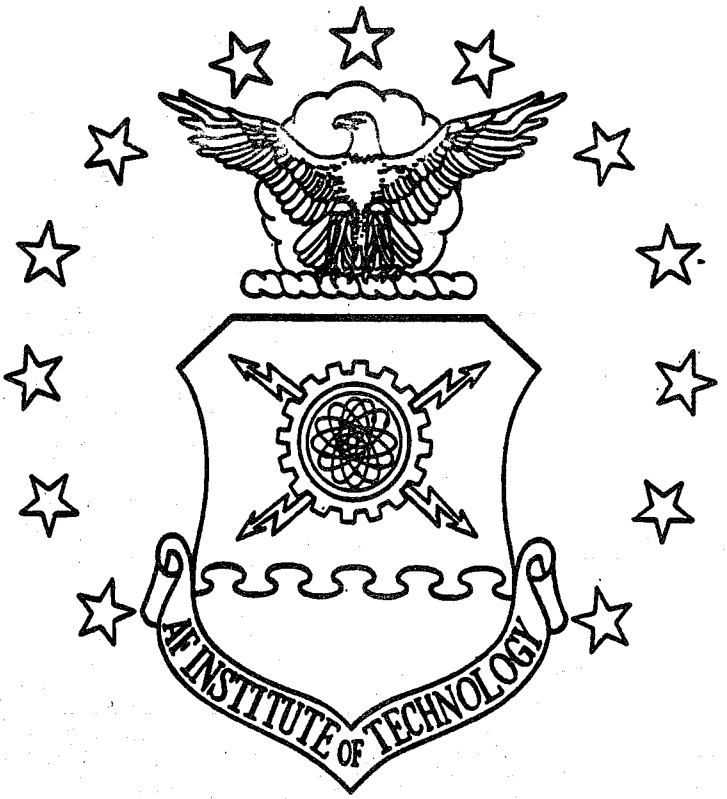

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/95D-24

Dynamic Transfer of Control Between Manned and Unmanned Simulation Actors

THESIS

Neal Wayne Schneider
Captain, USAF

AFIT/GE/ENG/95D-24

Approved for public release; distribution unlimited

AFIT/GE/ENG/95D-24

Dynamic Transfer of Control Between Manned and Unmanned Simulation Actors

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 (Computer Science)
Neal W. Schneider, B. S.
Captain, USAF
December, 1995

Acknowledgments

I would like to thank first and foremost the most important person in the world to me, my wife, Vivian. This would not have been possible without her help, endless love, and much needed encouragement. A special thanks goes to all of my fellow sufferers in the graphics program, Mark Edwards and Lynda Myers. Another special thanks goes to a veteran of the Graphics lab, Steven Sheasby, who was instrumental in the work on the Object Manager and Distributed Simulation. Finally, I must thank a long line of family, friends, faculty, and fellow believers who made this a great experience.

Table of Contents

Table of Figures

Abstract

This thesis continues the ongoing research at the Air Force Institute of Technology's Virtual Environments Laboratory in the area of distributed simulation. As the relevance and interest of interactive simulation as a training medium continues to grow, there is a pressing need to provide more realistic and numerous intelligent autonomous agents for simulations. As those autonomous agents mature and become more realistic, the need exists to be able to handle individual agents by taking control of them and operating them as manned agents at certain points within the simulation. The author started with a protocol proposed in a working draft of the Distributed Interactive Simulation (DIS) Protocol Standard 2.1.1 (Draft). He demonstrates how this protocol can be improved by swapping control between two entities involved in a distributed simulation. The new protocol provides simultaneous transfer while being compatible with the one proposed in the draft standard.

The protocol is implemented on two applications developed in the Virtual Environments Laboratory, the Virtual Cockpit (VC) and the Automated Wingman (AW). The anticipated flow of execution begins with the AW requesting assistance. The operator of the VC then can reply by assuming control of the AW. Once the required human operation has been performed, the operator may switch back to the lead aircraft, completing the full cycle of execution.

1 Introduction

The Department of Defense has made a substantial investment in the development of interactive training simulators. The Air Force Institute of Technology (AFIT) has played an active role in the continuing research and development of applications to enhance these interactive simulations. One such area has been the development of the Virtual Cockpit. The Virtual Cockpit (VC) application was first developed at AFIT in 1991 [SWIT92] and has evolved into a realistic, interactive immersive simulation.

One of the present goals for the VC is to reduce the overall cost of simulation systems. In fact, present simulators are quite expensive and immobile. For example, Wright Laboratory's simulation facility has cost the Air Force over $40 million dollars. The facility's very specialized mission is to perform avionics integration testing, thereby serving a critical role [CHRI95]. Since it is not possible to have many such facilities, the VC is being researched as a lower cost alternative. At a small fraction of the cost, AFIT's VC also implements many the features necessary for successful training. Unfortunately, the VC is limited in several respects. Most notably, it does not have full fidelity like large scale simulators, nor does it provide haptic (gross motion) feedback through the controls, or cockpit motion.

In addition to the cost of the individual simulators, Congress has mandated that the Department of Defense be able to support distributed

interactive simulation capable of supporting 100,000 entities [ROGE94]. To meet this goal, the Department of Defense must multiply its simulation capabilities.

In order to increase the number of flying entities in a simulation, AFIT has developed the Automated Wingman. The Automated Wingman is an intelligent entity that takes cues from another entity on the network, a lead aircraft, and operates on a fuzzy logic inferencing engine to perform normal flying tasks [EDWA95]. When the ability of the Automated Wingman is overloaded or needs assistance, the lead pilot may intervene and take control of the wingman's entity.

The focus of this research effort has been to incorporate this switching capability into the applications at AFIT. This research has been tightly coupled with the ongoing research on the Automated Wingman.

1.1 Problem Statement.

Given the aforementioned requirements, the need for a general capability to switch between manned and unmanned simulators exists. The goal of this research is to give the operators of AFIT's Virtual Cockpit and Automated Wingman the ability to switch between different entities in a distributed interactive simulation to facilitate semi-autonomous airborne forces.

2 Background

2.1 Distributed Simulation.

A distributed interactive simulation has the capability to be distributed globally, across computers and networks from different geographic areas, and from completely different simulation centers. Being distributed in nature, there is no central host. Each machine involved has its own portrayal of the state of the simulation. Communication is typically over a best-effort packet based network. This form of networking has a smaller overhead than guaranteed delivery and requires less bandwidth [DIS94][GARD93].

Distributed Interactive Simulation (DIS) is a communications protocol for the orchestration of a simulation over a wide area network. DIS has been adopted as one of several protocols for interactive simulations for the DoD. The protocol defines the terms and concepts involved in operating such a simulation.

A DIS-compliant application is software on a host computer that communicates via a DIS simulation network. The application controls the state and actions of at least one simulation entity. Simulation entities such as tanks, aircraft, targets, or personnel are actors in the simulation environment and may be either controlled by a person or the application. An entity controlled by the application is considered as an autonomous force, or a computer-generated force. Alternatively, the entity may be controlled directly by a person participating in the simulation. In latter case, the application responds by rendering the other

entities in the simulation in its own view of the world. The application acts like the device the person is operating, whether it be a tank, aircraft or ship [BELL93].

Simulation applications that have been developed at AFIT interface with the DIS protocols using the Object Manager. The Object Manager is an AFIT-developed library of C++ classes that have encapsulated the details of the network protocols. Object Manager manages a few other entity related issues including entity lists, position data, and dead-reckoning [SHEA92].

Dead reckoning is the method DIS uses to reduce the amount of information required to be sent over the network. Rather than send only a position update of an entity for each frame of the simulation, the application sends both the position and velocity. The host that is rendering the entity interpolates between position updates, giving the appearance of smooth motion. Both the transmitting and receiving hosts run the dead-reckoning algorithm. The transmitting host continuously compares the entity's true position with where other hosts would believe its entity should be located. When the difference between these two values reaches a specified error tolerance, a new position update is transmitted [DIS94]. The DIS standard also specifies that all entities must transmit a Protocol Data Unit (PDU) at least every five seconds.

The DIS protocol uses a broadcast User Datagram Protocol (UDP). The protocol is defined in the Internet Engineering Task Force's (IETF) request for comments [POST80]. According to the DIS protocol, the receiving hosts are not required to check the source of the packets. This can be used to seamlessly

change the host that maintains the state of an entity. Since the source of the PDUs are not checked, changes in the physical host that broadcasts the entity's state PDUs will not be noticed by other applications on the network. The network communication that would need to take place to ensure the proper transition of control is not yet part of the approved standard and would take place outside the DIS protocol architecture. Development of this communication protocol was a major component of my research and will be discussed in the following pages

2.2 DIS 2.1.1(Draft) Recommendations Entity Handoff.

The Distributed Interactive Simulation Standard has a recommended technique to transfer control of an entity from one application to another in the DIS 2.1.1 draft version. This protocol is called the entity handoff protocol (EHP) in the standard. EHP will be described briefly, and its shortcomings for the swap of entities will be demonstrated.

The recommended technique for EHP is a one-way transfer of control from the application that is currently broadcasting the status of the entity to a management host. For this discussion, the application that gives up control of the entity will be called the "providing application," and the management application that requests control of the entity will be called the "requesting application."

The general exchange of messages within the DIS-defined entity hand-off protocol is shown in Figure 1. All communication on DIS is carried by Protocol

Data Units (PDU). In general, only "best effort" datagram delivery is required of the network, so there is always the potential for dropped or out-of-order PDUs. Further information is given in [POST80] and [DIS94].

The transfer is implemented with three different PDUs: the Transfer Control Request, Transfer Control, and Transfer Control Acknowledge PDUs. The requesting application initiates the transfer by issuing a Transfer Control

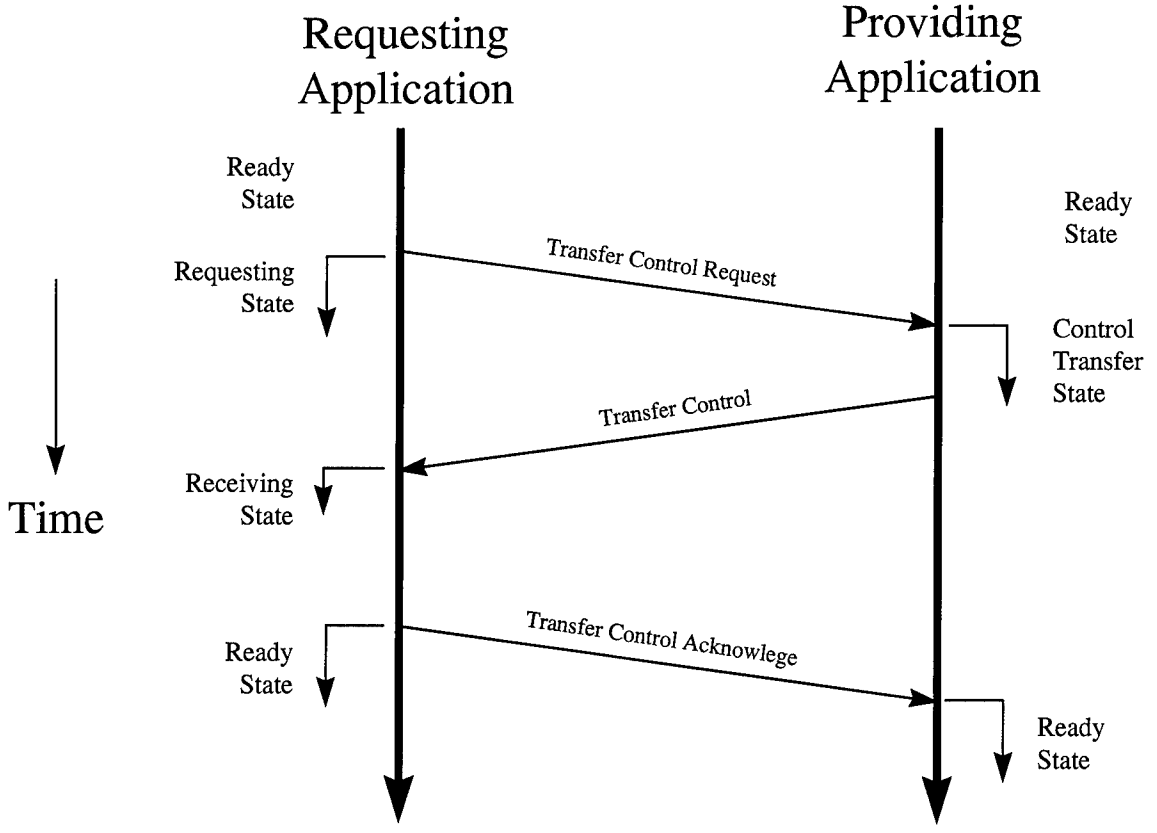

Figure 1: DIS Recommended Entity Hand-Off Protocol

Request PDU. It specifies the requested entity's identity: its site, application number, and entity identifier. Once received, the receiving application transmits

a Transfer Control PDU that relays the state of the entity. The Transfer Control PDU contains the position, orientation, linear and angular velocities of the original entity. In addition, the Transfer Control PDU provides space to transmit additional internal information of the entity. The Transfer Acknowledge PDU closes the loop and informs the original application of the new entity's identifier.

