDE intracommunication methodology.

--> Pass by reference parameter that points to a modified maneuver point list.

>>> The main decision logic function of the module should return an enumerated error code type that can be checked in the calling method prior to utilizing the referenced pointer.

2. How do I take the environment into account in decision making?

a) Required: A knowledge of the terrain.

Suggested: A terrain container module.

--> This module should have access to a terrain container module, and not necessarily own one itself. Perhaps the best technique would be to pass a reference as an argument parameter, or through an intracommunictaion mechanism like the CODB, as long as immediate use can be made of it.

>>> Use a module that reads in and stores a ModSAF terrain database and which returns a terrain altitude based on X Y coordinates.

b) Required: A knowlege other entities, where they are at, what they are, and what they are doing (when allowable).

Suggested: An array of sensor modules.

--> Similar to the terrain container module, this module should have access to the sensor modules that are owned by other modules in the system. Likewise, references to these modules could be made through argument parameters or through the intracommunictaion mechanism.

>>> For now, a simplified radar model is being integrated into the system that simply reports the entities that are visible as a result of applying a filter algorithm.

c) Required: A knowledge of atmospheric conditions.

Suggested: An atmosphere module.

--> This module should have access to an atmospheric module, and not necessarily own one itself. Perhaps the best technique would be to pass a reference as an argument parameter, or through an intracommunictaion mechanism like the CODB, as long as immediate use can be made of it.

>>> No atmospheric module is planned at this time.

Generic Algorithm:

Given the following:

1) my location,

2) an entity to which I'm supposed to move with or against and its location, OR an area that I'm to move within,

3) an initial goal;

Determine if the goal is obtainable;

Determine how much time is available to make a decision;

In the given time, choose the best maneuver which comes closest to accomplishing the goal that:

1) doesn't violate terrain knowlege,

2) takes into account atmospheric conditions,

3) considers other entities and their potential adverse actions;

Report the decision.

APPLIED ALGORITHM: JET FIGHTER

--> Mission Type:

Defensive Counter Air (DCA)

--> Target Domain:

Air

--> Artificial Intelligence Methodology:

Asynchronous, adversarial game tree with fuzzy spatial relationship evaluators.

--> Software Engineering Methodology:

Modular Pseudocode Decomposition of Object Oriented Design

1. - Construct the module object with the needed objects.

2. - Determine pilot performance parameters.

3. - Determine locations and orientations of aircraft and target.

4. - Determine if this decision process is the first applied to this target.

5. - If target is offensive capable, evaluate geometry for a no-win situation.

6. - As a function of geometry, determine the maximum allowable decision making time.

7. - Depending on the target domain and the geometry involved, construct a game tree based on the known maneuvers for those posture states.

8. - Determine pruning heuristics.

9. - Based on positions, target domain type, velocities, maneuver

selections, weapon choices, and terrain info, evaluate as many combinations as possible without exceeding the maximum allowable decision making time. Retain the highest scoring combination.

10. - If this is a follow-on maneuver evaluation for a target, then compare the results of this decision with the previous one. Choose the better of the two and report that for implementation.

11. - If directed, record necessary data for a decision trace.

COMPLETE ALGORITHM: Jet Fighter: Wingman

--> Mission Type:

Defensive Counter Air (DCA)

>>> Combat Air Patrol as defined by Shaw.

--> Target Domain:

Air

>>> As defined by IST-CR-95-14.

--> Artificial Intelligence Methodology:

Asynchronous, adversarial game tree with fuzzy spatial relationship evaluators.

--> Software Engineering Methodology:

Modular Pseudocode Decomposition of Object Oriented Design

Constructor:

1. - Construct the module object with the needed objects.

- Pass in ptr for REU module.

- Pass in ptr for Maneuver Container Module

- Initialize class attributes

- Load clips file

- Setup debug/dump flags

- Output file setup.

Central Algorithm: Make_Decision

- Pass in ptr to entity_identification_record that identifies specific target

- Pass in target index.

2. - Determine pilot performance parameters.

- Obtain Pilot Profile (PP) rating.

3. - Determine locations and orientations of Wingman and target.

- Get current XYZhpr of Wingman and target.

Determine if Bandit ID as passed in is accurate, if not, search for appropriate ID. If not found, pass back some sort of error flag.

4. - If target is offensive capable, evaluate geometry for a no-win situation.

If Target's score is above a certain threshold, and the PP rating is high enough to realize this, limit tree construction to disengaging maneuvers and set goal to "disengage".

5. - As a function of geometry, determine the maximum allowable decision making time.

   - As a function of relative distance to Wingman and Bandit, determine longest allowable decision time for game tree decision making.

6. - Determine if this decision process is the first applied to this target. If not, then evaluate whether or not the actual movements are in keeping (by some fuzzified amount) with those decided on. If either the aircraft or the bandit are sufficiently off the chosen maneuver, then scrap the current maneuver and reevaluate. As long as the aircraft and bandit are performing reasonably as expected, then keep the score of the current maneuver as a comparrison to consequent evaluations.

   - Passed in as a parameter - decided by the calling method.

   - If following up on a previous decision, evaluate geometry based on actual movements.

     Retrieve previous XYZhpr of both acft, the maneuver decision made, and resulting score.

     Place both acft in the Wingman centered coordinate system, but offset each acft's XYZhpr by the changes observed from the previous iterations.

     Determine how far off the maneuver path both acft have drifted. Fuzzify this value.

     As a function of PP, determine if the algorithm can detect a discernable difference from either acft's actual position from the previously projected position.

       If so - then scrap the current maneuver and reevaluate completely.

       Otherwise, consider the current maneuver score as a baseline to compare all new maneuvers against.

7. - Depending on the bandit type and the geometry involved, construct a game tree based on the known maneuvers for those posture states.

   - If goal has not already been set to disengage, and based on the BFM state (BFM state being determined by CDE), set the goal to "engage", and construct the basic tree by

     (Wingman BFM State maneuvers) x
     (Bandit's opposite BFM State maneuvers)

8. - Determine pruning heuristics.

   - Determine by PP if the algorithm will receive any additional pruning heuristics in the way of hints to which combination of maneuvers to look first (could be given as a .dat file, easilly modifiable, that is read in from program start)

   - Determine "look ahead" time. As a function of PP, determine the intervals of time that will be used to evaluate the intermediate spatial orientations of the Wingman and Badit as they proceed through the maneuver combinations. For instance, the higher the PP rating, the farther ahead the pilot will be able to accurately discern the potential outcome of his chosen maneuver.

9. - Based on positions, bandit type, velocities, maneuver selections, weapon choices, and terrain info, evaluate as many combinations as possible without exceeding the maximum allowable decision making time. Retain the highest scoring combination.

   - Translate and rotate both the Wingman and Bandit according to a Wingman centered coordinate system.

   - Establish maneuver orientation and scaling changes according to the following:

X = distance between Wingman and Bandit
OR = optimum firing range for specified weapon
LBR = weapon lethal blast radius
OR/AAM.vel = T1
W.vel*T + OR + LBR + B.vel*T1 + B.vel*T = X

Take W.vel*T and make that the end point of the maneuver.

Scale maneuver to match last maneuver pt w/ W.vel*T in the XY plane.

Take note of hpr in this position, and extend weapon employment zone according to weapon type.

- First, take "snapshots" of what the spatial geometry will be at the end of the maneuver. Fuzzy evaluations will be based on the following:

  Distance
  Weapon range
  Weapon lethal blast radius
  Bandit aspect
  Wingman orientation

- Use a breadth first search (BFS) algorithm to begin the search and evaluations.

- Start the BFS with the maneuver combination that turned up on the most desired list & which had the highest evaluation. If none fit this category, then start with the highest evaluation.

- Evaluate a given maneuver combination:

  Translate and rotate both the Wingman and Bandit according to the possible distance traveled along the maneuver path.

- Inbetween each fuzzy evaluation, determine the remaining time allowable and if another evaluation is possible. If yes, then continue the BFS algorithm. If not, return the maneuver combination with the best evaluation.

10. - If this is a follow-on maneuver evaluation for a bandit, then compare the results of this decision with the previous one. Choose the better of the two and report that for implementation.

    - If this is a follow up evaluation, determine if the new maneuver score is higher than the newest projected score for the current maneuver.

