EXPERIENCE WITH ADA ON THE F-18 HIGH ALPHA RESEARCH VEHICLE FLIGHT TEST PROGRAM

VICTORIA A. REGENIE, MICHAEL EARLS, AND JEANETTE LE
NASA Dryden Flight Research Facility
Edwards, California

MICHAEL THOMSON
PRC Inc.
Edwards, California

Summary

Considerable experience has been acquired with Ada at the NASA Dryden Flight Research Facility during the on-going High Alpha Technology Program. In this program, an F-18 aircraft has been highly modified by the addition of thrust-vectoring vanes to the airframe. In addition, substantial alteration was made in the original quadruplex flight control system. The result is the High Alpha Research Vehicle. An additional research flight control computer was incorporated in each of the four channels. Software for the research flight control computer was written in Ada. To date, six releases of this software have been flown. This paper provides a detailed description of the modifications to the research flight control system. Efficient ground-testing of the software was accomplished by using simulations that used the Ada for portions of their software. These simulations are also described. Modifying and transferring the Ada flight software to the software simulation configuration has allowed evaluation of this language. This paper also discusses such significant issues in using Ada as portability, modifiability, and testability as well as documentation requirements.

Nomenclature

A/D	analog-to-digital
D/A	digital-to-analog
DDI	digital display indicator
DPRAM	dual port random-access memory
EEPROM	electrically erasable programmable read-only memory
FAST	F-18 FCS automated software testing
FCS	flight control system
FORTRAN	FORmula TRANslation
GE	General Electric, Lynn, Massachusetts
HARV	High Alpha Research Vehicle
HUD	head-up display
JOVIAL	Jules' Own Version of the International Algorithmetic Language
LEF	leading-edge flaps
McAir	McDonnell Aircraft Division, McDonnell Douglas Corporation, St. Louis, Missouri
MDTOT	parameter identifier
MIL-STD	military standard
OBES	on-board excitation system
PASCAL	Philips Automatic Sequence CALculator
RAM	random-access memory
RFCS	research flight control system
ROM	read-only memory
TEF	trailing-edge flaps
UART	universal asynchronous receiver-transmitter
UMN	universal memory network
UVPROM	ultraviolet programmable read-only memory

Introduction

Higher order languages have not been extensively used to develop flight control systems because of the lack of speed and capacity in the flight control computers. With the large improvements in computer speed, or throughput, and in memory, use of higher order languages is now practical. Examples of higher order languages used for aircraft include PASCAL (Philips Automatic Sequence CALculator), JOVIAL (Jules' Own Version of the International Algorithmetic Language), and Ada.

Because the United States military selected Ada for use as the common language, more aircraft will be flown using this software. Thus, NASA Dryden Flight Research Facility (DFRF) personnel must become familiar with the language and its capabilities.

An F-18 testbed aircraft, the High Alpha Research Vehicle (HARV), offered an opportunity to acquire experience with the use of Ada for flight control applications.[1] The aircraft was built by the McDonnell Aircraft Division (McAir), McDonnell Douglas Corporation, St. Louis, Missouri, and the Northrop Corporation, Newbury Park, California.

This paper describes the DFRF experience with Ada and details the observed advantages and disadvantages to using this language. The conclusions reached here through the use of Ada in the real-time control environment are applicable to other control areas as well. Many real-time control systems using Ada to control complex systems would be expected to have similar experiences.

Research Flight Control System Description

The following subsections describe the hardware, control laws, and software of the system in which Ada was used:

Hardware

The HARV is a modified preproduction F-18 aircraft equipped with spin chute and emergency hydraulic and electrical systems. These modifications include a simple, low cost, thrust-vectoring system. This installation required modifications to the flight control system and mission computer.[2]

The basic F-18 flight control system consists of quadruplex redundant GE 701E (General Electric, Lynn, Massachusetts) computers and was modified for HARV by adding an analog interface to the thrust-vectoring vane actuators and a research flight control system (RFCS). Figure 1 shows the F-18 HARV computer architecture. The analog input card and RFCS were added to spare card slots in the basic flight control computer. This basic flight control computer maintains control of the aircraft; controls input, output, or both processing functions; communicates with the F-18 mission computer for outer loop control; and displays information through a military standard (MIL-STD) 1553 data bus. The RFCS was added to provide a flexible system for control law research. Ada was chosen as the programming language for the RFCS.

The RFCS central processing unit is a MIL-STD-1750A processor with a 20-MHz clock slaved to the GE 701E computer (Fig. 1). The RFCS contains 32,000 words of electrically erasable programmable read-only memory (EEPROM), 16,000 words of ultraviolet programmable read-only memory (UVPROM), 2,000 words of random-access memory (RAM), and 2,000 words of dual port RAM (DPRAM). The RFCS communicates to the basic flight control computer through the DPRAM. Hence, RFCS may be called an embedded control system. It, however, has no direct control of the aircraft. The aircraft is under RFCS control only during the research phases of a HARV flight. First, the RFCS is armed or enabled by a cockpit switch. Then, it is engaged or activated through use of a switch on the control stick. The RFCS is manually disengaged via the arm switch or a control-stick-mounted paddle switch. Autodisengagement occurs as a result of internally defined limits on rates, accelerations, engine sensors, and airdata sensors.

Control Laws

The longitudinal control laws contain an angle-of-attack command system that uses angle-of-attack, pitch rate, and inertial coupling feedbacks (Fig. 2).[3] The lateral–directional control laws contain a feet-on-the-floor stability axis roll rate command system (Fig. 3). This system provides the control for the roll and yaw axes.[3] The lateral–directional system uses roll, yaw, and sideslip rates as well as lateral acceleration and inertial coupling as feedback signals.