2.3 Coordinate Transforms.

There are many coordinate transformations that are made each computation cycle within the VC. All of these transformations must be used and understood for the VC to execute a swap with any entity on the network. An overview of the different systems is given in Figure 2. The normal flow of calculations proceeds from right to left. The aerodynamics model that computes the active forces and current positional data performs its calculations in the aircraft body coordinate (ABC) system. The conversion between the ABC system and the flat world is handled internally within the aerodynamic model. This transformation is represented by the arc labeled "Quaternion." More information about this transformation and the operation of the aerodynamics model can be found in [COOK92]. This transformation is actually not needed, and will most likely removed in future modifications of the aerodynamic model. The flat world coordinate system is an artifact from the original source of the aerodynamic model. The display of all of the objects in the visual system begins in the Performer coordinate system. Fortunately, the transform between the flat

world and Performer is a fixed axis translation, and the calculations reduce to swapping the X and Y coordinates and negating the Z.

The final coordinate transformation is executed just prior to transmitting the location onto the network. To transmit onto a DIS simulation, the

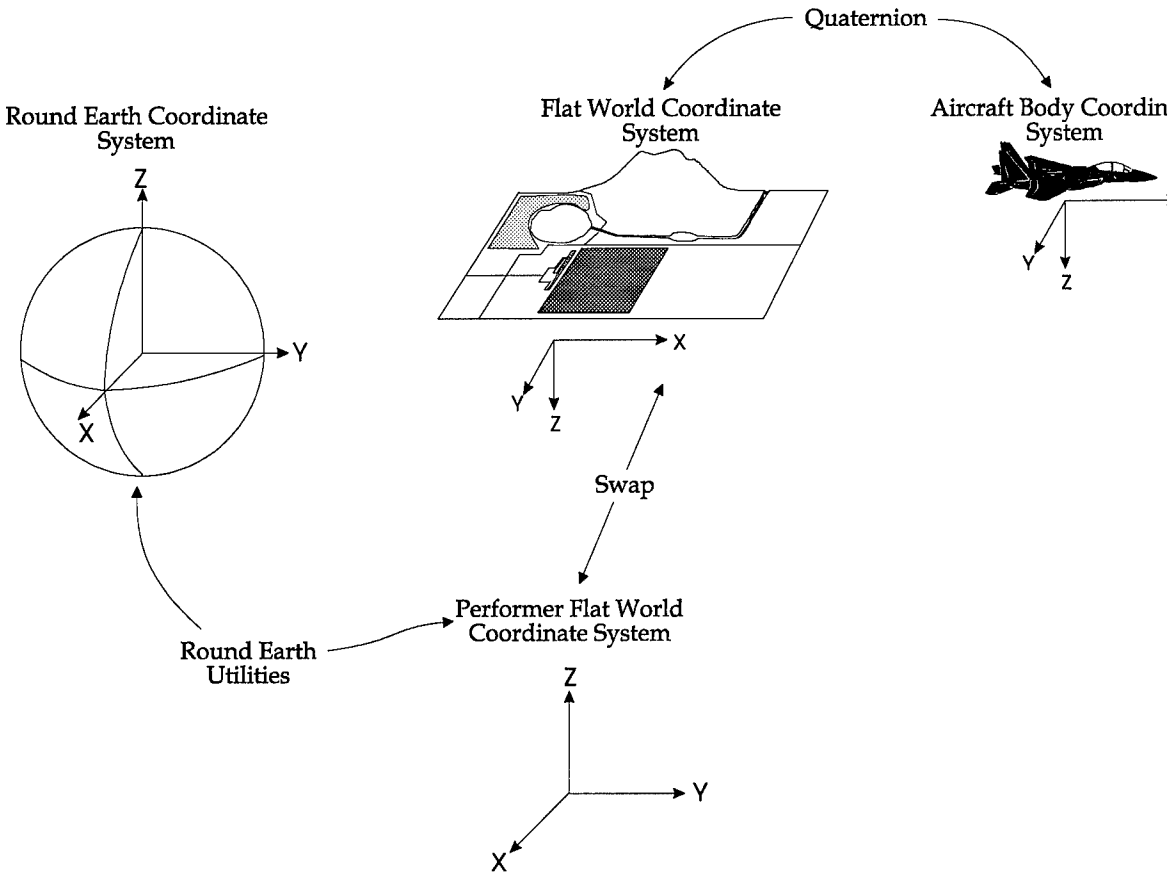

Figure 2: Relationship of Coordinate Systems [EDWA95]

coordinates must be transformed into a standard coordinate frame. The DIS's coordinate origin is at the centroid of the earth, with the Z axis passing through the north pole, the X axis passing through prime meridian at the equator, and the Y axis passing through 90 degrees east longitude at the equator. The

ObjectSim class, Round_Earth_Utils, provides the necessary routines to transform coordinates between the Performer and DIS coordinate systems [ERIC93].

2.4 Semi-Autonomous Forces.

A semi-autonomous force (SAF) operates in a mode somewhere between being completely computer controlled (autonomous) and human controlled. Semi-autonomous forces are necessary because the operation of fully autonomous forces has not yet been able to model the maneuvers of actual forces, and require additional human interaction. Rather than have each entity controlled by an individual, a compromise is made between automation and human control. The operator of the forces controls the goals or methods of the entities involved in the simulation, and the computer controls the detailed actions and maneuvers. These entities may be used to augment the friendly or enemy forces [CERA94].

The application discussed in Ceranowicz's paper, ModSAF, was applied to ground-based forces. An analogous research project for aircraft currently being conducted at AFIT is the Automated Wingman. Once developed, the Automated Wingman will be a semi-autonomous force system. Development of the Automated Wingman is challenging. As Ceranowicz noted, it is very difficult to get tanks to behave intelligently in a synthetic environment. High performance aircraft have even more degrees of freedom and must be responsive

in time as well. This added complexity requires the need for human intervention when the autonomous wingman does not behave properly thereby making the Automated Wingman a SAF.

2.5 Real-Time Issues of Entity Swapping.

Several issues involving time must be considered to swap entities in real time. Cheung and Loper addressed several of the difficulties introduced by the distributed nature of DIS, including latency between hosts and uncertainties in timing and delivery. However, these problems are not difficult to overcome on a single host or shared-memory multiprocessor machine, but they are more profound if the responsibility of controlling an agent moves from one host to another as in the VC environment [CHEU94].

Other important issues in the real time control of entities are PDU ordering and event sequencing. Any control hand-off between an autonomous agent and the controlling agent during the simulation must be carefully coordinated to overcome the difficulties noted by Cheung and Loper.

The controlling agent can be considered a single resource among the set of autonomous forces. This problem can be solved using a distributed mutual exclusion algorithm addressed by Singhal [SING94]. Since the collection of agents will be small, the use of a single controlling host to maintain requests of the dependent agents is sufficient.

2.6 Multiple Aircraft Simulation.

The Air Force Armstrong Laboratory reported on the implementation and training effectiveness of multiship air combat simulation [BELL93]. They provided empirical evidence that a distributed architecture can provide effective training. The architecture of the simulation tested is similar to AFIT's VC.

This simulation used a distributed collection of simulators interacting together through an Ethernet backbone. A team of two pilots flew dome-based simulators against four pilots operating enemy aircraft simulators. The underlying communications protocol was SIMNET, a predecessor to the current DIS protocol.

Overall, the results were positive. This experiment demonstrated air combat simulations could be conducted on a lower-cost distributed architecture. The pilots noted that the test was a realistic and positive training experience, in most cases.

The experiment also had some shortcomings. The resolution was insufficient to discern distant targets accurately. Also, any addition of new simulation devices into the scenario had to be carefully examined due to different levels of fidelity. These difference could have introduced unrealistic advantages, which "may degrade rather than enhance the quality of training" [BELL93].

2.7 The Virtual Cockpit.

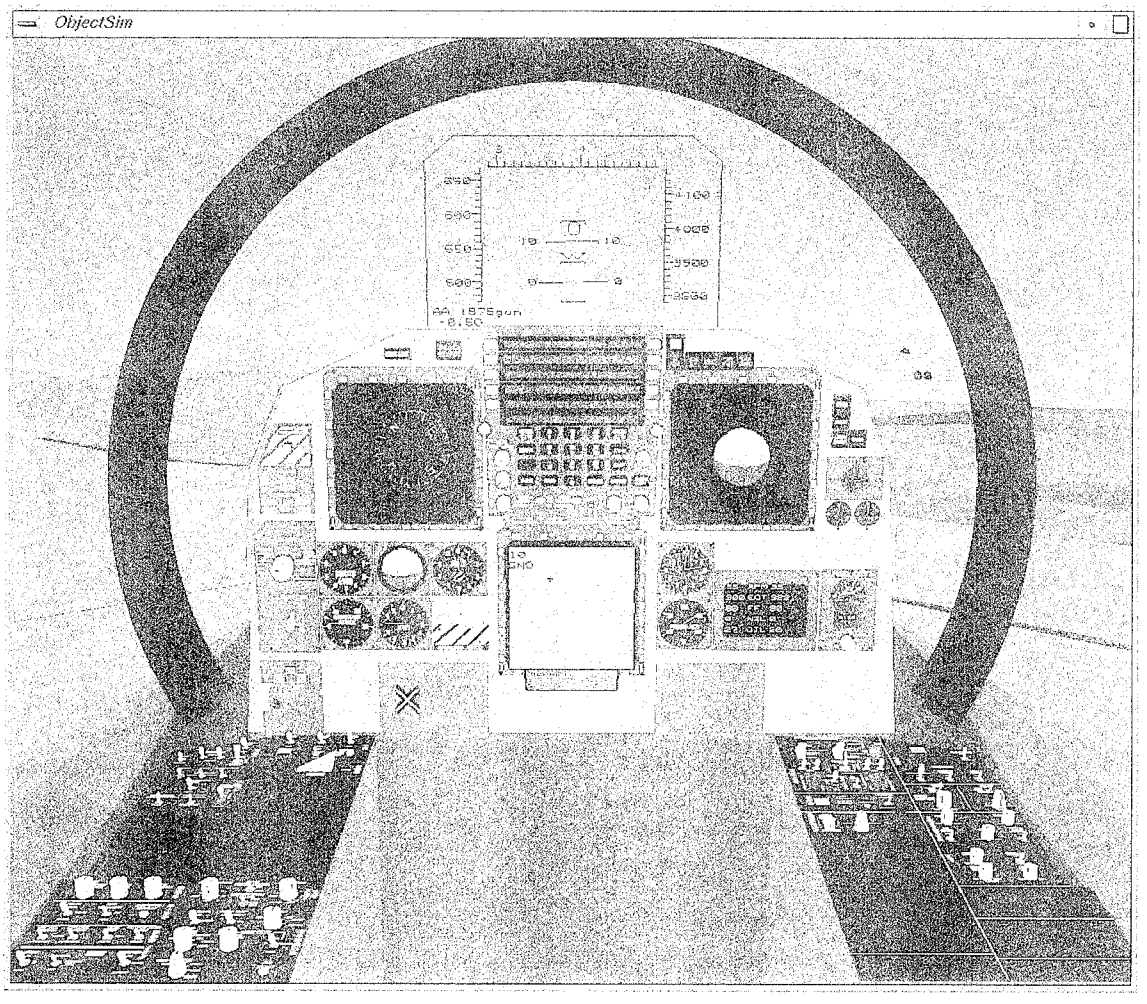

Figure 3: The Interior View of the Virtual Cockpit

The AFIT Virtual Cockpit (VC) is a DIS-compliant application that simulates the operation of an F-15E aircraft. The VC can give the operator either a screen view of the cockpit or use a head-mounted display to give a fully immersive presentation. The pilot controls the VC with the use of a hands-on throttle and stick (HOTAS), and the flight motion is based on a compute-optimized aerodynamic model obtained from the Naval Postgraduate School

[SWIT92][COOK92]. The interior of the cockpit is an accurate three-dimensional representation of an actual F-15E cockpit. The 3-dimensional control panel modeling is especially evident when the head-mounted display is used because the instrument panels with their instrumentation appear properly in perspective, and give the correct motion parallax. Figure 3 shows the interior of the Virtual Cockpit, and what an operator would see, either using the head-mounted display or the screen display. The figure also shows the indicator that is used for the Automated Wingman, which is the yellow status light with red lettering at the top of the console.