    - If the new maneuver score is higher, save the geometry and score for the next comparrison, then report a change in tactic with the new maneuver combination.

    - If the new maneuver score is NOT higher, disregard the newest maneuver, and report no change in tactic.

11. - If directed, record necessary data for a decision trace.

Appendix B
Common Object Database Source Code

The Common Object Database (CODB) implements shared memory techniques to allow communication between independent processes. Its functionality is described in detail in Chapter Six.

B.1. dbtypes.h

```
#ifndef _DBTYPES_H_
#define _DBTYPES_H_

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif

// The maximum number of structure types allowed in the system.
#define MaxDoubleBuffers            50
#define MaxCODBProcesses            16
#define CODBArenaBaseAddress 536870912 // 512MB (2^29 = 0x20000000)
#define MaxCODBArenaBytes      1048576 //   1MB (2^20 = 0x00100000)
//#define MaxCODBArenaBytes     131072 // 128KB (2^17)

// List of structure types.  The number of enumerations here must not exceed
// MaxDoubleBuffers.
enum StructureType{
   MouseStruct,          //  0
   KeyboardStruct,       //  1
   AircraftStruct,       //  2
   RendererStruct,       //  3
   HotasStruct,          //  4
   FastrakStruct,        //  5
   WSMStruct,            //  6
   PodStruct,            //  7
   AcftCtrlStruct,       //  8
   StateStruct,          //  9
   EnvironStruct,        // 10
   PerformerWSMStruct,   // 11
   PlanetStruct,         // 12
   MoonStruct,           // 13
   SDEStruct,            // 14
   TDEStruct,            // 15
   CDEStruct,            // 16
   AsteroidStruct,       // 17
   CometStruct,          // 18
   TimeStruct,           // 19
   TrailsStruct,         // 20
   SMStruct,             // 21
   SatStruct,            // 22
   GUIStruct,            // 23
   DummyStruct,          // 24
   WSMEntityStruct,      // 25
   WSMEventStruct,       // 26
   WSMMgtStruct,         // 27
   WSMpfEntityStruct,    // 28
   WSMpfEventStruct,     // 29
   OwnStateStruct,       // 30
   OwnMgtStruct,         // 31
   WSMGPSStruct,         // 32
   PatientVitals,        // 33
   DoctorTreatment,      // 34
   IVPumpStruct,         // 35
   PatientWarmerStruct,  // 36
   DefibStruct,          // 37
```

```
   WingmanStruct,        // 38
   CockpitStruct,        // 39
   RadarStruct,          // 40
   INSStruct,            // 41
   MFDStruct,            // 42
   LocalCoordStruct,     // 43
   WeaponStruct,         // 44
   DISMgrCtrlStruct,     // 45
   WingmanReadyStruct,   // 46
   BombDataStruct,       // 47
   AIMDataStruct,        // 48
   AGMDataStruct         // 49
};

#endif
```

B.2. commonobjdb.h

```
#ifndef __CommonObjectDB__
#define __CommonObjectDB__

#include <ulocks.h>
#include "dbtypes.h"

struct DoubleBufferData{
   ulock_t  ReadAccess;
   ulock_t  WriteAccess;
   ulock_t  ReadCountAccess;
   int      ReadCount;
   void    *ReadBufferPtr;
   void    *WriteBufferPtr;
   size_t   BufferSize;
};//end struct DoubleBufferData

// Data structure to be accessed by the Common Object Database
struct CommonObjectDBData{
   ulock_t          AccessLock;
   DoubleBufferData DoubleBuffer[MaxDoubleBuffers];
   unsigned int     BytesAllocated;
};//end struct CommonObjectDBData

// Arena Control Structure
struct CommonObjectDBControl_t{
   ulock_t            UpdateLock;
   CommonObjectDBData *Data;
};//end struct CommonObjectDBControl_t

// CommonObjectDB Class
template <class T>
class CommonObjectDB{

public:

   // Methods:
            CommonObjectDB(StructureType structure);
      void  *BeginRead(     StructureType structure);
      void   EndRead(       StructureType structure);
      void  *BeginWrite(    StructureType structure);
      void   EndWrite(      StructureType structure);
      void  *BeginReadWrite(StructureType structure);
      void  *BeginMagic(    StructureType structure);
      void   EndReadWrite(  StructureType structure);
      void   EndMagic(      StructureType structure);
      int    Registered(    StructureType structure);

   // Attributes:
      // (None)

private:

   // Methods:
```

```
      void   initCommonObjectDB();
      void   initDoubleBuffer(StructureType  structure);
      void   ReportArenaInfo( StructureType  structure);

   // Attributes:
      usptr_t                    *Arena;
      CommonObjectDBControl_t    *ArenaControl;

};
#endif
```

B.3. commonobjdb.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>
#include <fstream.h>
#include "commonobjdb.h"
#include <sys/types.h> // for getpid
#include <unistd.h>    // for getpid


// The CommonObjectDB constructor joins a specified arena, creating it and
// initializing it if necessary.
template <class T>
CommonObjectDB<T>::CommonObjectDB (StructureType  structure) {

   boolean            init_arena = FALSE;

   int                lock_status;
   int                mem_status;
   int                config_status;

   mode_t             arena_mode;


   // Set usinit tracing variable
   //_utrace = 1;
   _uerror = 1;

   // First, set up the lock type --- Set to DEBUGPLUS will allows for error messages
   // to be printed to the screeen.
   config_status = usconfig(CONF_LOCKTYPE, US_DEBUGPLUS);
   if (config_status == -1)
      cerr <<  "CODB Error #" << errno << " in setting the lock type for the shared memory
arena!" << endl;

   // Next, set up the size and virtual base address of the arena.
   usconfig(CONF_INITSIZE, MaxCODBArenaBytes);
   usconfig(CONF_ATTACHADDR, (void*)CODBArenaBaseAddress);

   // Identify the max number of processes that can leech onto this CODB.
   usconfig(CONF_INITUSERS, MaxCODBProcesses);

   // Join the arena, creating it if necessary.   Exit on error.
   Arena = usinit ("/usr/tmp/CommonObjectDB.arena");
   usconfig(CONF_ATTACHADDR, (void*)~0);
   cerr << "CODB<" << structure << "> (pid " << getpid()
        << ") has Arena at " << Arena << "\n";
   if (Arena == NULL) {
      cerr << "CODB Error #" << errno << " in creating the shared memory arena!" << endl;
      exit(0);
   }

   // Now set the arena to rw permissions across the board (owner, group, world) for the
   // arena.
   arena_mode = ((S_IRUSR) | (S_IWUSR) | (S_IRGRP) | (S_IWGRP) | (S_IROTH) | (S_IWOTH));
   config_status = usconfig (CONF_CHMOD, Arena, arena_mode);
   if (config_status == -1)
```

```
      cerr << "CODB Error #" << errno << " in changing file permissions on the shared
memory arena!" << endl;


   // Do the following as many times as necessary until the arena has been
   // initialized.
   while (!init_arena) {
      ArenaControl = (CommonObjectDBControl_t *) usgetinfo (Arena);
      if (ArenaControl == NULL) {

         // This process appears to be first to call usinit.  Allocate an
         // ArenaControl structure with its UpdateLock already held and 1
         // process joined, and try to swap it into place as the active one.
         // We expect no errors in setting up ArenaControl.  If one occurs,
         // the arena is simply unusable.
         ArenaControl = (CommonObjectDBControl_t *) usmalloc
(sizeof(CommonObjectDBControl_t), Arena);
         if (ArenaControl == NULL) {
            cerr << "CODB Error in memory allocation of shared memory arena!" << endl;
            exit(0);
         } else {

            ArenaControl->UpdateLock = usnewlock (Arena);
            if (ArenaControl->UpdateLock == NULL) {
               cerr << "CODB Error #" << errno << " in allocating lock from shared memory
arena!" << endl;
               exit(0);
            } else {

               lock_status = uscsetlock (ArenaControl->UpdateLock, 1);
               if (lock_status == 0) {
                  cerr << "CODB Error in acquiring lock from the shared memory arena!" <<
endl;
                  exit(0);
               }

            }

         }

         mem_status = uscasinfo (Arena, NULL , ArenaControl);
         if (mem_status == NULL) {