Figure 4 shows a simplified diagram of the thrust vane mixer section. This section converts the command pitch and yaw-vectoring moments computed in the longitudinal and lateral–directional control laws into vane commands. The mixer also uses estimated thrust and current vane positions to calculate new vane commands. The RFCS gross thrust estimator uses nozzle pressure ratio, nozzle exit radius, power level angle, and static pressure to calculate gross thrust.

Software

The RFCS software is programmed in Ada and was developed on a separate minicomputer system and cross-compiled to the MIL-STD-1750A processor. The software is loaded into the flight control computers through an RS232 serial port using a personal computer.

The original RFCS software was designed and tested by McDonnell Douglas Corporation under a NASA contract. None of the real-time kernel capabilities or elements available with Ada, such as taskings, priorities, terminations, and exceptions, were used for this system because of concerns about timing.[4–6] The RFCS software consists mainly of the control laws with a few redundancy management functions. Because it can always downmode safely to the F-18 basic flight control system, the RFCS is not considered flight critical. Choice of a language impacts neither the number of redundancy management functions nor their complexity. Redundancy management functions of the RFCS include such elements as reasonability checks and engage logic.

The RFCS software consists of approximately 78 Ada specifications, which define the interface to the outside world, and 13 Ada bodies, which give the details of the program. These specifications and bodies consist of approximately 130 modules, 175 procedures, 5 functions, and 4,600 lines of code (16,302 sixteen-bit words of EEPROM and 1,699 words of RAM). The RFCS software can be divided into six functional areas. These areas include input-output functions, disarm-disengage logic, longitudinal control laws, lateral–directional control laws, thrust vane mixer, and gross thrust estimator. Figure 5 shows these functional areas. Timing estimates for the current RFCS software indicate less than 85 percent worst case throughput and 50 percent memory use.

The input-output functional area transfers data through DPRAM, converts these data to and from a fixed point machine (the basic flight control system) to the RFCS floating point format, and checks for data validity. The disarm-disengage logic functional area determines whether the RFCS should arm or engage. This functional area includes such elements as envelope limits and reasonability checks on control law feedbacks and RFCS outputs.

Simulations

Three configurations of the real-time HARV simulation are used: an all-software, a hardware-in-the-loop, and an ironbird. Figures 6, 7, and 8 show that these configurations use many of the same elements. Detailed descriptions of these configurations are provided next.

All-Software Simulation

Written in FORmula TRANslation (FORTRAN) and Ada, the all-software simulation is used for engineering development of control laws, for pilot training, and for flight test planning. Figure 6 shows the elements of the all-software simulation. The aircraft model is performed in the simulation computer and includes the basic flight control laws as well as the aerodynamic, propulsion, thrust-vectoring, sensor, and actuator models. The only element of the all-software simulation coded in Ada is the RFCS control laws. These control laws are in the RFCS control law computer, a Unix-based workstation. Both the simulation computer and RFCS control law computer cycle at 80 Hz. The simulation cockpit includes the flight digital display indicators (DDI) and a head-up display (HUD) along with the simulated instrumentation and the pilot controls. Other flight hardware include mission computers and communication system control. An interface between the research flight control laws and the basic flight control laws in the simulation emulates the actual flight system interface as closely as possible. Four MIL-STD-1553 multiplex buses are included in the simulation. Three are for communication between the simulated and flight avionics. One is for an aircraft model display communication path. In addition, the three MIL-STD-1553 buses model the three HARV MIL-STD-1553 buses.

Hardware-in-the-Loop Simulation

Hardware-in-the-loop, the most frequently used simulation configuration, is the primary tool for developing and testing software. This configuration is also used for pilot training, flight test planning, and, to a lesser degree, engineering development. In addition, this configuration is extensively used for failure modes and effects testing and for control law validation. Actual flight control computers replace the control laws modeled in the simulation computer and the workstation. Figure 7 shows the hardware-in-the-loop simulation. Actuator models are also moved from the simulation computer and modeled using analog models. All other elements of the all-software simulation remain the same.

Ironbird Simulation

Figure 8 shows the ironbird simulation. As a final check for the system configuration, this simulation configuration is used to measure the closed-loop response of the control laws and to verify actuator models. A decommissioned F-18 airplane with hydraulic lines is used. With the exception of the leading- and trailing-edge flaps (LEF and TEF), the ironbird simulation replaces the analog actuator models with the actual flight actuators.

Compilers

Two Ada compilers were used: one for the RFCS software and the other for the simulation software. The cross-compiler used for the RFCS is a TLD Systems, Limited, Torrance, California, compiler hosted on a minicomputer. This compiler conforms with MIL-STD-1815A-1983 requirements. For the simulation software, a SunPro (Sun Microsystems, Incorporated, Mountainview, California) Ada language compiler is used. This compiler also conforms to MIL-STD-1815A-1983 requirements. No evaluation was done on the different compilers, and only obvious differences, such as one compiler flagging errors that the other compiler missed, were noted.

Software Modifications

Two major areas of software modifications are discussed in this section. These areas include modifications to the flight software and adaptations of the flight software to the simulation. McAir developed the RFCS software in a simulation and then transferred it to the flight hardware. The DFRF tested the software in the

hardware-in-the-loop simulation and later added the RFCS software to the all-software simulation.

Flight Software Modifications

The RFCS software delivered from McAir to DFRF was not tested in a closed-loop system but was verified by McAir in an open-loop environment on the flight hardware. The contract stated that NASA would complete the closed-loop validation testing. The compiler that McAir and DFRF used to develop the Ada software includes a profiling tool that allows timing estimates to be generated for the target computer. Results of the timing estimates made by McAir using this tool significantly underestimated the actual execution time in the MIL-STD-1750A computer. McAir modified the RFCS software during the open-loop tests to improve its throughput. When the RFCS was delivered, it was installed in the hardware-in-the-loop simulation for validation testing. During the hardware-in-the-loop validation testing, RFCS exceeded the allocated cycle time for one unusual set of conditions. The code required modification to allow some throughput margin.