The VC application itself is very complex. Pieces of the code date back to 1991, and each year since, substantial modifications were made to the design of the system. As a result, much of the original design has been lost, or has been covered up with modifications. In most cases, all of the attributes and methods of classes were left public, and there are many references to other objects by pointers, accessing internal data members of other objects, even those that are were not designed as 'global' objects. The end result makes it quite difficult to maintain and modify. Figure 4 shows some of the design of the VC. It is by no means a complete view of the simulator, but shows how some of the communication occurs through the "Globals" object, and gives an overview of the object relationships. The diagram uses the notation presented by Rumbaugh [RUMB91], with diamonds representing components of a class, while the dotted diamonds and lines indicate references to an object by a pointer. The top level

object is My_Simulation, which is an subclass of ObjectSim's Simulation class. More detail about ObjectSim framework are given in [SNYD93].

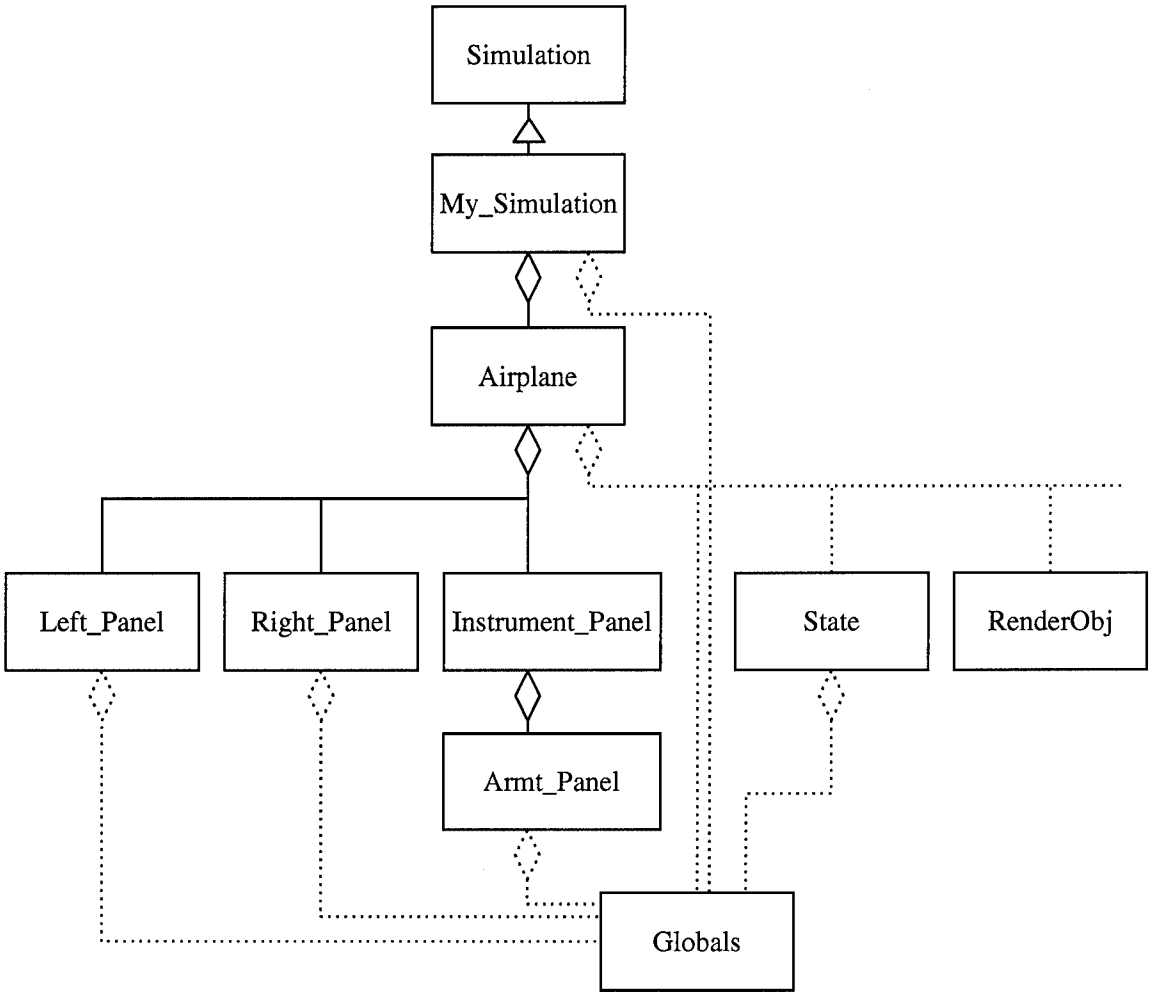

Figure 4: Some Components of the VC Application

2.8 The Automated Wingman.

The Automated Wingman (AW) application, beginning development this year, is a semi-autonomous force that will take queues based on a lead aircraft in

the simulation. The AW and VC are the first applications that will incorporate the switching capability. The AW is implemented using a fuzzy logic-based engine. The wingman actually has a "Fuzzy Pilot" that controls the aircraft, using throttle, stick, and rudder controls as would normal pilot. The goal of the AW is to be a force multiplier in a training scenario on a DIS network [EDWA95].

2.9 For More Information...

More detailed discussions of various pertinent papers and readings are summarized in the on-line annotated bibliography collection provided by AFIT's World Wide Web server. The address for the Virtual Environments, 3D Medical Imaging, and Computer Graphics Laboratory is

http://www.afit.af.mil/ENGgraphics/

The latest information about the current projects and the future directions can always be found there. The author's collection of annotated bibliographical sources may be found at

http://www.afit.af.mil/ENGgraphics/annobib/annobib.nws.html

2.10 Summary.

The use of computer controlled forces is a method for reducing the cost of interactive simulations. The biggest barrier to achieving fully autonomous simulated agents is the incredibly complex and instantaneous decisions a pilot must make that must be simulated in an autonomous force controller. One

would expect the operation of such a system would not be able to respond in all situations, so the capability to respond to this situation needs to be addressed. One manner is to allow a human operator to take control of the automated system for a brief time in order to take any action that might be necessary. Therefore, switching between a manned simulator and individual autonomous agents need to be added to the VC and AW, and in particular, to better support the required functions of the AW.

3 Requirements.

We anticipate that the Automated Wingman will not be able to respond to all potential situations. Given the Automated Wingman can determine when conditions require human intervention, it must be able to notify the lead aircraft and request assistance, or rather a swap of control. The request must be made known to the operator of the lead aircraft via a visual and audible queue. The current configuration has minimal audio capabilities and a visual cue can be used to further clarify the request to the operator. Furthermore, auditory cues further increase the operators awareness [DOLL86] Once the operator is notified, he must have the ability to ignore the incoming request for assistance. The operator's response must be detected in an intuitive and easy-to-operate fashion, and not require the operator to move his hands from the throttle or stick of the aircraft. Once the operator has confirmed the request, the exchange of entity states must be conducted in one second as per the DIS standard [DIS94]. Once the transfer has been carried out, the operator must have the ability to initiate a swap back to the original entity and to resume the lead of the formation, so that the operation of the scheduled mission would not be too disrupted.

The design of the Switcher should be Object-Oriented, to fit within the ObjectSim and Object Manager framework. Modification of the current Virtual

Cockpit application should be kept to a minimum, and the Switcher itself should be self-contained.

The Virtual Cockpit needs to provide a visual and audible cue to the operator to inform him of a request for assistance. The operator then will either respond with a command or button-press, or ignore the request. The request will time out after a defined time and either be re-issued if the assistance is still required or terminate. This operator time out is not specified in the DIS standard. Once the operator has confirmed the request, control of the wingman's entity will be transferred to the Virtual Cockpit. The information that is updated for the Virtual Cockpit will be at a minimum position, oriention, and velocities, with potential for growth in the future for more capable simulation applications.

The exchange of the entities must be transparent to other applications that are involved in a distributed simulation with the VC and AW. Applications which are not aware of the entity hand-off protocol (EHP) must not lose information due to entities being deleted and re-created on the network. Additionally, existing applications that have been implemented must be able to interpret the exchange and not lose information as well. This requirement implies that the standard will be used in some fashion. Changes made to the standard will be kept to a minimum in order to apply it to the swap.

3.1 Standards.

The applications developed will be compliant with the DIS version 2.0 (fourth draft). The Department of Defense has adopted this standard as the defining document to interface distributed simulations [DIS94].

3.3 Materials and Equipment.

The development of the context switching occurred within the development environment that has evolved in the past five years in AFIT Virtual Environments Laboratory. Several tools and libraries have been developed here as the subject of past research. The most pertinent to this research are ObjectSim and Object Manager. In addition to these libraries, the available hardware and system libraries of the graphics laboratory will be discussed.

3.3.1 ObjectSim.

ObjectSim is an object-oriented application framework that eases the development of interactive graphical simulations. It encapsulates the complexity of the three-dimensional rendering that is specific to the hardware of the target system. It provides an organizational structure for implementing the various functions necessary for an interactive simulation. Each function is each contained in a separate class that is overridden based on the requirements of the particular application.

3.3.2 *Object Manager.*

Simulation applications that have been developed at AFIT interface with the DIS protocols using the Object Manager. The Object Manager is a library of C++ classes encapsulating the details of the network protocols. Object Manager also manages a few other entity related issues including entity lists, position data, and dead-reckoning [SHEA95].

3.3.3 *IRIS Performer.*

Performer is a set of graphic rendering libraries specific to the Silicon Graphics machine architecture. For much of the implementation of the Switcher, ObjectSim will make the necessary calls to Performer to render the visual scenes. Some portions, especially modifying the visual aspects of the interior of the cockpit, will require using Performer directly.