            // uscasinfo either did not find a current value (indicating a
            // race with another process executing this code) or it failed
            // for some other reason.  In any case, release allocated memory
            // and repeat the loop.
            usfreelock (ArenaControl->UpdateLock, Arena);
            usfree (ArenaControl, Arena);

         } else {

            // Our ArenaControl structure is now installed.  Initialize it.
            // The loop now ends.
            initCommonObjectDB ();

            lock_status = usunsetlock (ArenaControl->UpdateLock);
            if (lock_status == -1) {
               cerr << "CODB Error #" << errno << " in releasing lock from the shared
memory arena!" << endl;
               exit(0);
            }

            init_arena = TRUE;

         } // end if-then-else

      } else {

         // Another process has created the arena, and either has initialized
         // it or is initializing it right now.  Acquire the lock, which will
         // block us until initializing is done.
         lock_status = ussetlock (ArenaControl->UpdateLock);
         if (lock_status == 0) {
            cerr << "CODB Error in acquiring lock from the shared memory arena!" << endl;
            exit(0);
```

```
         }

         lock_status = usunsetlock (ArenaControl->UpdateLock);
         if (lock_status == -1) {
            cerr << "CODB Error #" << errno << " in releasing lock from the shared memory
arena!" << endl;
            exit(0);
         }

         init_arena = TRUE;

      } // end if-then-else

   } // end while loop


   // Last thing done in the constructor is to initialize the appropriate
   // DoubleBuffer structure.
   initDoubleBuffer(structure);

}; // end CommonObjectDB constructor



// This method sets up the data structure of the arena.
template <class T>
void CommonObjectDB<T>::initCommonObjectDB () {

   int     i;


   // Allocate new memory for data structure.
   ArenaControl->Data = (CommonObjectDBData *) usmalloc (sizeof(CommonObjectDBData),
Arena);
   if (ArenaControl->Data == NULL) {
      cerr << "initCommonObjectDB(): Error in memory allocation of shared memory arena!"
<< endl;
      exit(0);
   }

   // Initialize the values in the data structure.
   // First the two locks.
   ArenaControl->Data->AccessLock = usnewlock (Arena);
   if (ArenaControl->Data->AccessLock == NULL){
      cerr << "initCommonObjectDB(): Error #" << errno << " in allocating lock from shared
memory arena!" << endl;
      exit(0);
   }

   // Finally, initialize all data.
   for (i = 0; i < MaxDoubleBuffers; ++i){
      // No need to initialize the locks...
      ArenaControl->Data->DoubleBuffer[i].ReadCount      = 0;
      ArenaControl->Data->DoubleBuffer[i].ReadBufferPtr  = NULL;
      ArenaControl->Data->DoubleBuffer[i].WriteBufferPtr = NULL;
      ArenaControl->Data->DoubleBuffer[i].BufferSize     = 0;
   }//next i

   // Report the amount of memory allocated from the shared arena.
   ArenaControl->Data->BytesAllocated = sizeof(CommonObjectDBData);
   cerr << "CODB data structure consuming "
        << ArenaControl->Data->BytesAllocated << " bytes ("
        << (int)((float)
           (ArenaControl->Data->BytesAllocated/MaxCODBArenaBytes)*100)
        << "%) of shared memory arena.\n";

}; // end initCommonObjectDB


template <class T>
void CommonObjectDB<T>::initDoubleBuffer(StructureType structure) {

   int              lock_status;
   DoubleBufferData *ThisDoubleBuffer;
   size_t           DoubleBufferSize;
```

```
    lock_status = ussetlock(ArenaControl->Data->AccessLock);
    if (lock_status == 0) {
       cerr <<  "CODB - initDoubleBuffer(): Error in acquiring lock from the shared memory
arena!" << endl;
       exit(0);
    }

    ThisDoubleBuffer = &ArenaControl->Data->DoubleBuffer[structure];
    if (!Registered(structure)){

       // Initialize the locks.
       ThisDoubleBuffer->ReadAccess = usnewlock(Arena);
       if (!ThisDoubleBuffer->ReadAccess){
          cerr << "Error allocating ReadAccess from shared memory arena.\n";
          exit(0);
       }//endif
       ThisDoubleBuffer->WriteAccess = usnewlock(Arena);
       if (!ThisDoubleBuffer->WriteAccess){
          cerr << "Error allocating WriteAccess from shared memory arena.\n";
          exit(0);
       }//endif
       ThisDoubleBuffer->ReadCountAccess = usnewlock(Arena);
       if (!ThisDoubleBuffer->ReadCountAccess){
          cerr <<"Error allocating ReadCountAccess from shared memory arena.\n";
          exit(0);
       }//endif

       // Initialize the read counter.
       ThisDoubleBuffer->ReadCount = 0;

       // Grab some shared memory space for the two buffers.
       ThisDoubleBuffer->ReadBufferPtr  = usmalloc(sizeof(T), Arena);
       ThisDoubleBuffer->WriteBufferPtr = usmalloc(sizeof(T), Arena);

       // If we've run out of room in the arena, say so!
       if (ThisDoubleBuffer->ReadBufferPtr == NULL ||
           ThisDoubleBuffer->WriteBufferPtr == NULL){
              cerr << "Error allocating Doublebuffer " << structure
                   << " from shared memory arena.\n"
                   << "\nINCREASE SIZE OF ARENA BY "
                   << "INCREASING MaxCODBArenaBytes!\n\n";
              exit(0);
       }//endif

       //  Hang on to the size of this bad boy (we need it for memcpys).
       ThisDoubleBuffer->BufferSize = sizeof(T);

       // Report the amount of memory allocated from the shared arena.
       DoubleBufferSize = 2 * ThisDoubleBuffer->BufferSize;
       ArenaControl->Data->BytesAllocated =
               ArenaControl->Data->BytesAllocated + DoubleBufferSize;
       cerr << "DoubleBuffer <" << structure << "> consuming "
            << DoubleBufferSize << " bytes ("
            << (int)((float)(DoubleBufferSize/MaxCODBArenaBytes)*100)
            << "%) of shared memory arena.\n";
       cerr << "    " << ArenaControl->Data->BytesAllocated << " bytes ("
            << (int)((float)
               (ArenaControl->Data->BytesAllocated/MaxCODBArenaBytes)*100)
            << "%) of shared memory arena is now being used.\n";
    }//endif

    lock_status = usunsetlock(ArenaControl->Data->AccessLock);
    if (lock_status == -1) {
       cerr <<  "CODB - initDoubleBuffer(): Error #" << errno << " in releasing lock from
the shared memory arena!" << endl;
       exit(0);
    }

}


template <class T>
void *CommonObjectDB<T>::BeginRead (StructureType structure) {
```

```
   DoubleBufferData *ThisDoubleBuffer;

   ThisDoubleBuffer = &ArenaControl->Data->DoubleBuffer[structure];

   if (!Registered( structure)){
      cerr << "CODB BeginRead(" << structure << ") Error in pid "
           << getpid() << "!\n";
      ReportArenaInfo(structure);
      exit(0);
   }

   // WAIT your turn to access ReadCount.
   ussetlock(ThisDoubleBuffer->ReadCountAccess);

      // BEGIN ReadCountAccess CRITICAL SECTION----------------------------
      ThisDoubleBuffer->ReadCount++;

      // If this is the first reader of a group, WAIT for WAITing writer to
      // finish swapping buffers.
      if (ThisDoubleBuffer->ReadCount == 1){
         ussetlock(ThisDoubleBuffer->ReadAccess);
      }//endif
      // END ReadCountAccess CRITICAL SECTION-----------------------------

   // Let others mess with ReadCount.
   usunsetlock(ThisDoubleBuffer->ReadCountAccess);

   return ThisDoubleBuffer->ReadBufferPtr;
}//end BeginRead


template <class T>
void CommonObjectDB<T>::EndRead(StructureType structure) {

   DoubleBufferData *ThisDoubleBuffer;

   ThisDoubleBuffer = &ArenaControl->Data->DoubleBuffer[structure];

   if (!Registered( structure)){
      cerr << "CODB EndRead(" << structure << ") Error in pid "
           << getpid() << "!\n";
      ReportArenaInfo(structure);
      exit(0);
   }