The following list shows the changes made to the RFCS software to date. Several functions were changed from 80- to 40- and 20-Hz functions (items 1 and 2 in list). At the same time, the code was reviewed to find additional changes to increase the throughput margin (item 3 in list).

1. RFCS multirate tasking
2. Modify order of rate structure
3. RFCS Ada code cleanup
4. Code reconfiguration
5. Change mixer-predictor constant
6. Thrust estimation modification
7. Betadot sign change miscompare
8. On-board excitation system (OBES)
9. RFCS 701E fader gain
10. Fix OBES frequency sweeps by overlay*
11. Fix OBES frequency sweeps and cleanup syntax
12. Static pressure with weight on wheels
13. Fix RFCS flag word outside envelope indication
14. OBES requirements
15. Incorrect differential stabilator, TEF, and LEF computations
16. MDTOT sign change
17. OBES cleanup
18. Persistence on betadot and angle of attack
19. Engine parameters channel 1/3 miscompare
20. Change instrumentation scaling of error signal
21. Update configuration identification to version 24.0
22. Sideslip rate delta tolerance—overlay*
23. Sideslip rate delta tolerance—compile
24. Add test variables for FAST command limit tests
25. Replace message 8 RFCS parameters
26. Change scales of angles of attack and sideslip in RFCS
27. Change parameters for angle of attack and inertial navigation system angle of attack scaling to $\pm 180°$
28. OBES aileron rate limit
29. Add component of alphadot and betadot in mission computer
30. Parameter identification OBES
31. Move RFCS message 17 code
32. Thrust estimator
33. Enable RFCS go
34. Angle-of-attack filter coefficient
35. Message 17 parameter change
36. Message 8 RFCS modification
37. RFCS persistence counter for channel 1/3 miscompare
38. Change constants in pitch and roll trim processing
39. RFCS scaling for message 8 instrumentation
40. Static pressure with weight on wheels by overlay*
41. Downlink OBES signal
42. Change configuration identification to version 22.0 in RFCS software
43. Version 22.0—message 8
44. Change instrumentation error signal by overlay*
45. Pitch rate lead and gain changes

*Indicates an overlay generated.

46. Update configuration identification to version 23.0
47. Message 8 word 20-bit toggle
48. RFCS thrust failures
49. Sideslip rate delta on instrumentation
50. Angle-of-attack rate gain fix
51. Update fade rate
52. Angle-of-attack scaling and inertial components—overlay*
53. Update configuration identification to version 25.0
54. Add test variables for FAST command limit tests
55. Parameter identification OBES modification
56. Update configuration identification to version 26.0
57. OBES command limiting
58. Collective trailing-edge flap test command

*Indicates an overlay generated.

The original RFCS control law software was developed as two parts: longitudinal and lateral–directional. While the delivered code was modularized, some functions were distributed through several modules. Airdata was the principal segment calculated in more than one module and was processed in the input-output and in the lateral–directional control law sections. To allow completion of updates to one functional group without affecting another functional group, the software was modified to include all airdata functions in input-output (item 4 in list). The update rate for airdata-dependent gain scheduling was at 80 Hz, but airdata was updated at 20 Hz. Consequently to increase the throughput margin, the code was modified to update the airdata-dependent gains at 20 Hz. Unless better profiling tools are developed, these problems in throughput margin will continue to be found in final hardware-in-the-loop testing.

During the flight program, modifications were made to correct problems or make improvements. The majority of these changes involved a simple constant or a couple of line changes. A few were more extensive and included new capabilities. An OBES was incorporated in RFCS to generate commands to the surfaces using a function generator for sine waves and doublets.

Simulation Software Modifications

The RFCS Ada code was ported to the software simulation. This code was developed on a minicomputer system and ported to a computer where it could be validated using the real-time, all-software simulation. Because the simulations were developed on the simulation computer, the Ada RFCS code was initially ported to this computer where it could interface with the residing simulation through shared memory. The simulation computer was incapable of supporting the Ada code in the time required. The code was then ported to a Unix-based workstation RFCS control law computer with a different Ada compiler. Here, the RFCS code communicated with the simulation computer through the universal memory network (UMN) instead of shared memory.[7] Real-time performance speed improved significantly on this computer. This performance improvement was the result of several factors. These factors included the limited time available on the simulation computer and the improved Ada compiler available on the RFCS control law computer.

Additional code was added to set-up a means of exchanging data between the Ada RFCS code and the real-time simulation. Because of timing restrictions, the calling order of the routines in the executive RFCS program was also changed. The hardware-in-the-loop code's executive operates at a 160-Hz frame rate overall. The individual routines are called at various rates. Originally, two 80-Hz tasks ran alternately on an even or an odd frame. One task handled the longitudinal control laws, while the other frame handled the lateral–directional control laws. The Ada on the RFCS control law computer was unable to support the 160-Hz schedule without time overruns. As a result, the even-odd-frame arrangement was replaced by a new calling sequence. This sequence first calls the longitudinal mode calculations and then calls the lateral–directional mode calculations. Otherwise, the source code developed on the minicomputer system is easily transferred to the RFCS control law computer.

Significant Issues

This section describes major issues relating to Ada and its use in real-time embedded control systems. These issues include porting, documenting, modifying, and testing the software. In addition, software development is discussed.

Portability

The RFCS Ada code was fairly portable. This code was transferred from the MIL-STD-1750A processor to the simulation computer to the control law computer. The majority of modifications needed for Ada to run in the simulation were changes to account for differences in the flight control and simulation systems. Because the hardware-in-the-loop RFCS source code resides on the minicomputer system and the all-software code is on the Unix-based workstation, two Ada compilers were used to achieve optimal performance on the individual machines. Use of two compilers can also

result in differences if one compiler is more nearly accurate than the other. For example, the Unix-based compiler would flag errors that the minicomputer compiler would accept. The two compilers provided an extra test for errors in the Ada software.