4 Design

4.1 The "Big Picture."

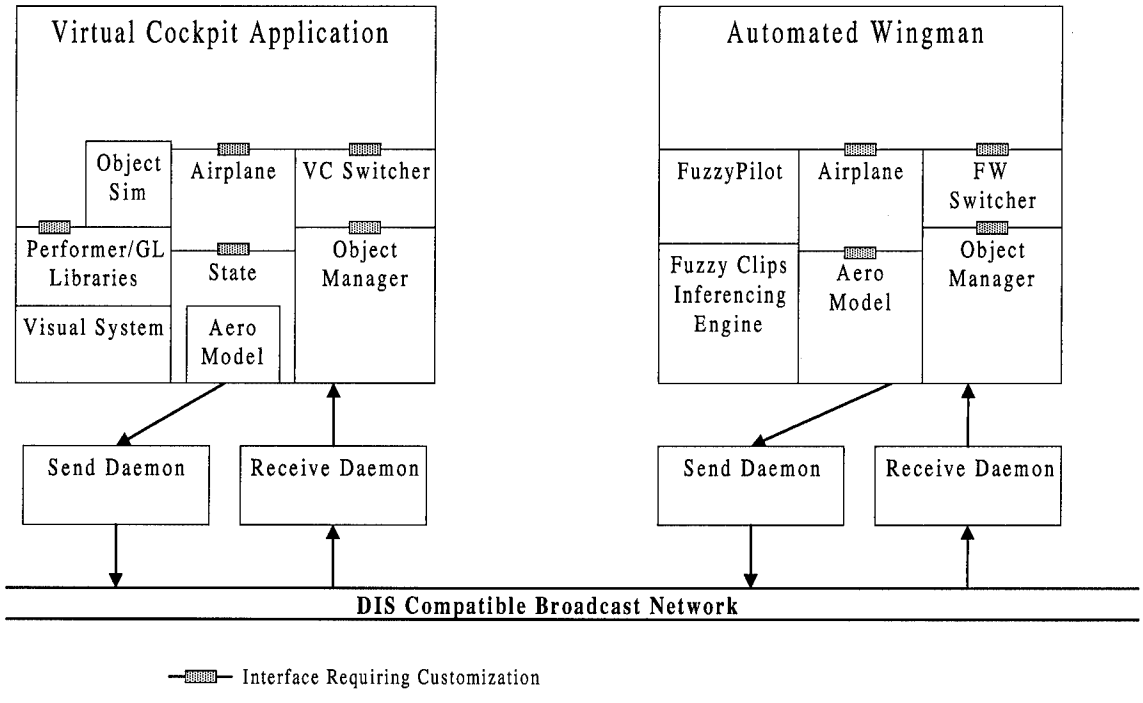

Figure 5: Architectural View of the VC, AW and Switchers

As alluded to earlier, the VC and AW communicate through a DIS compatible network. Figure 5 is a simplified high level architectural diagram showing the relationship between the VC and AW applications and the Switchers. Objects touching one another vertically within the diagram communicate directly. From this diagram, one can tell that the Switcher objects themselves only communicate with the Object Manager and the main application, and are inserted between the applications and the Object Manager. In the case of the VC, the application must respond to user requests to initiate the

swap, and then make modifications to the display, which are handled through Performer and ObjectSim. In the VC and AW, the applications still communicate to the network through the Object Manager for other actions.

4.2 Analysis of DIS recommendations.

The EHP as defined within the draft standard does not fully meet the needs of the entity switcher for the Virtual Cockpit and Automated Wingman. The reasons noted are as follows:

- EHP is unidirectional.
- EHP may confuse other applications.

Since EHP is unidirectional, the protocol either must be applied twice in order to get the appearance of a swap, or may be further modified to minimize the latency of network transmission. I chose to modify the ordering of the PDUs to decrease the latency of the transfer. The reordering interleaves the two exchanges, allowing some of the information to be transferred simultaneously between the Virtual Cockpit and Automated Wingman. The sequence of request-transfer-acknowledge is still maintained, but now some of the transmissions overlap. To an observer on the network, this looks like two entity handoffs, as it should, but introduces more complexity that must be accounted for in the internal states of the involved players. The internal states of the objects are discussed in more detail later.

Since EHP is still defined only in a draft standard, it will not be implemented in most active DIS applications, leading to inconsistent information contained within stealth viewers and other agents on the network. The Transfer Acknowledgment PDU specifies the old and new entity identification triple, specifying the entity's site, application, and entity identifier. With this information, the original application deletes the old entity from its list, and the receiving application adds the entity to its list. A problem arises when another application is interacting on the network. For a brief time, that application will witness two entities, possibly with diverging paths. DIS dictates that an entity should be removed from the simulation if it does not have a entity state PDU after twelve seconds. Therefore, this duplication will be temporary in nature, but may still cause inconsistent information in an application's entity lists. Any additional information or statistics that applications are keeping on entities will also be disrupted.

To correct this duplication problem, I preserved the entity's identification. The entity's site, application and ID are transferred across the network in the Transfer Control PDU, as specified by the draft standard. But now, rather than create a new triple based on the receiving application's ID, the application maintains the original information of that entity. Thus, other applications on the network do not see any change of the entity, which makes for a smooth transition, and the transfer of control is transparent to other networked applications. This approach has been discussed with members involved with the

development of the standard and was accepted as an viable alternative to the current specification [NEFF95].

4.3 Message Communication Patterns.

As stated in the background section, the entity hand-off protocol as given by the draft standard was insufficient to apply to the entity swapping for the Virtual Cockpit and Automated Wingman. In order to minimize changes to the standard, the same PDUs that are described in the standard are used. To an

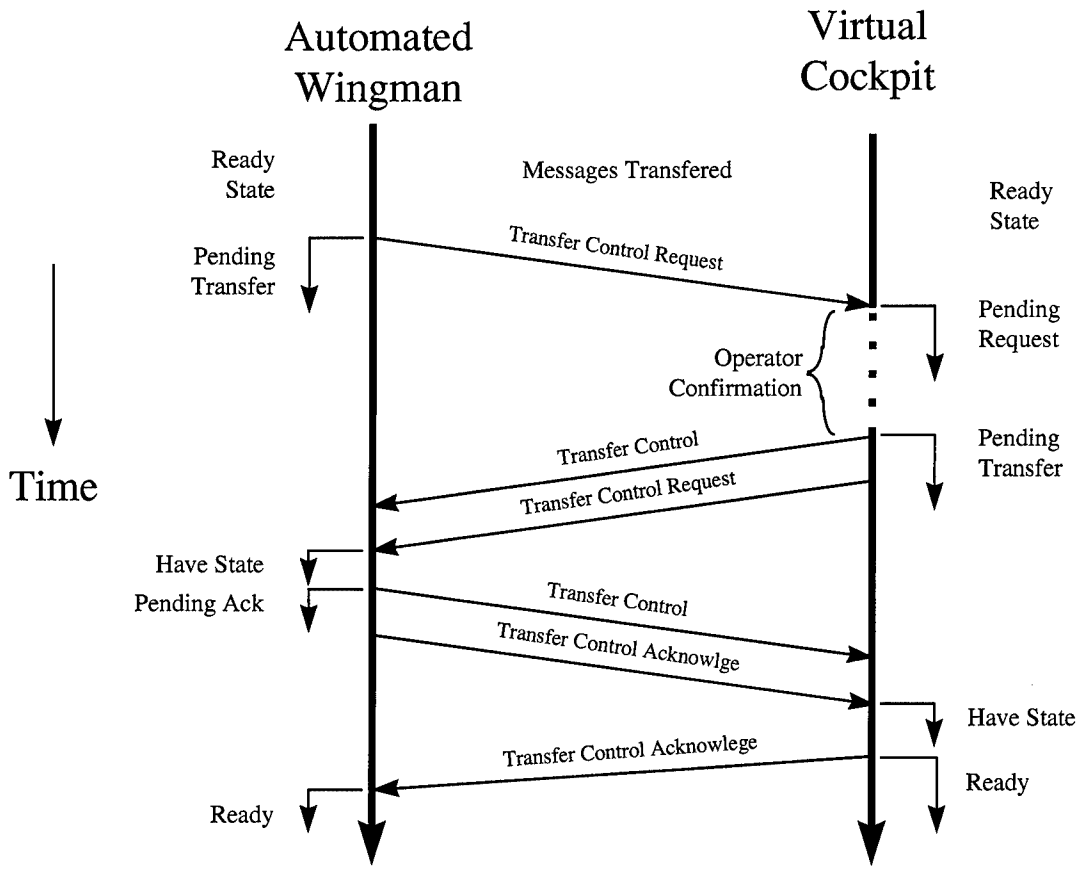

Figure 6: Common Flow of Messages (AW Initiated)

application that is aware of the standard, this exchange appears as two normal entity hand-offs. To an application that is not aware of the EHP, these messages are discarded. The entity identifiers are kept the same during the transfer, so any other application that is on the network will not be aware of the transfer of control.

Figure 6 shows the flow of communication between the AW and the VC when the AW initiates the exchange. The exchange shown represents the case when the AW requests assistance from the operator. One can see that there are six messages, representing two transfers of control that might be picked up by other applications. Figure 7 shows the common case of control transfer when the transfer is initiated at the Virtual Cockpit. The diagram depicts the operator unilaterally taking control of the wingman, which would happen when the operator has completed performing the actions of the wingman. The cases are similar as would be expected, but the state names are changed to keep the transactions separated. Additionally, there is no operator confirmation for the Automated Wingman, since it has no operator.

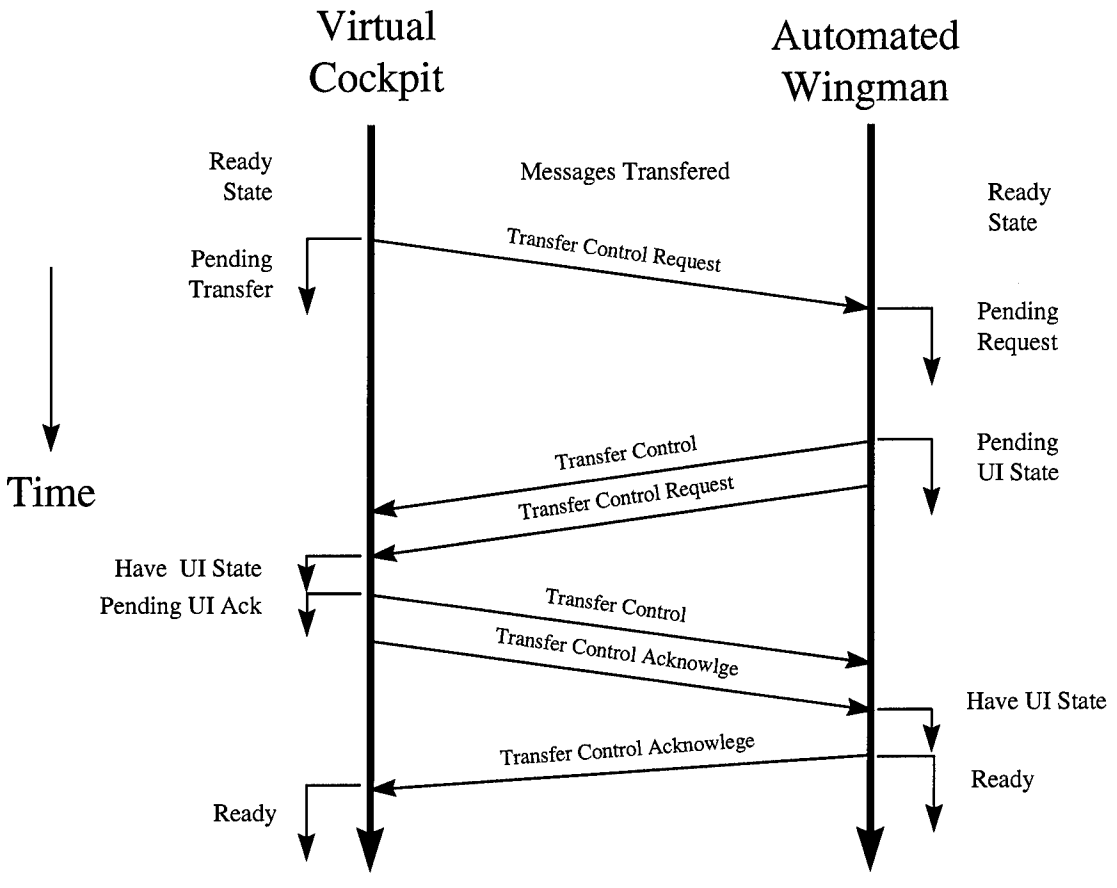

Figure 7: Common Flow of Messages (VC Initiated).

4.4 The Switcher Objects.

The design of the switcher objects was developed using the Rumbaugh [RUMB91] object-oriented software engineering methods. The first step in the development of the design was to identify all of the potential objects in the development. In this case, I selected a single object, the Switcher, to be the top level object. The more specific implementations for the Automated Wingman and Virtual Cockpit are subclasses of the Switcher class. The top object, the Switcher, contains skeletal attributes and methods. The switcher class

constructor calls Init() and does nothing else. This design enables a class object to be re-initialized by calling the Init() method directly. The overall object-oriented diagram is presented in Figure 8.