   // WAIT your turn to access ReadCount.
   ussetlock(ThisDoubleBuffer->ReadCountAccess);

      // BEGIN ReadCountAccess CRITICAL SECTION----------------------------
      ThisDoubleBuffer->ReadCount--;

      // If this is the last reader of the group, let any WAITing writer
      // start swapping buffers.
      if (ThisDoubleBuffer->ReadCount == 0){
         usunsetlock(ThisDoubleBuffer->ReadAccess);
      }//endif
      // END ReadCountAccess CRITICAL SECTION-----------------------------

   // Let others mess with ReadCount.
   usunsetlock(ThisDoubleBuffer->ReadCountAccess);

}//end EndRead


template <class T>
void *CommonObjectDB<T>::BeginWrite(StructureType structure) {

   DoubleBufferData *ThisDoubleBuffer;

   ThisDoubleBuffer = &ArenaControl->Data->DoubleBuffer[structure];

   if (!Registered( structure)){
      cerr << "CODB BeginWrite(" << structure << ") Error in pid "
           << getpid() << "!\n";
      ReportArenaInfo(structure);
```

```
      exit(0);
   }

   // WAIT for any other writers to finish up their business.
   ussetlock(ThisDoubleBuffer->WriteAccess);

   return ThisDoubleBuffer->WriteBufferPtr;
}//end BeginWrite


template <class T>
void CommonObjectDB<T>::EndWrite(StructureType structure) {

   // Temp pointer for swapping buffers.
   void            *temp;
   DoubleBufferData *ThisDoubleBuffer;

   ThisDoubleBuffer = &ArenaControl->Data->DoubleBuffer[structure];

   if (!Registered( structure)){
      cerr << "CODB EndWrite(" << structure << ") Error in pid "
           << getpid() << "!\n";
      ReportArenaInfo(structure);
      exit(0);
   }

   // Wait for any existing readers to finish up.
   ussetlock(ThisDoubleBuffer->ReadAccess);

      // BEGIN ReadAccess CRITICAL SECTION----------------------------------
      // Swap the pointers.
      temp                            = ThisDoubleBuffer->WriteBufferPtr;
      ThisDoubleBuffer->WriteBufferPtr = ThisDoubleBuffer->ReadBufferPtr;
      ThisDoubleBuffer->ReadBufferPtr  = temp;
      // END ReadAccess CRITICAL SECTION------------------------------------

   // Open up the floodgates and let the readers pour in.
   usunsetlock(ThisDoubleBuffer->ReadAccess);

   // Let other writers in to do their thing.
   usunsetlock(ThisDoubleBuffer->WriteAccess);
}//end EndWrite


template <class T>
void *CommonObjectDB<T>::BeginReadWrite(StructureType structure) {

   void            *ReadBuff;
   void            *WriteBuff;
   DoubleBufferData *ThisDoubleBuffer;

   ThisDoubleBuffer = &ArenaControl->Data->DoubleBuffer[structure];

   if (!Registered( structure)){
      cerr << "CODB BeginReadWrite(" << structure << ") Error in pid "
           << getpid() << "!\n";
      ReportArenaInfo(structure);
      exit(0);
   }

   WriteBuff = BeginWrite(structure);
   ReadBuff  = BeginRead(structure);

   memcpy(WriteBuff, ReadBuff, ThisDoubleBuffer->BufferSize);

   EndRead(structure);

   return WriteBuff;
}//end BeginReadWrite


template <class T>
void CommonObjectDB<T>::EndReadWrite(StructureType structure) {
```

```
   if (!Registered( structure)){
      cerr << "CODB EndReadWrite(" << structure << ") Error in pid "
           << getpid() << "!\n";
      ReportArenaInfo(structure);
      exit(0);
   }

   EndWrite(structure);
}; // end EndReadWrite


template <class T>
void *CommonObjectDB<T>::BeginMagic(StructureType structure) {

   return (BeginReadWrite(structure));

}; // end BeginMagic


template <class T>
void CommonObjectDB<T>::EndMagic (StructureType structure) {

   EndReadWrite(structure);

}; // end EndMagic


template <class T>
int CommonObjectDB<T>::Registered(StructureType structure){

   int ThisThingIsRegistered =
      !(ArenaControl->Data->DoubleBuffer[structure].ReadBufferPtr == NULL);

   return ThisThingIsRegistered;

}//end Registered(StructureType structure)


template <class T>
void CommonObjectDB<T>::ReportArenaInfo(StructureType structure){
   ofstream ErrLog;
   char     FileName[21];
   char     Buffer[20];

   sprintf( FileName, "CODBArena%06u.dat", (int)(getpid()));
   ErrLog.open(FileName, ios::out, 0664);
   ErrLog << "CODB Failure Report\n";
   ErrLog << "Structure that caused failure: " << structure << "\n";
   ErrLog << "Failure occurred in process " << getpid() << "\n";
   ErrLog << "CODB Arena Address: " << Arena << "\n";
   ErrLog << "BytesAllocated: " << ArenaControl->Data->BytesAllocated << "\n";

   ErrLog << "\nDoubleBuffer resides at address: "
          << ArenaControl->Data->DoubleBuffer << "\n";
   ErrLog << "Contents of DoubleBuffer array:\n";
   ErrLog << "Index ReadCount ReadBufferPtr WriteBufferPtr BufferSize\n";

   for (int i = 0; i < MaxDoubleBuffers; ++i){
      sprintf(Buffer,"%5i", i);
      ErrLog << Buffer;
      sprintf(Buffer,"%10i",
              ArenaControl->Data->DoubleBuffer[i].ReadCount);
      ErrLog << Buffer;
      sprintf(Buffer,"%14p",
              (void*)(ArenaControl->Data->DoubleBuffer[i].ReadBufferPtr));
      ErrLog << Buffer;
      sprintf(Buffer,"%15p",
              (void*)(ArenaControl->Data->DoubleBuffer[i].WriteBufferPtr));
      ErrLog << Buffer;
      sprintf(Buffer,"%11u",
              (void*)(ArenaControl->Data->DoubleBuffer[i].BufferSize));
      ErrLog << Buffer;
      ErrLog << "\n";
```

```
    }//next i
    ErrLog.close();