Documentability

An often mentioned feature of Ada is the fact that it is a self-documenting code. Although very easy to read, Ada is self-documenting only on a detailed level; that is, Ada is more similar to self-commenting. The self-documenting feature of Ada does not remove the need for developing specifications and system documentation. Any system requires a specification for the software to be developed against; otherwise, errors propagate throughout the system. Use of a higher order language, such as Ada, makes it easier to design and code a system without developing specifications. As with any other programming language, such program specifications as specification block diagrams, program requirements, software design specifications, and program flowcharts are needed to give an overall picture of the entire system.

Modifiability

Use of Ada or any higher order language simplifies all but the most difficult software updates. The compiler can show the assembly-level code along with the Ada, which helps when trying to understand the operation of the software. An assembly-level listing is necessary when the software is not performing as expected, and debugging is required. The assembly-level listing and the memory map are used to examine the system memory and to assist in locating errors. This technique was used several times during the system integration stage. The Ada code proved fairly easy to modify, but assembly-level modifications were still used.

Updates to the RFCS software are done either by overlay or by recompiling. To change constants, an overlay is performed. For an overlay, no source code is changed. The majority of overlays are then added to later software versions by modifying the source code and recompiling. Load files, the machine code in hexadecimal that is loaded onto the flight control computers, are updated on the minicomputer system. Once completed, the newly overlaid code is downloaded to the flight control computers. Because a recompile is not performed, a bit-for-bit comparison can be done to verify any memory changes.

For all other changes, the program is recompiled. This process involves changing the source code to meet the new requirements. Once the changes have been added to the code, a compilation is performed. Then, the new software is downloaded to the flight control computers. Software changes made by recompiling require significantly more testing than those done by overlay. Because a bit-for-bit comparison cannot be performed, it cannot be assumed that the source code updates did not affect any other software functionality.

One disadvantage in using Ada is that changes in the calling sequence, addition of new routines to the code, or both require changes in the compilation order of the dependent routines. The proper order or sequence must be established to ensure that any routine which depends on another routine is compiled before the calling routine is compiled. This ordering process can become a difficult task when major changes in the calling sequence are required.

Another disadvantage of higher order languages versus assembly languages is that software overlays cannot be inserted on-line. With assembly language, a logic overlay can be inserted into the source code and reassembled. Overlays can be written to branch to a predetermined patch area in read-only memory (ROM), execute the new code, and return to the point of origin. This type of change requires less testing than a complete reassembly because a bit-for-bit comparison can be performed.

Testability

The language used to implement the software has no impact on the testing requirements. The level of testing required is determined by the criticality of the system. Obviously, flight-critical systems require more testing than those systems that are less essential. Regardless of the programming language used, verification and validation tests are required to flight qualify a new software release. Verification is the process of determining that the software performs as specified. This process is accomplished by devising individual tests for each specified software task, conducting the test, and observing that the task was completed according to the specification. Validation, the broader task, seeks to determine if the system of which the software is a part performs adequately to fulfill the flight requirements. Open- and closed-loop failure modes and effects tests are among the techniques used in software validation. In these tests, failures are artificially induced, and a correct system response to those failures is verified.

Verification. When a higher order language is used, the compiler and linker must provide outputs which give the tester the information required to understand and verify the code. This information includes a listing of the assembly language code generated by the compiler and a memory map showing the locations of all modules, constants, and variables. The ability to complete the testing without modifying in any way the code under test is highly desirable. If the required test interfaces exist, then the locations of the input and output variables provide the interfaces to the code under

test. The tester may inject and monitor inputs and outputs to determine if the code performs as specified. If modification of the software is necessary to allow the tests to be performed, then a test patch is written.

Digital flight control systems seldom have the test interfaces required to perform complete verification testing without modification of the code under test. Of course in many instances, the change being verified involves inputs and outputs which are available during normal system operation. Test patches are not required in these cases. When test patches are required for higher order languages, these patches are coded in assembly language using areas of program and variable memory that are not used by the compiled software. The software under test is minimally impacted.

Validation. Software is validated in conjunction with the system of which it is a part. In the case of the RFCS, validation is accomplished on the HARV hardware-in-the-loop simulation. Time histories, failure modes, and effects tests are performed while the simulated aircraft is flying closed-loop. Depending on the interfaces available, occasionally test patches are needed to simulate system failures which cannot be induced in any other way.

Software Development

Development of real-time code requires an understanding of the requirements and limitations of memory and time. Real-time software generally requires more time than is readily available; therefore, care must be taken in developing the code. Use of a higher order language makes it more difficult to control the timing directly. The compiler generates the code and, even if optimized, may not produce the most time-efficient code. As discussed in the Compiler section and in the Portability subsection, one of the two compilers used by HARV detects more errors than the other. Although not required, use of two compilers provides a good check-and-balance scheme for any software development.

The use of two or more compilers is not required and was only used on this program to facilitate the transfer of the Ada software to the all-software simulation. The majority of the Ada software in the all-software simulation is identical to the flight software. Using the same software in the simulation and in the flight software saves time when transferring the software between systems. Software implementation differences between the hardware-in-the-loop and all-software simulations are also minimized.

The developer also needs to be aware of any microcode errors within the target processor. Many compiler developers work closely with processor manufacturers. Such cooperation allows the developers to correct microcode errors within the compiler, but not all errors will be necessarily corrected. Validated Ada compilers can also have errors. The assembly-code listing also gives the implementer the information required to deal with possible compiler errors and with known microcode errors in the target processor hardware. Knowledge of the system is still necessary for the development of software for real-time systems.