4.4.1 Switcher_Entity Class.

The switcher_entity class mirrors the information in the header of a DIS PDU. The switcher_entity class contains the information of the entity's site, host

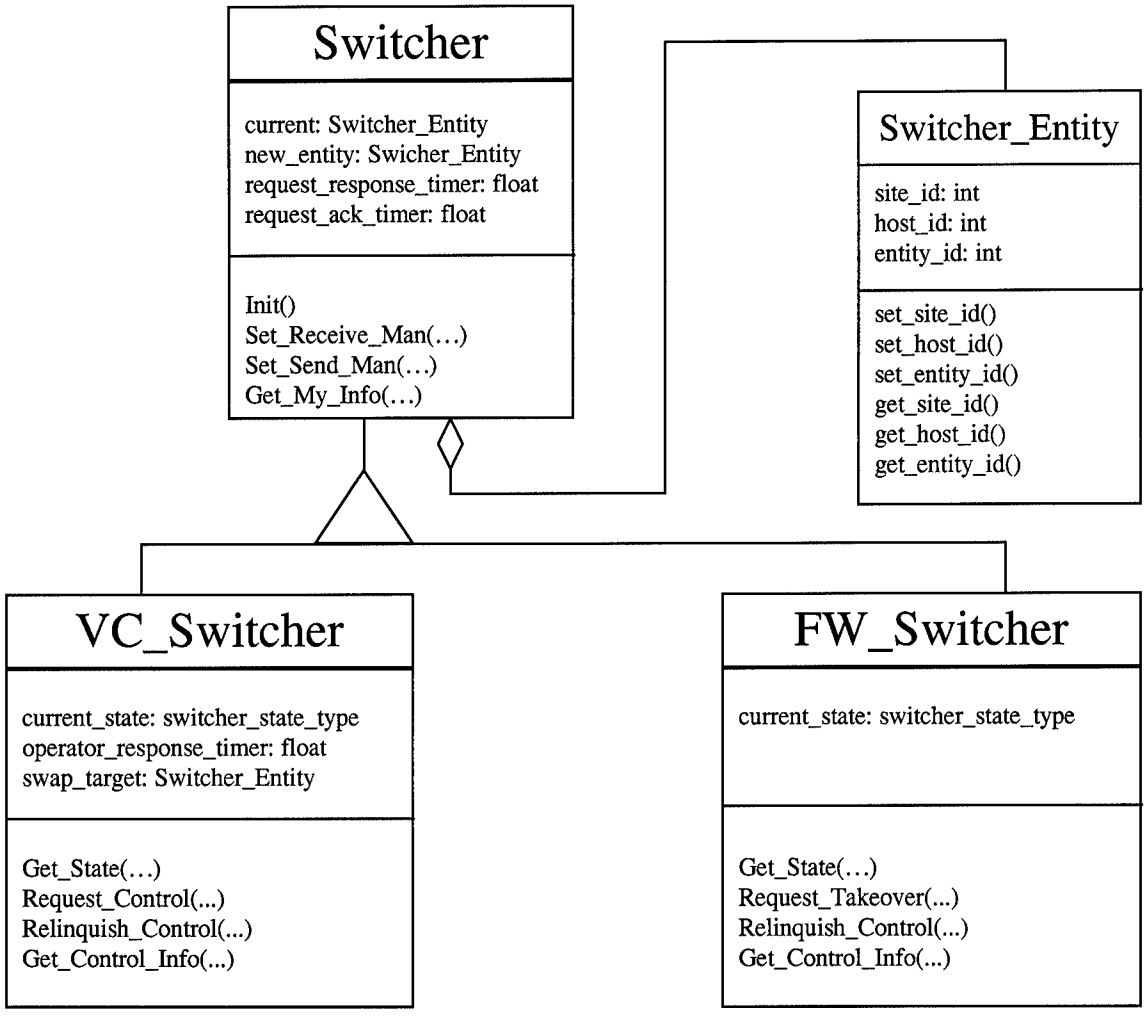

Figure 8: Switcher Object Hierarchy.

and entity identifiers. This is the identifier triplet used by DIS applications to distinguish the different actors on the network. Naturally, the get and put operations act on the associated attributes.

4.4.2 Switcher Class

The attributes current and new_entity are class instances of the switcher_entity class. They are used to keep track of the important players in the entity swap. Current points to the information of the current entity controlled by the application. New_entity points to the information that has been received from the network and will become the current when the transfer is complete. The request_response_timer and request_ack_timer are used store the time at which a request and state are set respectively, and are used to determine when the time-out periods expire.

In addition to these attributes, the Switcher base class also defines public methods that set up the Switcher for communication with the Object Manager. The Set_Send_Man() and Set_Receive_Man() accept pointers to the send and receive portions of the Object Manager respectively because the Object Manager has separate objects for each function. These methods allow the Switcher sub-classes to access the Object Manager directly to update and maintain the entity's proper DIS information. The Get_My_Info() method returns the DIS identification values for the current entity.

4.4.3 VC_Switcher Class.

The VC_Switcher class is specific to the operation of the Virtual Cockpit. The VC_Switcher's operation is modeled as a Moore state machine. The current_state attribute defines the current state of the switcher and guides the actions of the Switcher and the VC application. Figure 9 gives the definition of the switcher_state_type from the header file.

```
enum switcher_state_type {ready,
                          pending_request,
                          pending_transfer,
                          have_state,
                          pending_ack,
                          pending_ui_transfer, // ui = user initiated
                          have_ui_state,
                          pending_ui_ack};
```

Figure 9: Definition of switcher_state_type

This variable provides synchronization between the VC application and the Switcher. The meaning and use of each state is defined in more detail within the implementation section along with the presentation of the state transition diagrams.

The state transition diagrams for the VC_Switcher are given in Figure 10 and Figure 11. The Switcher has two independent paths of control starting at the Ready state, so the overall state diagram can be partitioned into two independent graphs that share only the Ready state. The loop in Figure 10 is initiated by the receipt of a transfer request that is initiated by the AW. The loop in Figure 11 is initiated by the operator.

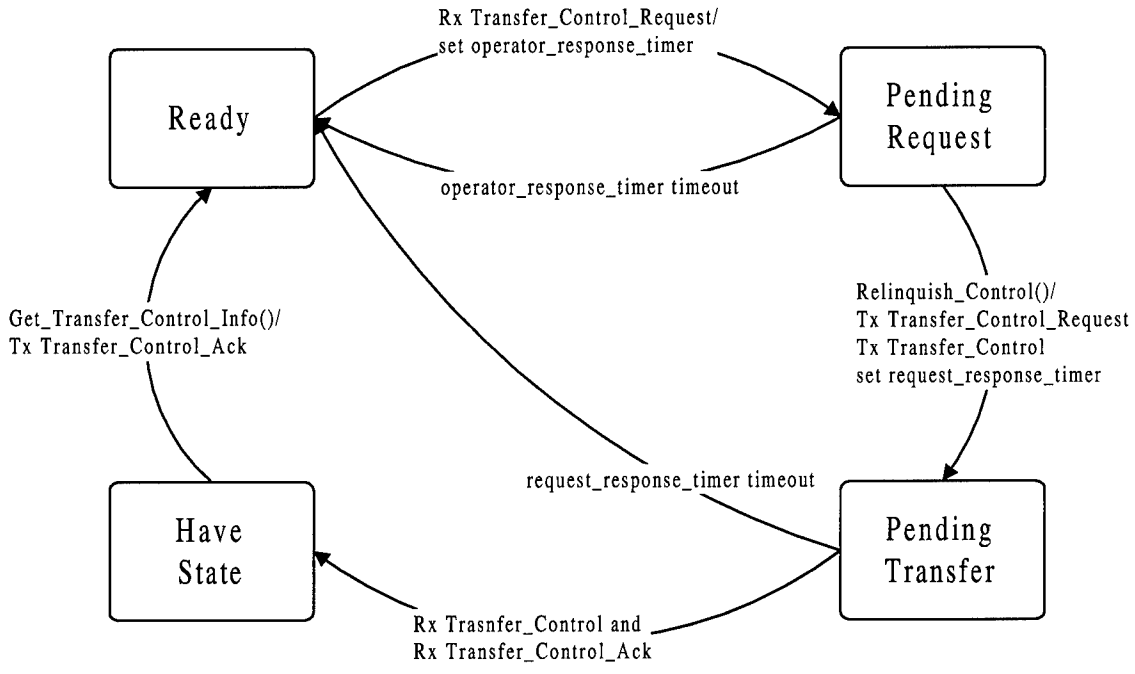

Figure 10: VC_Switcher State Transition Diagram (AW Initiated)

4.4.3.1 AW Initiated

Ready. The Ready state is the common case and the beginning of this loop. The VC_Switcher will be waiting in this state until a request is received or the operator initiates the transfer. When the Transfer Control Request PDU is received, it transitions to the Pending_Request state. Before transitioning to the Pending_Request state, the operator_response_timer is set.

Pending Request. The VC has just received the request and is now waiting for the operator to confirm. If the operator does not respond in the amount of time specified by OPERATOR_TIMER_TIMEOUT symbol, control flows back to the Ready state. If the operator confirms, the VC calls the

Relinquish_Control() method. This method sets the current_state to pending_transfer and issues two PDUs: Transfer Control Request and Transfer Control. The request initiates the transfer of control from the wingman. The Transfer Control sends the current position, angular and velocity along with internal state information that the AW requires.

Pending Transfer. The VC is waiting on the state information of the wingman and acknowledgment for the state it sent. When it receives those two PDUs, it proceeds to the Have_State state. If the request_response_timer exceeds the value in the REQUEST_RESPONSE_TIMER symbol, the transfer aborts and control passes back to the Ready state.

Have State. This state notifies that the application that the information from the AW is ready to be received. The application responds by calling the Get_Transfer Control_Info() method. This method returns the values provided by the AW and transmits the acknowledgment back to the AW. There is no timer associated with this state because the application itself makes the call and there is no involvement outside of the application for this transition.

4.4.3.2 VC Initiated.

The other half of the state diagram for the VC_Switcher is initiated by the operator initiating the request. The operator's request is given in the same manner as he acknowledges a request, by using a spare throttle switch button.

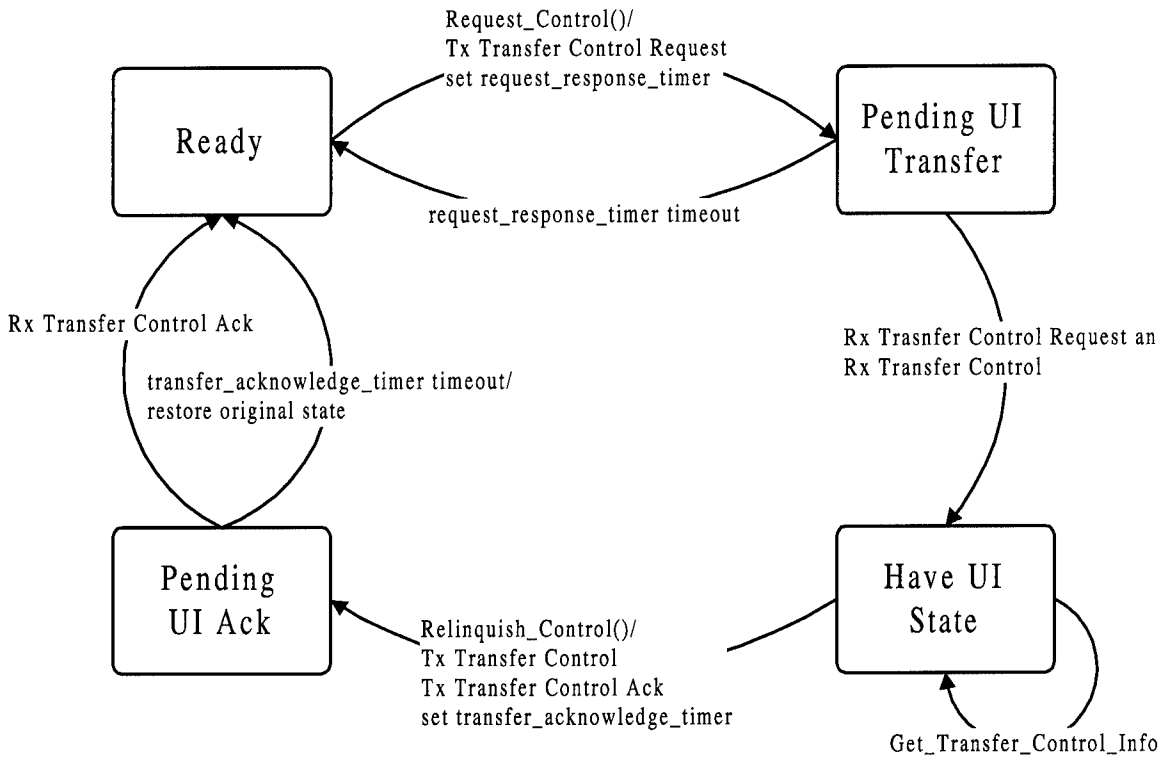

Figure 11: VC_Switcher State Transition Diagram (VC Initiated)

Ready. This is the normal, waiting state. When the operator presses the AW activate button, the application calls the Request_Control() and transitions to the Pending_UI_Transfer state. Pressing the AW activate button means the operator wants to take control of the attached wingman. Request_Control() sets the current state to Pending_UI_Transfer and sets the request_response_timer before it exits.