}//end ReportArenaInfo(StructureType structure)
```

Appendix C

Fuzzy Terms Sets and Rules for Flying Point To Point

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Filename:      wingclips.pt2pt.htm
;; Project:       Air Force Institute of Technology Fuzzy Wingman Thesis 1996
;; Functionality: Contains term sets and rules for flying point to point in
;;                FLTCntrl::PointToPoint(). Intended to be loaded on top of
;;                wingclips.fltcntrl.htm so that its common functions are
;;                available.
;; Author(s):     Capt Vincent Brian Zurita GCS 96-D
;;                1Lt James L Benslay GCS 96-D
;; Revision History:
;; 17 Sep 96, VBZ: Initial write.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;<A NAME="TERMSETS">
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;; FLTCntrl PointToPoint Fuzzy Term Sets
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;---------------------------------------------------------------------------
;; Term Set:      RelativelyX
;; Category:      Measurement Term Set
;; Semantic:      Target is ______ Wingman
;; Functionality: Measures relative X in NED coordinate system in angular
;;                fashion.
;; Assertion:     asin(Point->Coord.X/Distance(Point->Coord))*RAD_TO_DEG
;; Author(s):     Capt Vincent Brian Zurita GCS 96-D
;;                1Lt James L Benslay GCS 96-D
;; Revision History:
;; 17 Sep 96, VBZ: Initial write.
;; 08 Oct 96, VBZ: Changed from (-90 to 90) to (0 180), reversed order of terms.
;; 24 Oct 96, VBZ: Went back to asin, -90.0 90.0 degrees.

(deftemplate RelativelyX
   -90.0 90.0 degrees
   (
      (behind      (Z  -12.5  -2.5))
      (even_with   (PI   5.0   0.0))
      (in_front_of (S    2.5  12.5))
   )
)


;;---------------------------------------------------------------------------
;; Term Set:      RelativelyY
;; Category:      Measurement Term Set
;; Semantic:      Target is ______ Wingman
;; Functionality: Measures relative Y in NED coordinate system in angular
;;                fashion.
;; Assertion:     asin(Point->Coord.Y/Distance(Point->Coord))*RAD_TO_DEG
;; Author(s):     Capt Vincent Brian Zurita GCS 96-D
;;                1Lt James L Benslay GCS 96-D
;; Revision History:
;; 17 Sep 96, VBZ: Initial write.
;; 08 Oct 96, VBZ: Changed from (-90 to 90) to (0 180), reversed order of terms.
;; 24 Oct 96, VBZ: Went back to asin, -90.0 90.0 degrees.

(deftemplate RelativelyY
   -90.0 90.0 degrees
   (
      (left_of   (Z  -12.5  -2.5))
      (even_with (PI   5.0   0.0))
      (right_of  (S    2.5  12.5))
   )
)


;;---------------------------------------------------------------------------
;; Term Set:      RelativelyZ
```

```
;; Category:       Measurement Term Set
;; Semantic:       Target is _____ Wingman
;; Functionality: Measures relative Z in NED coordinate system in angular
;;                fashion.
;; Assertion:      asin(Point->Coord.Z/Distance(Point->Coord))*RAD_TO_DEG
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
;; Revision History:
;; 17 Sep 96, VBZ: Initial write.
;; 08 Oct 96, VBZ: Changed from (-90 to 90) to (0 180), reversed order of terms.
;; 24 Oct 96, VBZ: Went back to asin, -90.0 90.0 degrees.

(deftemplate RelativelyZ
   -90.0 90.0 degrees
   (
      (above_    (Z  -12.5  -2.5))
      (even_with (PI   5.0   0.0))
      (below_    (S    2.5  12.5))
   )
)


;;-------------------------------------------------------------------------
;; Term Set:       Roll
;; Category:       Measurement Term Set
;; Semantic:       Wingman is _____
;; Functionality: Measures absolute roll in AeroModel coordinate system.
;; Assertion:      Wingman's roll (phi) value in degrees.
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
;; Revision History:
;; 30 Oct 96, VBZ: Initial write.

(deftemplate Roll
   -180.0 180.0 degrees
   (
      (rolled_left  (Z  -180  -5))
      (level        (PI   10   0))
      (rolled_right (S     5 180))
   )
)


;;-------------------------------------------------------------------------
;; Term Set:       RelativelyHeading
;; Category:       Measurement Term Set
;; Semantic:       Target is heading _____ Wingman
;; Functionality: Measures relative heading in NED coordinate system in angular
;;                fashion.
;; Assertion:      Point->HPR.h
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
;; Revision History:
;; 17 Sep 96, VBZ: Initial write.

(deftemplate RelativelyHeading
   -180.0 180.0 degrees
   (
      (against      (Z  -180 -125))
      (to_the_left  (PI   55  -90))
      (with         (PI   55    0))
      (to_the_right (PI   55   90))
      (against_the  (S   125  180))
   )
)


;;-------------------------------------------------------------------------
;; Term Set:       RelativelyPitching
;; Category:       Measurement Term Set
;; Semantic:       Target is pitching _____ Wingman
;; Functionality: Measures relative pitch in NED coordinate system in angular
;;                fashion.
;; Assertion:      Point->HPR.p
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
```

```
;; Revision History:
;; 17 Sep 96, VBZ: Initial write.

(deftemplate RelativelyPitching
   -90.0 90.0 degrees
   (
      (down_to    (Z  -20 -10))
      (even_with (PI  15   0))
      (up_to      (S   10  20))
   )
)


;;---------------------------------------------------------------------------
;; Term Set:       RelativelyDist
;; Category:       Measurement Term Set
;; Semantic:       Target is _____ Wingman
;; Functionality: Measures relative distance in NED coordinate system in angular
;;                 fashion.
;; Assertion:      Distance(Point->Coord)
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
;; Revision History:
;; 17 Sep 96, VBZ: Initial write.

(deftemplate RelativelyDist
   0.0 1000000.0 feet
   (
      (on_top_of (Z     0 125))
      (near      (PI  200 300))
      (far_from  (S   475 600))
   )
)


;;---------------------------------------------------------------------------
;; Term Set:       StickLat
;; Category:       Control Term Set
;; Semantic:       _____ on the stick
;; Functionality: Provides stick latitude settings.  Stick latitude is the
;;                 movement from side to side that controls the roll of the
;;                 aircraft.
;; Assertion:      TAS_n rule firings.
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
;; Revision History:
;; 17 Sep 96, VBZ: Initial write.
;; 30 Oct 96, JLB: Reworked to narrow ranges.  After doing
;;                  some real-time analysis of the algorithm
;;                  run times, have decided on an initial
;;                  adjustment of 1/2 of present value.
;; 30 Oct 96, JLB: Still needs to be narrowed.  Go 1/2 more.
;; 30 Oct 96, JLB: Still needs to be narrowed.  Go 1/2 more.

(deftemplate StickLat
   -0.5 0.5 setting
   (
      (hard_left   (Z  -0.50 -0.25))
      (left        (PI  0.20 -0.25))
      (nudge_left  (PI  0.04 -0.06))
      (nil         (PI  0.04  0.00))
      (nudge_right (PI  0.04  0.06))
      (right       (PI  0.20  0.25))
      (hard_right  (S   0.25  0.50))
   )
)


;;---------------------------------------------------------------------------
;; Term Set:       StickLong
;; Category:       Control Term Set
;; Semantic:       _____ on the stick
;; Functionality: Provides stick longitude settings.  Stick longitude is the
;;                 movement from front to back that controls the pitch of the
;;                 aircraft.
;; Assertion:      TAS_n rule firings.
```

```
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
;; Revision History:
;; 17 Sep 96, VBZ: Initial write.
;; 30 Oct 96, JLB: Reworked to narrow ranges.  After doing
;;                 some real-time analysis of the algorithm
;;                 run times, have decided on an initial
;;                 adjustment of 1/2 of present value.
;; 30 Oct 96, JLB: By observation, seems to be working acceptably.

(deftemplate StickLong
   -1.0 1.0 setting
   (
      (hard_forward  (Z  -1.00 -0.60))
      (forward       (PI  0.32 -0.40))
      (nudge_forward (PI  0.04 -0.06))
      (nil           (PI  0.04  0.00))
      (nudge_back    (PI  0.04  0.06))
      (back          (PI  0.32  0.40))
      (hard_back     (S   0.60  1.00))
   )
)


;;-----------------------------------------------------------------------------
;; Term Set:       Rudder
;; Category:       Control Term Set
;; Semantic:       ______ on the rudder
;; Functionality: Provides rudder settings.  Rudder controls the yaw (heading)
;;                 of the aircraft.
;; Assertion:      TAS_n rule firings.
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
;; Revision History:
;; 17 Sep 96, VBZ: Initial write.

(deftemplate Rudder
   -1.0 1.0 setting
   (
      (hard_left   (Z  -1.00 -0.60))
      (left        (PI  0.32 -0.40))
      (nudge_left  (PI  0.04 -0.06))
      (nil         (PI  0.04  0.00))
      (nudge_right (PI  0.04  0.06))
      (right       (PI  0.32  0.40))
      (hard_right  (S   0.60  1.00))
   )
)


;;-----------------------------------------------------------------------------
;; Rule Set:       DefuzzStickLat
;; Category:       Defuzzification Rule
;; Functionality: Defuzzifies the StickLat value.
;; Assertion:      None (implicit).
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
;; Revision History:
;; 23 Sep 96, VBZ: Initial write.

(defrule DefuzzStickLat
   (declare (salience -10))
   ?controls <- (object (is-a FLIGHTCONTROLS))
   ?stick_lat <- (StickLat ?)
   =>
   (bind ?value (moment-defuzzify ?stick_lat))
   (send (instance-name ?controls) put-Stick-Lat-Safe ?value)
)


;;-----------------------------------------------------------------------------
;; Rule Set:       DefuzzStickLong
;; Category:       Defuzzification Rule
;; Functionality: Defuzzifies the StickLong value.
;; Assertion:      None (implicit).
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
```

```
;;                1Lt James L Benslay GCS 96-D
;; Revision History:
;; 23 Sep 96, VBZ: Initial write.

(defrule DefuzzStickLong
   (declare (salience -10))
   ?controls <- (object (is-a FLIGHTCONTROLS))
   ?stick_long <- (StickLong ?)
   =>
   (bind ?value (moment-defuzzify ?stick_long))
   (send (instance-name ?controls) put-Stick-Long-Safe ?value)
)


;;-------------------------------------------------------------------------
;; Rule Set:       DefuzzThrottle
;; Category:       Defuzzification Rule
;; Functionality: Defuzzifies the Throttle value.
;; Assertion:      None (implicit).
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
;; Revision History:
;; 23 Sep 96, VBZ: Initial write.

(defrule DefuzzThrottle
   (declare (salience -10))
   ?controls <- (object (is-a FLIGHTCONTROLS))
   ?throt <- (Throttle ?)
   =>
   (bind ?value (moment-defuzzify ?throt))
   (send (instance-name ?controls) put-Throttle-Safe ?value)
)


;;-------------------------------------------------------------------------
;; Rule Set:       DefuzzRudder
;; Category:       Defuzzification Rule
;; Functionality: Defuzzifies the Rudder value.
;; Assertion:      None (implicit).
;; Author(s):      Capt Vincent Brian Zurita GCS 96-D
;;                 1Lt James L Benslay GCS 96-D
;; Revision History:
;; 23 Sep 96, VBZ: Initial write.

(defrule DefuzzRudder
   (declare (salience -10))
   ?controls <- (object (is-a FLIGHTCONTROLS))
   ?rudder <- (Rudder ?)
   =>
   (bind ?value (moment-defuzzify ?rudder))
   (send (instance-name ?controls) put-Rudder-Safe ?value)
)

(defrule TAS_1
   (RelativelyX behind)
   (RelativelyY left_of)
   (RelativelyZ below_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_forward))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE out))
)


(defrule TAS_2
   (RelativelyX behind)
   (RelativelyY left_of)
   (RelativelyZ below_)
   (RelativelyDist near)
   =>
   (assert (StickLat hard_left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_3
   (RelativelyX behind)
   (RelativelyY left_of)
   (RelativelyZ below_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat hard_left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_4
   (RelativelyX behind)
   (RelativelyY left_of)
   (RelativelyZ even_with)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nudge_left))
   (assert (StickLong nil))
```

```
   (assert (Rudder nil))
   (assert (SPEED_BRAKE out))
)


(defrule TAS_5
   (RelativelyX behind)
   (RelativelyY left_of)
   (RelativelyZ even_with)
   (RelativelyDist near)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_6
   (RelativelyX behind)
   (RelativelyY left_of)
   (RelativelyZ even_with)
   (RelativelyDist far_from)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_7
   (RelativelyX behind)
   (RelativelyY left_of)
   (RelativelyZ above_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_8
   (RelativelyX behind)
   (RelativelyY left_of)
   (RelativelyZ above_)