Concluding Remarks

The NASA Dryden Flight Research Facility experience with using Ada software for the F-18 High Alpha Research Vehicle has been positive. Although the Ada software developed was not for an extremely complex system, it is representative of most uses. Compiled Ada code can be used in a flight-critical system. The conclusions reached in this paper are not effected by the lack of a complex redundancy management or of a flight-critical system.

Positive conclusions reached concerning Ada are listed next. Ada is

- Portable—Ada was transferred among three computers using different compilers. The changes made to the transported code were to account for system changes.
- Documentable—For commenting purposes, this easy-to-read code is self-documenting. On the other hand, the self-documenting feature of Ada does not remove the requirement for system-level documentation or for a specification before coding.
- Modifiable—Ada is easy to modify, but it is still easier to make simple constant changes without recompiling. Individual changes in the code that are of major significance and numerous changes that are of less significance are easy to accomplish in Ada.
- Testable—Ada is no more difficult to test than any other language. The criticality of the system—not the language used to program the system—defines the testing requirements. Any system can be coded in Ada. For example, a system with complex redundancy management functions can easily be written in Ada, and the testing requirements would not change. A flight-critical system can easily use Ada, and the testing requirements would be the same as for other flight-critical systems.

Negative factors identified were not really Ada specific; that is, these factors are also found in other higher order languages. If a system does not follow standard software design practices, then problems will occur. Software and system specifications must be developed

before the software implementations. Compilers, even validated Ada compilers, can have errors. As a result, compiled software must be tested before use.

References

[1]Regenie, Victoria, Donald Gatlin, Robert Kempel, and Neil Matheny, "The F-18 High Alpha Research Vehicle: A High-Angle-of-Attack Testbed Aircraft," AIAA-92-4121, Aug. 1992. (Also available as NASA TM-104253, 1992.)

[2]Chacon, Vince, Joseph W. Pahle, and Victoria A. Regenie, *Validation of the F-18 High Alpha Research Vehicle Flight Control and Avionics Systems Modifications*, NASA TM-101723, 1990.

[3]Pahle, Joseph W., Bruce Powers, Victoria Regenie, Vince Chacon, Steve Degroote, and Steven Murnyak, *Research Flight-Control System Development for the F-18 High Alpha Research Vehicle*, NASA TM-104232, 1991.

[4]Honeywell Inc., Military Avionics Division, *DIGTAC III—Advanced Fault Tolerant Control Techniques: Software Final Report*, St. Louis Park, MN, Sept. 1990.

[5]Sodano, Nancy M., *Ada Realtime Performance Assessment Internal Research and Development Task: Final Report*, CSDL–C–5808, Charles Stark Draper Laboratory, Inc., Cambridge, MA, Oct. 1985.

[6]Software Productivity Consortium, *Ada Quality and Style: Guidelines for Professional Programmers*, SPC–91061–N, version 02.00.02, Herndon, VA, 1991.

[7]Reinwald, Carl, "Universal Memory Network Overview," *Universal Memory Network—Standalone Memory Interface (SMI–32) System Technical Manual*, TM/SMI32/001/00, Computer Sciences Corporation, Lompoc, CA, June 1,1992, pp. D–2 to D–12.