Pending UI Transfer. The operator has requested control of the wingman, and the VC_Switcher has sent the request for control. Now the VC_Switcher is waiting for the AW's information and request for the VC's information. If while

waiting, the request_response_timer becomes older than the REQUEST_RESPONSE_TIMER, the wait expires and the state is changed back to the Ready state. This will be a rare case since the AW's response to the Transfer Control Request PDU is automatic because there is no operator. When the VC_Switcher receives the two PDUs, it sets the state to Have_UI_State.

Have UI State. This state notifies the VC that a new state has been received and that it needs to call the Get_Transfer_Control_Info() method to proceed. There is no timer on this state since the methods will be called in the next execution loop. The Relinquish_Control() method passes the state information of the VC and the acknowledgment to the AW. The call finally sets the timer for the acknowledgment and sets current_state to Pending_UI_Ack.

Pending UI Ack. This state waits for the final acknowledgment from the AW. When the acknowledgment is received the state is set to Ready. If the timer expires, the old state is reloaded.

4.4.4 FW_Switcher Class.

The FW_Switcher class operates very closely to VC_Switcher class. Indeed, the original design did not differentiate between them, but the VC's requirement to give the operator the ability to confirm the swap made the operation different enough to need different objects.

The state transition diagram of the FW_Switcher is parallel to the VC_Switcher. The Ready state is the starting state, and two events cause it to

leave this state, each one starting it into separate loops of control. The first loop, shown in Figure 12, is initiated by the AW. This control loop interacts with the VC_Switcher's state diagram shown in Figure 10. The second loop, shown in Figure 13, is initiated by a request from the VC_Switcher shown in Figure 11.

4.4.4.1 FW Initiated.

Ready. The FW_Switcher waits in this state until the FW calls the

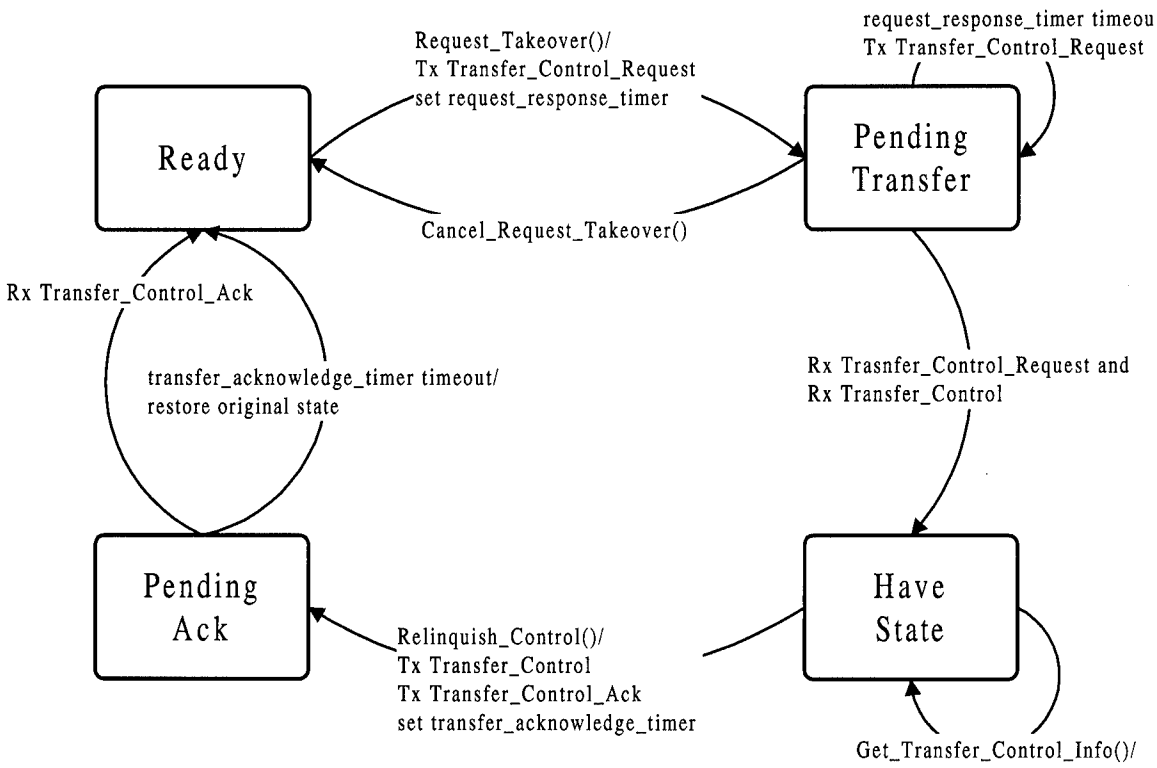

Figure 12: FW_Switcher State Transition Diagram (AW Initiated)

Switcher's Request_Takeover() method, or a Request_Control PDU is received from the VC. In this case, a Request_Control() call is made. The

Request_Control() method call is essentially the "Help Me!" message. In that method, the FW_Switcher transmits the Transfer Control Request PDU to the VC, sets the request_response_timer and advances to the Pending_Request state.

Pending Request. The Pending_Request state waits for a reply from the VC to relinquish control. It is possible for the operator of the VC to be occupied and unavailable. If that is the case, the transfer_request_timer will expire and another request is sent. This will repeat until the VC operator acknowledges or the AW no longer requires help. The AW cancels the request for assistance with the Cancel_Request_Takeover() method, which returns the Switcher to the ready State. If the VC responds after the transition back to Ready, the request is ignored. If both Transfer Control Request and Transfer Control PDUs are received, control passes to the Have_State state.

Have State. The Have_State state signals the AW application information is ready to be received, and the VC requires information in return. To collect the new entity information, the FW calls Get_Transfer_Control_Info(). After this method, the FW calls Relinquish_Control to send its information and acknowledge the receipt of the VC's information. Finally, the call advances the state to Pending_Ack and starts the transfer_ack_timer.

Pending Ack. The Pending_Ack waits for the final acknowledgment from the VC. Once the acknowledgment is received, the state advances to the Ready

state. If the acknowledgment is not received, then the state is restored from the original.

4.4.4.2 VC Initiated.

The final state diagram shown in Figure 13 gives the response of the FW_Switcher to a Transfer Control Request issued from the VC. It is tightly coupled with the states from the VC in Figure 11, where the operator initiates a swap.

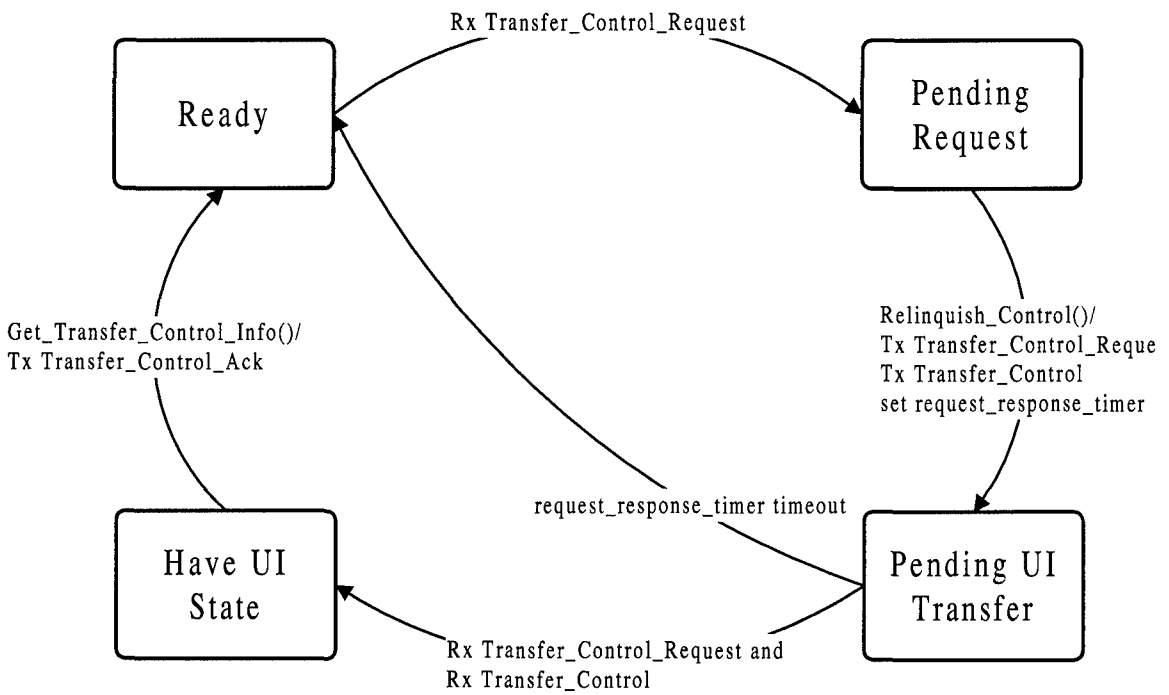

Figure 13: FW_Switcher State Transition Diagram (VC Initiated)

Ready. As with all of the other state transition diagrams, the starting state is Ready. The exit condition in this case is the receipt of a Transfer Control Request, and the state is advanced to Pending_Request.

Pending Request. This state notifies the AW that the operator of the VC requested its control. The AW responds by calling the Relinquish_Control() method. The Switcher then transmits the Transfer Control PDU followed by the Transfer Control Request PDU to get the VC's information. The method then sets the request_response_timer and advances to the Pending_UI_Transfer state.

Pending UI Transfer. This state signifies that the switcher is waiting for information from the VC. The waiting is complete when it receives both the Transfer Control and Transfer Control Request PDUs, and the state is advanced to Have_UI_State. If the request_response_timer expires before both PDUs are received, the state is returned to Ready, and the exchange is aborted.

Have UI State. This state signals the AW that the information from the VC has been received. The AW responds by calling Get_Transfer Control_Info(). This call returns the information that was transferred over the network, sends the acknowledgment and completes the transaction by returning to the Ready state.

5 Implementation.

5.1 Communication Between Applications and Switchers.

Several options exist to implement active objects such as the Switcher class within an application in a multitasking environment. The first implementation considered was to create a new thread of control in a separate process. This method, although well suited to the design as presented in chapter 4, requires substantial overhead, especially for the Virtual Cockpit. A Silicon Graphics four-processor Reality Engine^2 Onyx workstation is currently use to host the VC. The Performer graphics imaging system uses three of the processors [COOK92][SGI94]. The Object Manager, network daemons, and the rest of the VC application's aerodynamic and remaining functions execute on the final processor. Running the Switcher as a separate task would have required additional operating system level processes switches, and more interprocess communication channels.

Another possible method would be to implement callbacks. To implement callbacks, the calling procedure would provide a set of function pointers to the Switcher. When the Switcher would need to perform functions as "notify the user of an incoming request," the Switcher would call the function referenced by the pointer. This limits the interaction with the Switcher to instantiation time,

where the objects would be given the callback functions, but the flow of control is not evident, and there is little savings in execution time.

The final possible method, the one selected, was to use a state variable to allow the application to take the correct selection action on the Switcher object based on its current state. This method made the implementation straightforward and the interaction between the applications and the Switchers easier to understand.