   (RelativelyDist near)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_9
   (RelativelyX behind)
   (RelativelyY left_of)
   (RelativelyZ above_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_10
   (RelativelyX behind)
   (RelativelyY even_with)
   (RelativelyZ below_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_forward))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE out))
)


(defrule TAS_11
   (RelativelyX behind)
   (RelativelyY even_with)
   (RelativelyZ below_)
   (RelativelyDist near)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_12
   (RelativelyX behind)
   (RelativelyY even_with)
   (RelativelyZ below_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_13
   (RelativelyX behind)
   (RelativelyY even_with)
   (RelativelyZ even_with)
   (ROLL rolled_left)
   =>
   (assert (StickLat nudge_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE out))
)


(defrule TAS_14
   (RelativelyX behind)
   (RelativelyY even_with)
   (RelativelyZ even_with)
   (ROLL level)
   =>
   (assert (StickLat nil))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_15
   (RelativelyX behind)
   (RelativelyY even_with)
   (RelativelyZ even_with)
   (ROLL rolled_right)
   =>
   (assert (StickLat nudge_left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_16
   (RelativelyX behind)
   (RelativelyY even_with)
   (RelativelyZ above_)
```

```
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_17
   (RelativelyX behind)
   (RelativelyY even_with)
   (RelativelyZ above_)
   (RelativelyDist near)
   =>
   (assert (StickLat nil))
   (assert (StickLong back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_18
   (RelativelyX behind)
   (RelativelyY even_with)
   (RelativelyZ above_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat nil))
   (assert (StickLong hard_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_19
   (RelativelyX behind)
   (RelativelyY right_of)
   (RelativelyZ below_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_forward))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE out))
)


(defrule TAS_20
   (RelativelyX behind)
   (RelativelyY right_of)
   (RelativelyZ below_)
   (RelativelyDist near)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_21
   (RelativelyX behind)
   (RelativelyY right_of)
   (RelativelyZ below_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_22
   (RelativelyX behind)
   (RelativelyY right_of)
   (RelativelyZ even_with)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nudge_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE out))
)


(defrule TAS_23
   (RelativelyX behind)
   (RelativelyY right_of)
   (RelativelyZ even_with)
   (RelativelyDist near)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_24
   (RelativelyX behind)
   (RelativelyY right_of)
   (RelativelyZ even_with)
   (RelativelyDist far_from)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_25
   (RelativelyX behind)
   (RelativelyY right_of)
   (RelativelyZ above_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_26
   (RelativelyX behind)
   (RelativelyY right_of)
   (RelativelyZ above_)
   (RelativelyDist near)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_27
   (RelativelyX behind)
   (RelativelyY right_of)
   (RelativelyZ above_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)
```

```
(defrule TAS_28
   (RelativelyX even_with)
   (RelativelyY left_of)
   (RelativelyZ below_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_forward))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_29
   (RelativelyX even_with)
   (RelativelyY left_of)
   (RelativelyZ below_)
   (RelativelyDist near)
   =>
   (assert (StickLat hard_left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_30
   (RelativelyX even_with)
   (RelativelyY left_of)
   (RelativelyZ below_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat hard_left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_31
   (RelativelyX even_with)
   (RelativelyY left_of)
   (RelativelyZ even_with)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nudge_left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_32
   (RelativelyX even_with)
   (RelativelyY left_of)
   (RelativelyZ even_with)
   (RelativelyDist near)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_33
   (RelativelyX even_with)
   (RelativelyY left_of)
   (RelativelyZ even_with)
   (RelativelyDist far_from)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_34
   (RelativelyX even_with)
   (RelativelyY left_of)
   (RelativelyZ above_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_35
   (RelativelyX even_with)
   (RelativelyY left_of)
   (RelativelyZ above_)
   (RelativelyDist near)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_36
   (RelativelyX even_with)
   (RelativelyY left_of)
   (RelativelyZ above_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_37
   (RelativelyX even_with)
   (RelativelyY even_with)
   (RelativelyZ below_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_forward))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_38
   (RelativelyX even_with)
   (RelativelyY even_with)
   (RelativelyZ below_)
   (RelativelyDist near)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)

(defrule TAS_39
   (RelativelyX even_with)
   (RelativelyY even_with)
   (RelativelyZ below_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
```

```
   (assert (SPEED_BRAKE in))
)


(defrule TAS_40
   (RelativelyX even_with)
   (RelativelyY even_with)
   (RelativelyZ even_with)
   (ROLL rolled_left)
   =>
   (assert (StickLat nudge_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_41
   (RelativelyX even_with)
   (RelativelyY even_with)
   (RelativelyZ even_with)
   (ROLL level)
   =>
   (assert (StickLat nil))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_42
   (RelativelyX even_with)
   (RelativelyY even_with)
   (RelativelyZ even_with)
   (ROLL rolled_right)
   =>
   (assert (StickLat nudge_left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_43
   (RelativelyX even_with)
   (RelativelyY even_with)
   (RelativelyZ above_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_44
   (RelativelyX even_with)
   (RelativelyY even_with)
   (RelativelyZ above_)
   (RelativelyDist near)
   =>
   (assert (StickLat nil))
   (assert (StickLong back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_45
   (RelativelyX even_with)
   (RelativelyY even_with)
   (RelativelyZ above_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat nil))
   (assert (StickLong hard_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_46
   (RelativelyX even_with)
   (RelativelyY right_of)
   (RelativelyZ below_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_forward))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_47
   (RelativelyX even_with)
   (RelativelyY right_of)
   (RelativelyZ below_)
   (RelativelyDist near)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_48
   (RelativelyX even_with)
   (RelativelyY right_of)
   (RelativelyZ below_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_49
   (RelativelyX even_with)
   (RelativelyY right_of)
   (RelativelyZ even_with)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nudge_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_50
   (RelativelyX even_with)
   (RelativelyY right_of)
   (RelativelyZ even_with)
   (RelativelyDist near)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_51
   (RelativelyX even_with)
   (RelativelyY right_of)
   (RelativelyZ even_with)
   (RelativelyDist far_from)
```

```
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_52
   (RelativelyX even_with)
   (RelativelyY right_of)
   (RelativelyZ above_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_53
   (RelativelyX even_with)
   (RelativelyY right_of)
   (RelativelyZ above_)
   (RelativelyDist near)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_54
   (RelativelyX even_with)
   (RelativelyY right_of)
   (RelativelyZ above_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_55
   (RelativelyX in_front_of)
   (RelativelyY left_of)
   (RelativelyZ below_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_forward))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_56
   (RelativelyX in_front_of)
   (RelativelyY left_of)
   (RelativelyZ below_)
   (RelativelyDist near)
   =>
   (assert (StickLat hard_left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_57
   (RelativelyX in_front_of)
   (RelativelyY left_of)
   (RelativelyZ below_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat hard_left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_58
   (RelativelyX in_front_of)
   (RelativelyY left_of)
   (RelativelyZ even_with)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_59
   (RelativelyX in_front_of)
   (RelativelyY left_of)
   (RelativelyZ even_with)
   (RelativelyDist near)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_60
   (RelativelyX in_front_of)
   (RelativelyY left_of)
   (RelativelyZ even_with)
   (RelativelyDist far_from)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_61
   (RelativelyX in_front_of)
   (RelativelyY left_of)
   (RelativelyZ above_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_62
   (RelativelyX in_front_of)
   (RelativelyY left_of)
   (RelativelyZ above_)
   (RelativelyDist near)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_63
```

```
   (RelativelyX in_front_of)
   (RelativelyY left_of)
   (RelativelyZ above_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_64
   (RelativelyX in_front_of)
   (RelativelyY even_with)
   (RelativelyZ below_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_forward))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_65
   (RelativelyX in_front_of)
   (RelativelyY even_with)
   (RelativelyZ below_)
   (RelativelyDist near)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_66
   (RelativelyX in_front_of)
   (RelativelyY even_with)
   (RelativelyZ below_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_67
   (RelativelyX in_front_of)
   (RelativelyY even_with)
   (RelativelyZ even_with)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nudge_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_68
   (RelativelyX in_front_of)
   (RelativelyY even_with)
   (RelativelyZ even_with)
   (RelativelyDist near)
   =>
   (assert (StickLat nil))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_69
   (RelativelyX in_front_of)
   (RelativelyY even_with)
   (RelativelyZ even_with)
   (RelativelyDist far_from)
   =>
   (assert (StickLat nudge_left))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_70
   (RelativelyX in_front_of)
   (RelativelyY even_with)
   (RelativelyZ above_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_71
   (RelativelyX in_front_of)
   (RelativelyY even_with)
   (RelativelyZ above_)
   (RelativelyDist near)
   =>
   (assert (StickLat nil))
   (assert (StickLong back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_72
   (RelativelyX in_front_of)
   (RelativelyY even_with)
   (RelativelyZ above_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat nil))
   (assert (StickLong hard_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_73
   (RelativelyX in_front_of)
   (RelativelyY right_of)
   (RelativelyZ below_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_forward))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_74
   (RelativelyX in_front_of)
   (RelativelyY right_of)
   (RelativelyZ below_)
   (RelativelyDist near)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
```

```
)


(defrule TAS_75
   (RelativelyX in_front_of)
   (RelativelyY right_of)
   (RelativelyZ below_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat hard_right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_76
   (RelativelyX in_front_of)
   (RelativelyY right_of)
   (RelativelyZ even_with)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_77
   (RelativelyX in_front_of)
   (RelativelyY right_of)
   (RelativelyZ even_with)
   (RelativelyDist near)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_78
   (RelativelyX in_front_of)
   (RelativelyY right_of)
   (RelativelyZ even_with)
   (RelativelyDist far_from)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_79
   (RelativelyX in_front_of)
   (RelativelyY right_of)
   (RelativelyZ above_)
   (RelativelyDist on_top_of)
   =>
   (assert (StickLat nil))
   (assert (StickLong nudge_back))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_80
   (RelativelyX in_front_of)
   (RelativelyY right_of)
   (RelativelyZ above_)
   (RelativelyDist near)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)


(defrule TAS_81
   (RelativelyX in_front_of)
   (RelativelyY right_of)
   (RelativelyZ above_)
   (RelativelyDist far_from)
   =>
   (assert (StickLat right))
   (assert (StickLong nil))
   (assert (Rudder nil))
   (assert (SPEED_BRAKE in))
)
```