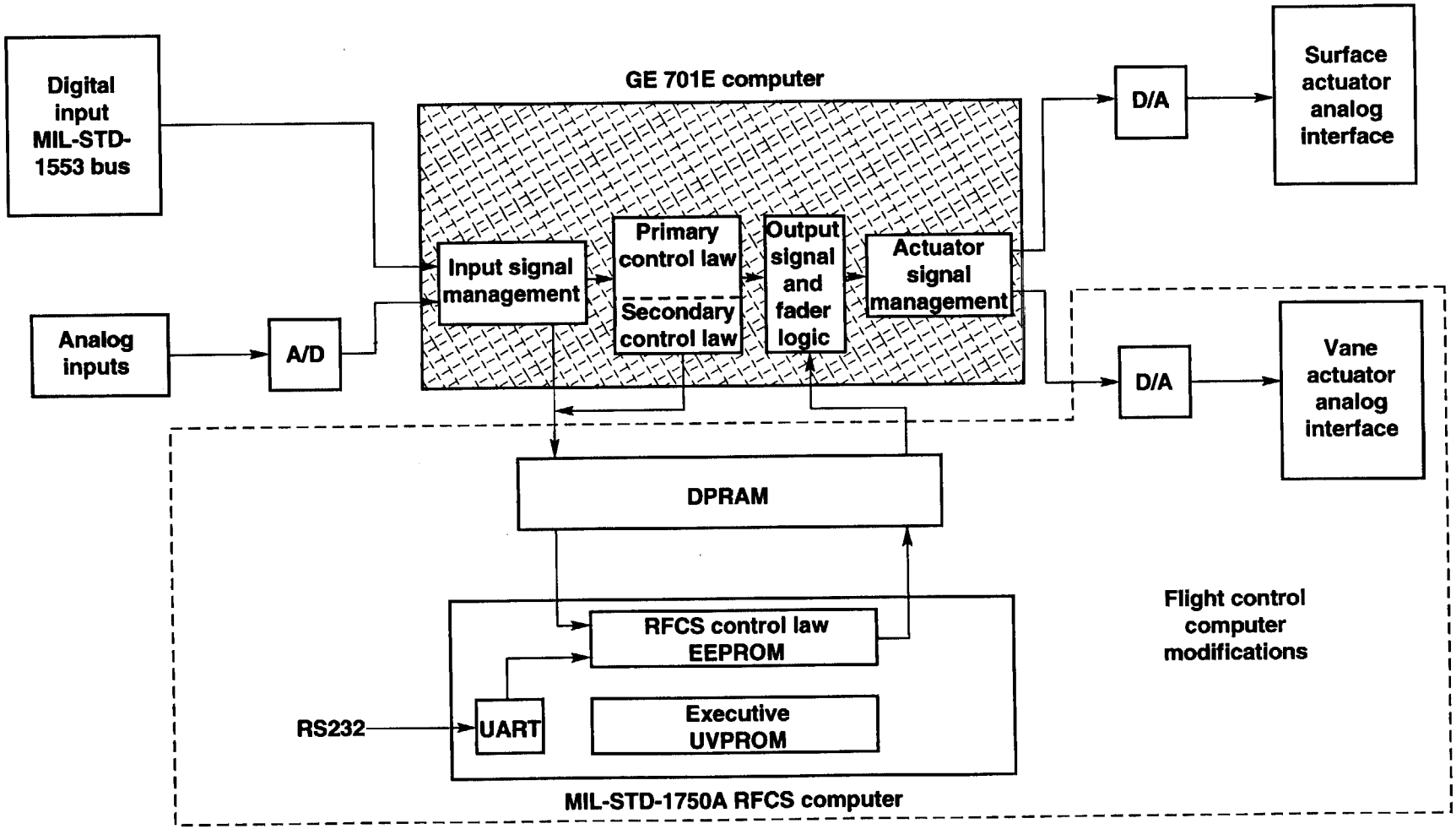

Fig. 1 The GE 701E and MIL-STD-1750A flight control computers.

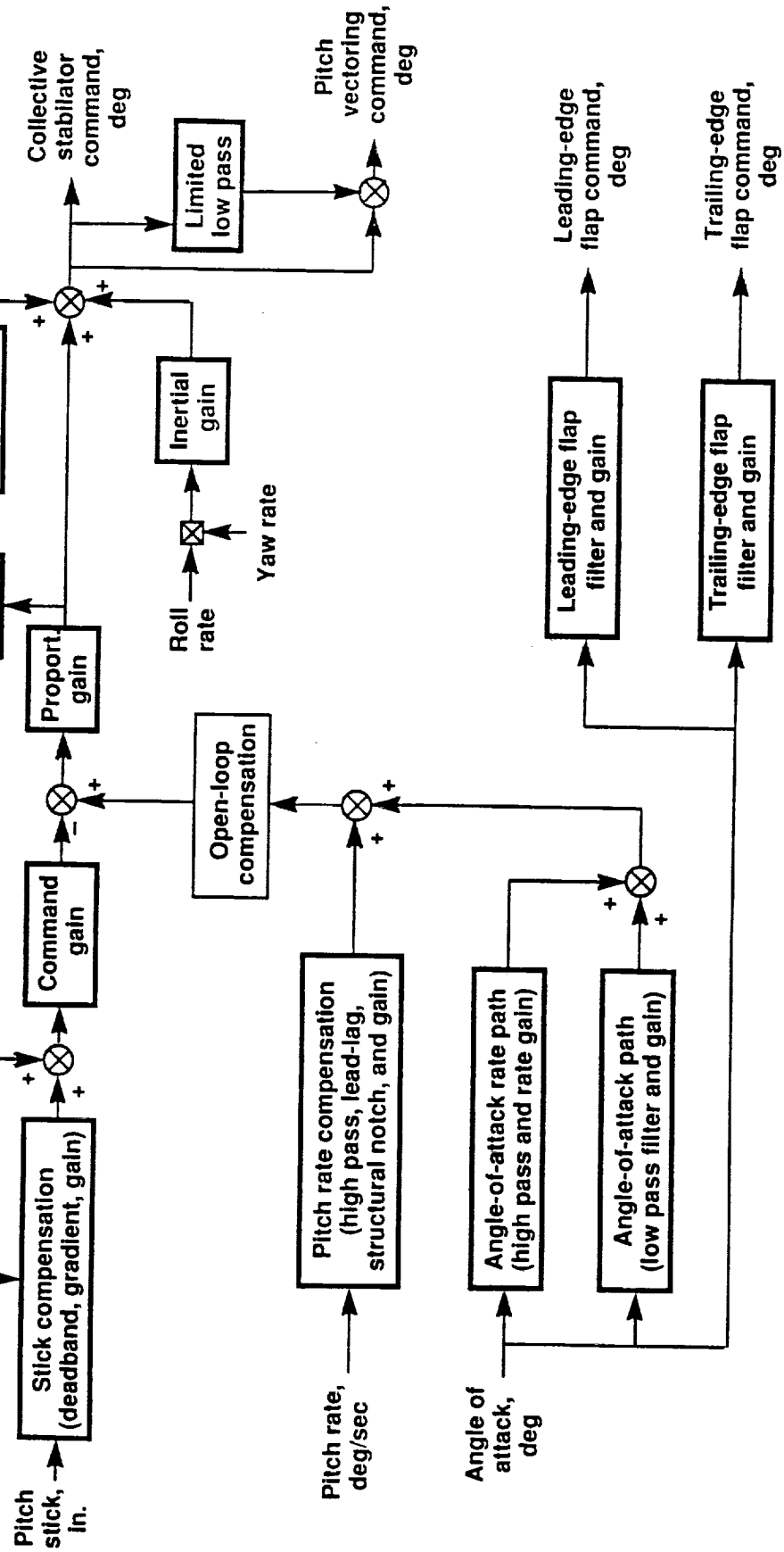

Fig. 2 Simplified research flight control system longitudinal control laws (angle-of-attack).