Both the VC_Switcher and FW_Switcher objects have a Get_State() method which checks which of the different transfer control PDUs have been received and its current state, determines the new state. The Get_State() method returns a value of switcher_state_type as defined in chapter 4. The application uses this value to determine its next course of action.

5.2 Modifications to the Object Manager.

Because the Object Manager had not been capable of sending and receiving transfer control PDUs prior to this research effort, methods were created to handle them. For each PDU type, Transfer Control Request, Transfer Control, and Transfer Control Acknowledge, one shared flag and two methods were created. Since the Object Manager is split into two separate objects, one send and one receive, a broadcast and receive method for each PDU type are added to the appropriate Object Manager class. The broadcast methods transmit the respective PDUs on the network. When a PDU is received for the

application, it sets the shared flag to true. The switcher clears the flags by using the act_on_Transmist_Control_Request_PDU() or the other corresponding methods.

In addition to manipulating PDUs, the Object Manager must also report and update its entity identifier so the Switchers can change the applications reported entity. The entity's ID is set and retrieved by the set_entity_id_rec() and get_entity_id_rec() respectively. Since the Object Manager is separated into two parts, the set_entity_id_rec() must be called on both to ensure that both are changed.

5.3 Modifications to the Aerodynamics Model

The Automated Wingman was based on the same aerodynamics code that was in the VC at the beginning of the year. For convenience and efficiency reasons, the VC's version of the aerodynamics model made extensive use of the matrix representation structures and matrix manipulation routines available in Performer. In order to make the AW more portable, all of these references to Performer were removed and a generic, template-based set of classes were used in the aerodynamics model instead. Once the AW's version of the aerodynamic model had stabilized, it was re-integrated into the VC. This was done to keep the underlying aerodynamics models the same between the VC and AW.

5.4 Modifications to the Virtual Cockpit.

In addition to the VC's aerodynamic model being closely tied to Performer, it was contained within the State class, which contained much more information than the internal variables of the aerodynamics code. All of the different coordinate systems were also stored in state class, along with the public aerodynamics variables. The code that performed the aerodynamic calculations was imbedded in the State class and modified different variables in the State Object. These problems were removed when the redesigned aerodynamics model was re-integrated into the VC. The State object was not removed because of how many times it was referenced by other parts of the VC, but its understandability was enhanced by the change. Now the aerodynamics module stands alone, and has no ties to the State object and is more reusable as well

In a ObjectSim application, prior to each frame display, the main procedure calls each object's Propigate() method. The calls to the Switcher are performed in the Airplane class's Propigate() method. Therefore, the switcher object is checked at each frame.

The VC's visual display was modified to provide the required visual cue. The topmost spare warning button on the panel was used for an indicator for the AW. The object that contains this functionality was the Instruments class. To make the indicator, a texture map with red lettering and a bright yellow background was applied to the button and the action of the button was tied to a

boolean variable that was added to the Globals class in the same manner as the remaining buttons and controls for the display. The boolean variable is then set by the section of code in the Propagate() method that performs all of the Switcher interfacing.

5.5 Modifications to the Automated Wingman.

The modifications to the AW were confined to the top level object of the AW application. In a similar fashion to the VC, each computational loop the AW checks the status of the Switcher and take the appropriate actions.

6 Results and Recommendations.

6.1 Testing

Testing was conducted with several tools developed in the Graphics Lab in addition to the debugging output. The first tool is a debugging tool developed in the Graphics Lab by Steven Sheasby, the PDU_reader. The PDU_reader monitors the network for DIS PDUs and displays their contents into a text stream. It can be configured to filter out arbitrary PDU types . The tool proved to be quite useful in debugging, and later, demonstrating the correct transmission of the PDUs sent by the switchers.

Although the debugging output provided a precise measure of the accuracy of the information that had been transferred across the network, the Synthetic Battle Bridge (SBB) gave a quick visual guide of how the objects were flying and their position relative to the terrain. The SBB is a stealth viewer of the network activity which uses the same terrain database and displays a 3-dimensional view of the battle space. The SBB has the capability to tether to an entity in the simulation and follow its position in the battle arena. The tethering mode was used to watch the transfer between the VC and AW. The transfer occurred transparently, but the error in the alignment was evident and the aircraft's rocking was noticeable.

6.2 Switching Delay.

A major concern is the delay associated with the transfer of control between entities. Another concern is how different numbers of entities affect this delay. Figure 14 illustrates the measured time used to transfer the state across the network and move to the new position as a function of network entities. Since the state of the switchers is checked once each frame cycle, it turns out that the delay associated with the switch is closely related to the frame rate of the application. The measurements for the transfer time were measured from the time that the operator depresses the switch consent button to the time that the information is loaded on the VC. Even in the case of 1000 entities, the typical delay was less than 0.2 seconds, and was humanly difficult to perceive.

At least ten switches were performed at each network load level. The minimum and maximum of each set were also reported. The frame period was calculated by averaging the times for frames 100 to 200 of the simulation. The time measured is the time from when the operator acknowledges with the throttle button switch to when the new state is loaded. The frame period provides a baseline and a minimum time for the switch, and almost exactly overlays the minimum delay in Figure 14.

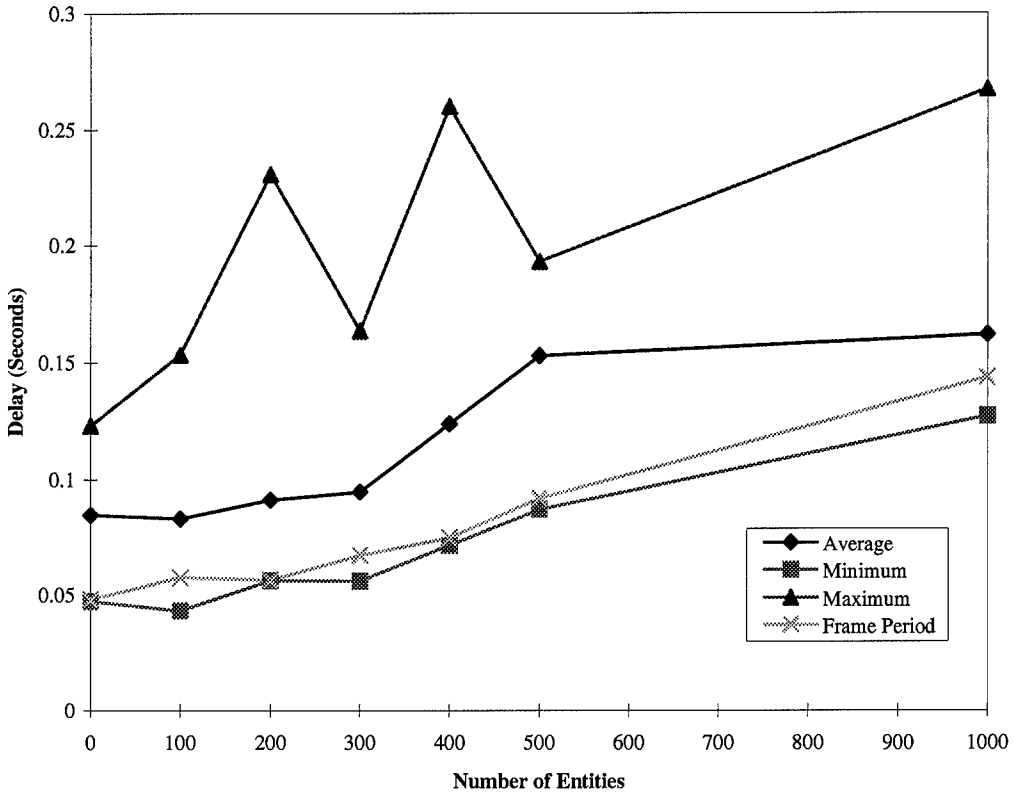

Figure 14: Transfer Time Vs Number of Network Entities

6.3 Operator Effectiveness In Switch.

Considerable confusion may result when the operator switches into a troubled aircraft. The pilot may not necessarily know the attitude of the aircraft or have any tactical knowledge of the situation that he is assuming. Thus, significant disorientation would be expected initially after a switch. The impact on the reality of the simulation for other players in the simulation is not known, since the actions of an operator switching into the cockpit of the wingman are unpredictable.

A potentially useful method to give the operator better situational awareness when switching into the AW's craft would be to allow the AW to continue to fly the aircraft while the operator get an understanding of the tactical situation. When the operator would feel comfortable in the new situation, he would make a final commitment to the transfer and would assume control of the new aircraft. This would require integrating the entire AW functionality into the VC, and would be a substantial undertaking.

6.4 Re-engineer the current VC code.

As alluded to briefly in chapter two, the code of the VC is difficult to modify. Although the code is divided into objects, most of the members of each class are public. References to members of other objects greatly complicate making modifications to the code due to dependencies that are created by those references. For instance, in the short code fragment from the VC shown in Figure 15, one can see how tightly the different objects are coupled. PlayerPtr is a public member of the VC_Net_Manager class and the first two statements traverse through three classes to get to the desired function. An occasional reference like this in the code might not be all that detrimental, but this kind of reference is quite commonplace in the code. The third statement demonstrates how the data types of the VC are dependent on Performer. In that statement, a Performer macro is used to set a data member two class references away. References to Performer are not detrimental in themselves; indeed, they are necessary for the

application, but also make the application extremely difficult to port to other systems, and add to the complexity of the code.

```
PlayerPtr->terrain->REU->rnd_pos_to_flat(tempx, tempy, tempz);
PlayerPtr->terrain->REU->rnd_euler_to_flat(temph, tempp, tempr);

PFSET_VEC3(PlayerPtr->Coords->xyz,float(tempx),
                                  float(tempy),
                                  float(tempz));
```

Figure 15: VC Code Example

6.5 Provide The Ability to Support Different Aircraft.

The current implementation of the switcher is restricted to the exchange of states between the VC and AW. Future enhancements should include the ability to switch between different types of aircraft, including the ability to represent the aerodynamic model of the different aircraft as well as a geometric representation of the interior of the different aircraft. These enhancements would allow the VC to be used in a broader array of military applications.

Bibliography

[BELL93] Bell, H. H., Peter M. Crane. "Training Utility of Multiship Air Combat Simulation," *Proceedings of the 1993 Winter Simulation Conference,* IEEE.

[CERA94] Ceranowicz, A. "Modular Semi-Automated Forces," *Proceedings of the 1994 Winter Simulation Conference,* IEEE.

[CHEU94] Cheung, S., M. Loper. "Synchronizing Simulations in Distributed Interactive Simulation," *Proceedings of the 1994 Winter Simulation Conference,* IEEE.

[CHRI95] Christian, T. Chief of Simulation Systems, Wright Labs. Meeting 27 April 1995.

[COOK92] Cooke, J. T., *Pameterized Plight Dynamics Simulation System Using Quaternions,* MS thesis, Naval Post-Graduate School (NPS), Monterey, CA, March 1992.

[DIAZ94] Diaz, M. E. *The Photo-Realistic AFIT Virtual Cockpit.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH AFIT/GCS/ENG/94D-02, December 1994.

[DIS94] US Government. "Standard for Distributed Interactive Simulation -- Application Protocols," Version 2.0 (Fourth Draft), 4 February 1994.

[DIS95] US Government. "Standard for Distributed Interactive Simulation -- Application Protocols," Version 2.11 (Working Draft), 4 February 1994.

[DOLL86] Doll, T. J., Folds, D. J., "Auditory signals in military aircraft: ergonomics principles versus practice," *Applied Ergonomics,* December, 1986

[EDWA95] Edwards, M. E., *The Automated Wingman: An Airborne Companion for Users of DIS-Compatible Flight Simulators.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH AFIT/GCS/ENG/95D-01, December 1995.

[ERIC93] Erichsen, M. N. *Weapon Systems Sensor Integration For a DIS-Compatible Virtual Cockpit.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH AFIT/GCS/ENG/93D-07, December 1993.

[GARD93] Gardner, M. A. *A Distributed Interactive Simluation Based Remote Debreifing Tool for Red Flag Missions,* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH AFIT/GCS/ENG/93D-09, December 1993.