Appendix D
Point To Point Data Outputs

```
--------------------------------------------------------------------------------
.ini file particulars for all runs:
initial x: 65000
initial y: 20000
initial z: -40000
initial heading: 90
initial mach: 0.75


--------------------------------------------------------------------------------
This set of runs was to test the effect of scaling RelativelyDist by 1/10.
(deftemplate RelativelyX
   -90.0 90.0 degrees
   (
      (behind      (Z  -12.5  -2.5))
      (even_with   (PI   5.0   0.0))
      (in_front_of (S    2.5  12.5))
   )
)
(deftemplate RelativelyY
   -90.0 90.0 degrees
   (
      (left_of   (Z  -12.5  -2.5))
      (even_with (PI   5.0   0.0))
      (right_of  (S    2.5  12.5))
   )
)
(deftemplate RelativelyZ
   -90.0 90.0 degrees
   (
      (above_    (Z  -12.5  -2.5))
      (even_with (PI   5.0   0.0))
      (below_    (S    2.5  12.5))
   )
)
(deftemplate RelativelyDist
   0.0 1000000.0 feet
   (
      (on_top_of (Z     0 125))
      (near      (PI  200 300))
      (far_from  (S   475 600))
   )
)
bfm.DiamondTest:
 60000.0   -30000.0      0.0   0.0   0.0   0.0
120000.0    30000.0      0.0   0.0   0.0   0.0
180000.0   -30000.0      0.0   0.0   0.0   0.0
240000.0    30000.0      0.0   0.0   0.0   0.0

yielded:

FLTCntrl receiving points:
(95018,80805.3,-39976.7)
(35035.7,140823,-39976.7)
(95053.4,200805,-39976.7)
(35071.1,260823,-39976.7)
   At (95252.3,80912.5,-39796.6),
Wingman has passed within 299 ft of (95018, 80805.3, -39976.7)
   At (34942.6,141081,-39804.9),
Wingman has passed within 294 ft of (35035.7, 140823, -39976.7)
   At (95303.1,200816,-39792.8),
Wingman has passed within 304 ft of (95053.4, 200805, -39976.7)
   At (34962.3,261092,-39800.7),
Wingman has passed within 303 ft of (35071.1, 260823, -39976.7)

FLTCntrl receiving points:
(95018.7,83410.8,-39903.9)
(35036.4,143428,-39903.9)
(95054.1,203411,-39903.9)
(35071.8,263428,-39903.9)
   At (95276.9,83506.6,-39743.5),
```

```
Wingman has passed within 291 ft of (95018.7, 83410.8, -39903.9)
   At (35006.6,143651,-39699.3),
Wingman has passed within 299 ft of (35036.4, 143428, -39903.9)
   At (95304.9,203425,-39724.5),
Wingman has passed within 291 ft of (95054.1, 203411, -39903.9)
   At (34995,263676,-39718.3),
Wingman has passed within 299 ft of (35071.8, 263428, -39903.9)

FLTCntrl receiving points:
(95018.4,82512.2,-39943)
(35036.1,142530,-39943)
(95053.8,202512,-39943)
(35071.6,262530,-39943)
   At (95245.1,82570.2,-39757.8),
Wingman has passed within 287 ft of (95018.4, 82512.2, -39943)
   At (34909.5,142804,-39785.6),
Wingman has passed within 295 ft of (35036.1, 142530, -39943)
   At (95338.9,202568,-39782.7),
Wingman has passed within 292 ft of (95053.8, 202512, -39943)
   At (34954.1,262784,-39755.7),
Wingman has passed within 298 ft of (35071.6, 262530, -39943)

FLTCntrl receiving points:
(95018.1,81318.1,-39982.3)
(35035.8,141336,-39982.3)
(95053.5,201318,-39982.3)
(35071.2,261336,-39982.3)
   At (95268.1,81419,-39821.5),
Wingman has passed within 287 ft of (95018.1, 81318.1, -39982.3)
   At (34988.1,141569,-39791.2),
Wingman has passed within 294 ft of (35035.8, 141336, -39982.3)
   At (95277.9,201300,-39785.6),
Wingman has passed within 294 ft of (95053.5, 201318, -39982.3)
   At (34970,261599,-39804.7),
Wingman has passed within 302 ft of (35071.2, 261336, -39982.3)

FLTCntrl receiving points:
(95018.7,83684.8,-39890.7)
(35036.5,143703,-39890.7)
(95054.2,203685,-39890.7)
(35071.9,263703,-39890.7)
   At (95249.4,83763.4,-39713),
Wingman has passed within 285 ft of (95018.7, 83684.8, -39890.7)
   At (34873,143997,-39748.4),
Wingman has passed within 296 ft of (35036.5, 143703, -39890.7)
   At (95373.9,203781,-39749.7),
Wingman has passed within 294 ft of (95054.2, 203685, -39890.7)
   At (34908.2,263998,-39740),
Wingman has passed within 311 ft of (35071.9, 263703, -39890.7)
```