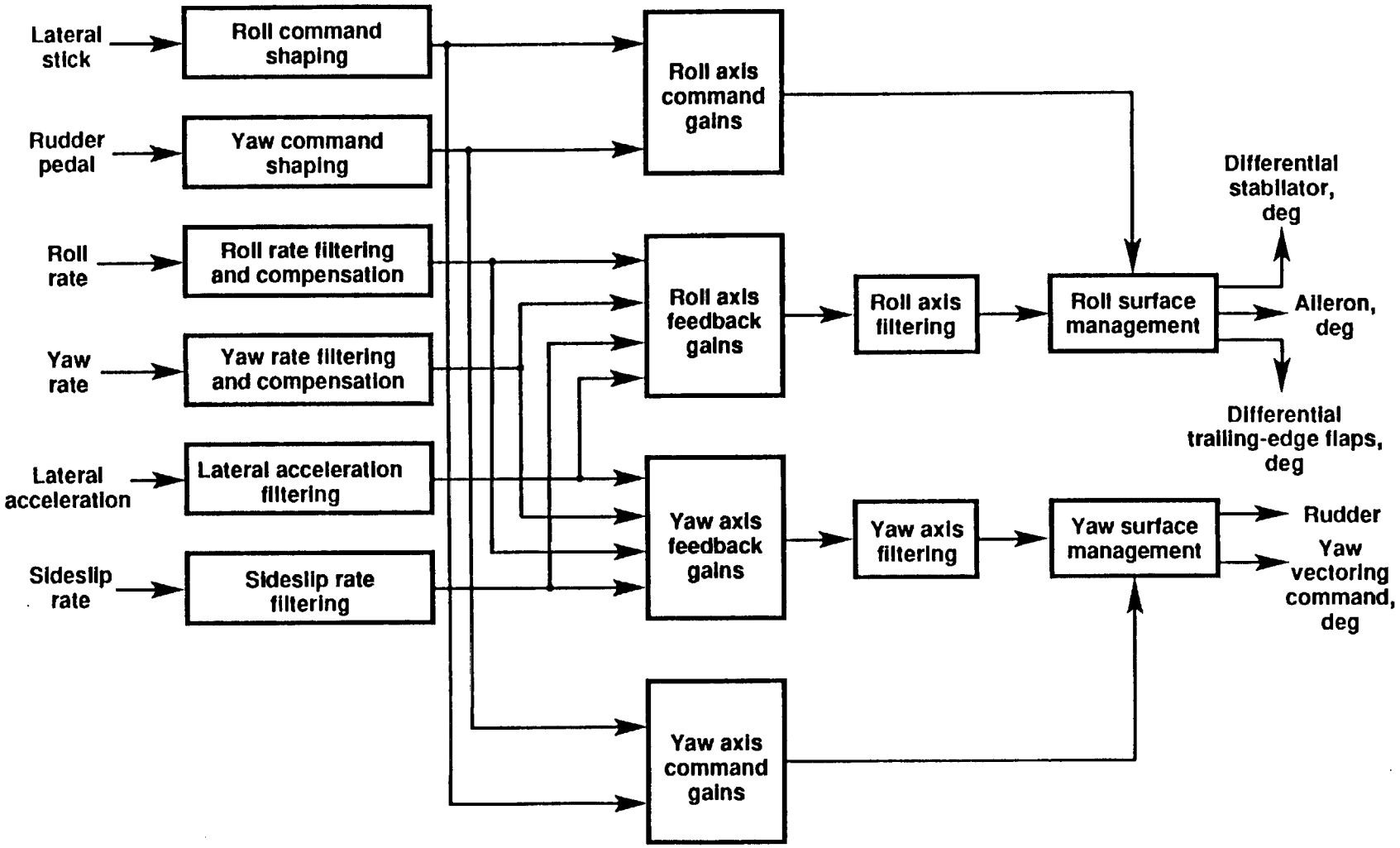

Fig. 3 Simplified research flight control system lateral—directional control laws.

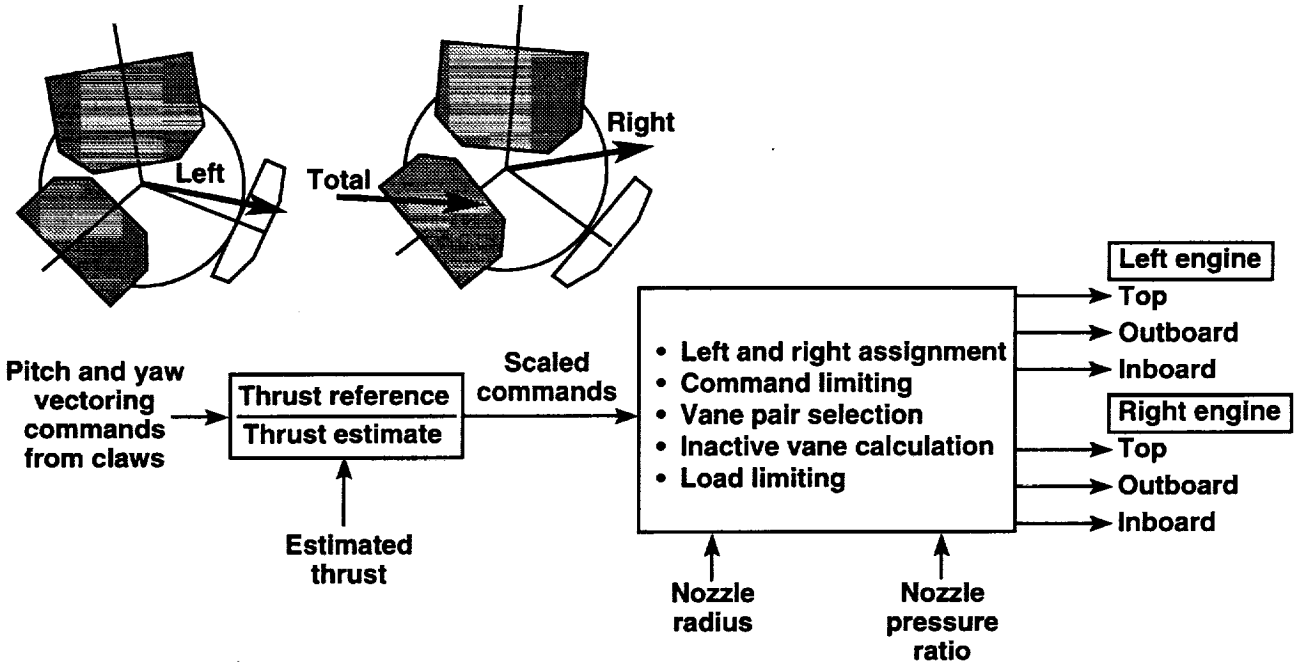

Fig. 4 Simplified thrust mixer.