[GERH93] Gerhard, W. E. *Weapon System Integration For The AFIT Virtual Cockpit,* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH AFIT/GCS/ENG/93D-10, December 1993.

[HOUCK91] Houck, M. R., G. S. Thomas, H. H. Bell, "Training Evaluation of the F-15 Advanced Air Combat Simulation (AL-TP-1991-00047), Williams Air Force Base, AZ: Armstrong Laboratory, Aircrew Training Research Division.

[NEFF95] Neff, K. Kaman Sciences Corporation, 2560 Huntington Ave. Alexandria VA 22303, E-Mail dated Sep 29, 1995.

[POST80] Postel, J. "User Datagram Protocol," USC/Information Sciences Institute, Internet Engineering Task Force Request for Comments 768, 28 August 1980.

[ROGE94] Rogers, D., "STOW-E Lessons Learned - Focused on the 3 Primary Army STOW-E Sites," Cubic Defense Systems, February 1995.

[RUMB91] Rumbaugh, J. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, N.J. :, c1991

[SGI94] Silicon Graphics, Inc. *IRIS Performer Programming Guide (version 1.2)* Silicon Graphics, Inc., Mountain View, CA, 1994.

[SHEA92] Sheasby, S. M. *Management of SIMNET and DIS Entities In Synthetic Environments.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH AFIT/GCS/ENG/92D-16, December 1992.

[SIMP91] Simpson, D. J. *An Application Of The Object-Oriented Paradigm To A Flight Simulator*. MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH AFIT/GCS/ENG/91D-22, December 1991.

[SING94] Shinghal, M., N. Shivaratri. *Advanced Concepts in Operating Systems*, McGraw-Hill, Inc., New York, NY, 1994.

[SWIT92] Switzer, J. C., *A Synthetic Environment Flight Simulator: The AFIT Virtual Cockpit*. MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH AFIT/GCS/ENG/92D-17, December 1991.

Appendix

Appendix A: Listing of Switcher.h

```
#ifndef _SWITCHER_H_
#define _SWITCHER_H_

////////////////////////////////////////////////////////////////////
// FILE:         FW_Switcher.h
//
// AUTHOR:       Capt Neal Schneider
//
// DATE:         21 July 1995
//
// DESCRIPTION:
////////////////////////////////////////////////////////////////////

#include "DIS_v2_cockpit_obj_mgr.h"
#include "DIS_v2_entity_obj_mgr.h"
#include "switcher_entity_class.h"
#include "ya_clock.h"

#define REQUEST_RESPONSE_TIMEOUT    5.0
#define TRANSFER_ACK_TIMEOUT      1.0

//-- Communication flags for the Switching capability

//char received_Transfer_Control_Request_PDU;     // These are declared in DIS_v2_entity_obj_mgr.h
//char received_Transfer_Control_PDU;             // and therefore are not needed here, but its nice
//char received_Transfer_Control_Acknowledge_PDU; // to know what they are for.


class Switcher {
public:

   Switcher() {Init();};
  ~Switcher();
   void Init();

   void Set_Receive_Man(entity_object_manager *net_manager);
   void Set_Send_Man  (DIS_v2_cockpit_object_manager *obj_manager);

   void Set_My_Info(unsigned short  target_id_site,
                    unsigned short  target_id_host,
                    unsigned short  target_id_entity);
   void Get_My_Info(unsigned short *target_id_site,
                    unsigned short *target_id_host,
                    unsigned short *target_id_entity);

protected:

   switcher_entity   *current,
                     *new_state;

   float              request_response_timer;
   float              request_ack_timer;
   int                request_response_timer_on;
   int                request_ack_timer_on;
   float              request_response_timeout_length;
   float              request_ack_timeout_length;

   ya_clock  clock;


   DIS_v2_cockpit_object_manager   *Send_Man;     //    Send Portion of Object Manager
   entity_object_manager           *Receive_Man;  // Receive Portion of Object Manager

};

#endif
```

Appendix B: Listing of switcher_entity_class.h

```
#ifndef _SWITCHER_ENTITY_CLASS_H_
#define _SWITCHER_ENTITY_CLASS_H_

//////////////////////////////////////////////////////////////////////
// FILE:        switcher_entity_class.h
//
// AUTHOR:      Capt Neal Schneider
//
// DATE:        21 July 1995
//
// DESCRIPTION:
//////////////////////////////////////////////////////////////////////

#include "entity_obj_mgr.h"

class switcher_entity  // : public vehicle
{

  public:

     switcher_entity()                              {};

     void set_site_id   (unsigned short site)       {site_id   = site;};
     void set_host_id   (unsigned short host)       {host_id   = host;};
     void set_entity_id (unsigned short entity)     {entity_id = entity;};

     unsigned short get_site_id ()                  {return site_id;};
     unsigned short get_host_id()                   {return host_id;};
     unsigned short get_entity_id()                 {return entity_id;};

  protected:

    unsigned short site_id;
    unsigned short host_id;
    unsigned short entity_id;
};

#endif
```

Appendix C: Listing of VC_Switcher.h

```
#ifndef _VC_SWITCHER_H_
#define _VC_SWITCHER_H_

/*********************************************************************
 * FILE:        VC_Switcher.h                                        *
 *                                                                   *
 * AUTHOR:      Capt Neal Schneider                                  *
 *                                                                   *
 * DATE:        21 July 1995                                         *
 *                                                                   *
 * DESCRIPTION:                                                      *
 *                                                                   *
 *********************************************************************/

#include "Switcher.h"

enum switcher_state_type {ready,
                          pending_request,
                          pending_transfer,
                          have_state,
                          pending_ack,
                          pending_ui_transfer,   // ui = user initiated
                          have_ui_state,
                          pending_ui_ack};

class VC_Switcher : public Switcher

{
  public:

       VC_Switcher();
   void Init();
   void Print_State();

    int Ack_Pending_Request();                          // App knows and

    int Request_Takeover(unsigned short target_id_site,
                         unsigned short target_id_host,
                        unsigned short target_id_entity ); // Help!


    int  Request_Control();
    int  Request_Control(unsigned short entity_id,
                         unsigned short target_id_host,
                         unsigned short target_id_entity); // Want to help (take control)


    int Relinquish_Control(double_vector_t   location,
                           Euler_vector_t    Euler_angles,
                           float_vector_t    linear_velocity,
                           float_vector_t    linear_acceleration,
                           float_vector_t    angular_velocity,
                          float32_t          q1,
                          float32_t          q2,
                          float32_t          q3,
                          float32_t          q4,
                          float32_t          throttle,
                          float32_t          ailerion,
                          float32_t          rudder,
                          float32_t          elevator,
                          float32_t          trim); // Send the information

   int Get_Transfer_Control_Info(double_vector_t    *location,
                                 Euler_vector_t     *Euler_angles,
                                 float_vector_t     *linear_velocity,
                                 float_vector_t     *linear_acceleration,
                                 float_vector_t     *angular_velocity,
                                 float32_t          *q1,
                                 float32_t          *q2,
                                 float32_t          *q3,
                                 float32_t          *q4,
                                 float32_t          *throttle,
```

```
                                float32_t          *aileron,
                                float32_t          *rudder,
                                float32_t          *elevator,
                                float32_t          *trim);

    int Confirm_Takeover();
    int Confirm_Control();

    //  This function checks the incomming messages and current state and
    //  returns the new state of the switcher object
    switcher_state_type Get_State();

  protected:

    switcher_state_type current_state;
    float               operator_response_timer;
    float               trigger_timer;  // used to give delay so swap will not be accidentally
                                        // followed by a swap request.
    switcher_entity    *swap_target;
};

#endif
```

Appendix D: Listing of FW_Switcher.h

```
#ifndef _FW_SWITCHER_H_
#define _FW_SWITCHER_H_

//////////////////////////////////////////////////////////////////
// FILE:        FW_Switcher.h
//
// AUTHOR:      Capt Neal Schneider
//
// DATE:        21 July 1995
//
// DESCRIPTION:
//////////////////////////////////////////////////////////////////

#include "Switcher.h"

enum switcher_state_type     {ready = 1,
                              pending_request = 2,
                              need_state = 3,
                              pending_transfer = 4,
                              have_state = 5,
                              pending_ack = 6,
                              pending_ui_transfer = 7,
                              have_ui_state = 8};

// Nothing new here.  The difference is in the definition of the
// virtual functions.

class FW_Switcher : public Switcher

{
  public:

        FW_Switcher();
    void Print_State();
    switcher_state_type Get_State();

    int  Request_Takeover     (unsigned short target_id_site,
                               unsigned short target_id_host,
                               unsigned short target_id_entity ); // Help!

    void Cancel_Request_Takeover();

    int  Request_Control      (unsigned short target_id_site,
                               unsigned short target_id_host,
                               unsigned short target_id_entity); // Want to help (take control)

    int  Relinquish_Control  (double_vector_t       location,
                              Euler_vector_t        Euler_angles,
                              float_vector_t        linear_velocity,
                              float_vector_t        linear_acceleration,
                              float_vector_t        angular_velocity,
                              float32_t             q1,
                              float32_t             q2,
                              float32_t             q3,
                              float32_t             q4,
                              float32_t             throttle,
                              float32_t             ailerion,
                              float32_t             rudder,
                              float32_t             elevator,
                              float32_t             trim); // Send the information

    int  Get_Transfer_Control_Info  (double_vector_t       *location,
                                     Euler_vector_t        *Euler_angles,
                                     float_vector_t        *linear_velocity,
                                     float_vector_t        *linear_acceleration,
                                     float_vector_t        *angular_velocity,
                                     float32_t             *q1,
                                     float32_t             *q2,
                                     float32_t             *q3,
                                     float32_t             *q4,
                                     float32_t             *throttle,
```

```
                                        float32_t             *aileron,
                                        float32_t             *rudder,
                                        float32_t             *elevator,
                                        float32_t             *trim);

    int Confirm_Takeover();
    int Confirm_Control();

    //  This method checks the status of the switcher, updates internal
    //  variables and returns the current status, letting the application
    //  know what to do next.

  protected:
    switcher_state_type current_state;
    int need_help;
};

#endif
```

Vita

Capt Neal W. Schneider [redacted]

He graduated from Lockhart High School in 1985 and entered undergraduate studies at Texas A&M at College Station, Texas. He graduated with a Bachelors of Science in Electrical Engineering in May 1989. He received his commission on 1 May 1989 upon completion of ROTC at Texas A&M. His first assignment was to Scott AFB as a weather radar evaluation team leader. His second assignment was on the same base with the Air Force Command, Control, Communications, and Computer Agency as a test engineer for C4 Systems. In May 1995, he entered the School of Engineering, Air Force Institute of Technology.

[redacted]

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 1995	Technical Report, Thesis

4. TITLE AND SUBTITLE
DYNAMIC TRANSFER OF CONTROL BETWEEN MANNED AND UNMANNED SIMULATION ACTORS

5. FUNDING NUMBERS

6. AUTHOR(S)
Captain Neal W. Schneider

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

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/95D-24

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)*

This thesis continues the ongoing research at the Air Force Institute of Technology's Virtual Environments Laboratory in the area of distributed simulation. As the relevance and interest of interactive simulation as a training medium continues to grow, there is a pressing need to provide more realistic and numerous intelligent autonomous agents for simulations. As those autonomous agents mature and become more realistic, the need exists to be able to handle individual agents by taking control of them and operating them as manned agents at certain points within the simulation. The author started with a protocol proposed in a working draft of the Distributed Interactive Simulation (DIS) Protocol Standard 2.1.1 (Draft). He demonstrates how this protocol can be improved by swapping control between two entities involved in a distributed simulation. The new protocol provides simultaneous transfer while being compatible with the one proposed in the draft standard.

The protocol is implemented on two applications developed in the Virtual Environments Laboratory, the Virtual Cockpit (VC) and the Automated Wingman (AW). The anticipated flow of execution begins with the AW requesting assistance. The operator of the VC then can reply by assuming control of the AW. Once the required human operation has been performed, the operator may switch back to the lead aircraft, completing the full cycle of execution.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Distributed Interactive Simulation, Virtual Cockpit, Autonomous Agents	67
	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.