Bibliography

[ACC92] Air Combat Command Manual 3-3 Volume XVII, *Mission Combat Aircraft Fundamentals, Fighter Fundamentals, F-15E*, HQ ACC/DOT, July 1992.

[ADAM96] Adams, Terry A., AFIT/GCS/ENG/96D-02 Requirements, Design, and Development of a Rapidly Reconfigurable, Photo-Realistic, Virtual Cockpit Prototype

[BOOC83] Booch, Grady. Software Engineering with Ada. The Benjamin/Cummings Publishing Company. Menlo Park, CA, 1983.

[CERA95] Ceranowicz, A., "Modular Semi-Automated Forces", Loral Advanced Distributed Simulation, Cambridge, MA, 1995

[HARV96] Edward P. Harvey, BMH Associates, Inc. Norfolk, Virginia, "AIR SYNTHETIC FORCE DEVELOPMENT", http://www.bmh.com/bmh/final_paper.html, 7 Jun 96

[DMA1] "Department of Defense World Geodetic System 1984, Its Definition and Relationships with Local Geodetic Systems," Second Edition. Stock number DMATR83502WGS84. DMA Combat Support Center, 1 September 1991.

[DMSO96] "DoD High Level Architecture (HLA)", http://www.dmso.mil/projects/hla/, 6 Nov 96.

[DOD95] DoD Modeling and Simulation Master Plan (DoD 5000.59-P, dated October 1995) http://www.dmso.mil/docslib/mspolicy/msmp/

[GOMA93] Gomaa, Hassan. *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley Publishing Company, 1993.

[KOUP96] L. Koup, N. Charity, P. Nielsen, P. Stiffler, M. Tanner, F. Rogers, H. Johnson, Houston Associates, Inc. (HAI), "DSI Overview", http://www.les.mil/aitsjpo/dsi/overview.htm.

[IEEE93] Standards Coordinating Committee of the IEEE Computer Society. "IEEE Standard for Information Technology - Protocols for Distributed Interactive Simulation Applications". Technical Report IEEE Std 1278-1993, March 1993.

[IST94] Institute for Simulation and Training, IST-SP-94-01, "The DIS Vision -- A Map to the Future of Distributed Simulation", Institute for Simulation and Training, Orlando, Florida, IST-SP-94-01, May 1994.

[IST94a] Institute for Simulation and Training, IST-CR-93-40, "Standard for Distributed Interactive Simulation--Application Protocols Version 2.0 Fourth Draft", Institute for Simulation and Training, Orlando, Florida, March 1994.

[IST95] "Enumeration and Bit Encoded Values for Use with Protocols for Distributed Interactive Simulation Applications". IST-CR-95-14, Institute for Simulation and Training, Orlando, Florida. March 1995.

[IST96] DMSO Definitions of Various Types of Models, http://OTT.sc.ist.ucf.edu/1_3/definit.htm, October 1996.

[IST96a] "DIS Standards - Background", http://STDS.sc.ist.ucf.edu/info.htm, 18 June 1996.

[IST96b] "Standard for Distributed Interactive Simulations--Application Protocols". IST-CR-96-11, Institute for Simulation and Training, Orlando, Florida. March 1996.

[LAIR91] Laird, John E. and Paul S. Rosenbloom. "Integrating Execution, Planning, and Learning in Soar for External Environments". *Proceedings of the National Conference on Artificial Intelligence*, pp 1022-1029, Menlo Park, CA, 1991. AAAI Press.

[LAIR95] Laird, John E. et al. "Simulated Intelligent Forces for Air: The Soar/IFOR Project 1995". *Proceedings of the Fifth Conference on Computer Generated Forces and Behavioral Representation*, pp 27-36, Orlando, FL, May 1995.

[ROSE91] Rosenbloom, Paul S., John E. Laird, Allen Newell, and Robert McCarl. "A Preliminary Analysis of the SOAR Architecture as a Basis for General Intelligence". *Artificial Intelligence*, 47:289-325, 1991.

[SANT96] Santos, Eugene Jr., Sheila B. Banks, Martin R. Stytz. "Engineering Intelligent Computer Generated Forces". Technical Report, Air Force Institute of Technology, Wright-Patterson Air Force Base, OH, August 1996.

[SILI91] Silicon Graphics, Inc. *Parallel Processing on Silicon Graphics Computer Systems*. Document Number 007-0770-020. Silicon Graphics, Inc, Mountain View CA, 1991.

[STYT96] Stytz, Martin R., "Distributed Virtual Environments", *IEEE Computer Graphics and Applications*, 16(3):19-31, 1996.

[STYT97] Stytz, Martin, Terry Adams, Brian Garcia, Steven Sheasby, and Brian Zurita, "Developments in Rapid Prototyping and Software Architecture for Distributed Virtual Environments," *IEEE Software*, 1997.

[THOR88] Thorpe, J., "Warfighting with SIMNET--A Report From the Front," *Proceedings of the 10th Interservice/Industry Training Systems Conference*, Orlando, FL, 1988.

Vita

Capt Vincent Brian Zurita [redacted] He graduated magna cum laude from McKendree College in December 1991 with a Bachelor of Science degree in Computer Science. In 1992 he was commissioned a second lieutenant in the U.S. Air Force. He attended AFIT from May 1995 to December 1996, earning a Master of Science degree in Computer Systems.

Capt Zurita is married to the former Leslie M. Coco of Darien, Illinois. They have two children, Amy and Anthony.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1996	Master's Thesis

4. TITLE AND SUBTITLE
A Software Architecture For Computer Generated Forces In Complex Distributed Virtual Environments

5. FUNDING NUMBERS

6. AUTHOR(S)
Vincent Brian Zurita
Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/96D-32

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Electronic Systems Command
ESC/AVM
20 Schilling Circle
Hanscom AFB, MA 01731-2816

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Complex Distributed Virtual Environments (DVEs) present an outstanding opportunity for the Department of Defense to train geographically separated units within a single realistic threat environment with minimal logistical considerations or safety concerns. To increase the fidelity of these simulations, minimize cost, and thereby maximize the training potential, DVEs must be populated with a realistic number of Computer Generated Forces (CGFs). These are currently expensive to design and build due to a lack of standard CGF architectures. A solution to this problem is presented in the form of a CGF Architecture that is applicable to CGFs that model any weapon system. Mapping techniques are discussed that take the architecture from generic templates to weapon system-specific templates ready for implementation. An application based on this architecture, the Fuzzy Wingman, is discussed and its results are presented.

14. SUBJECT TERMS
Distributed Interactive Simulation, Computer Generated Forces, Semi-Automated Forces (SAFOR), Distributed Virtual Environment, Fuzzy Logic, Software Architecture

15. NUMBER OF PAGES
103

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

★U.S.GPO:1993-0-336-043