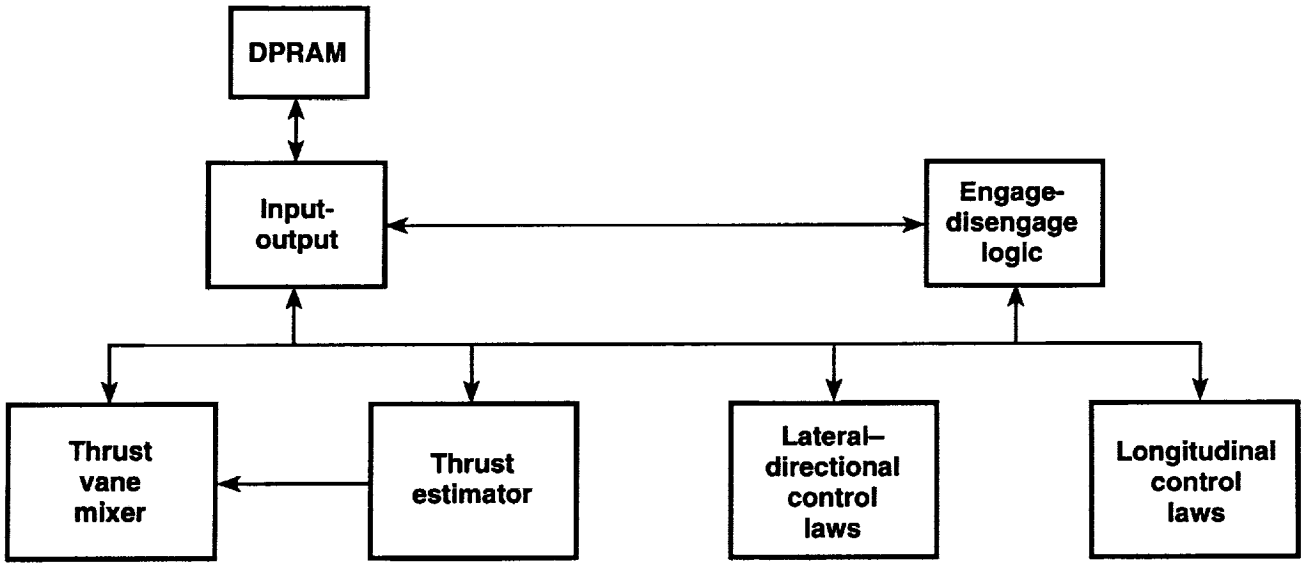

Fig. 5 The research flight control system software functional areas.

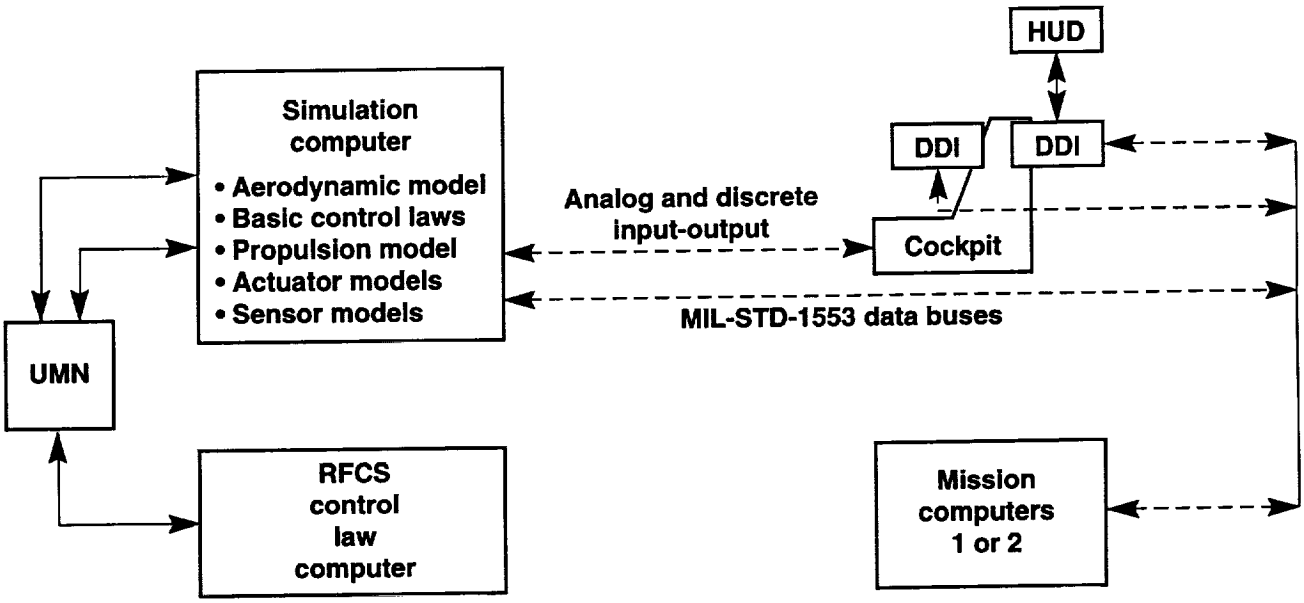

Fig. 6 The High Alpha Research Vehicle all-software simulation.

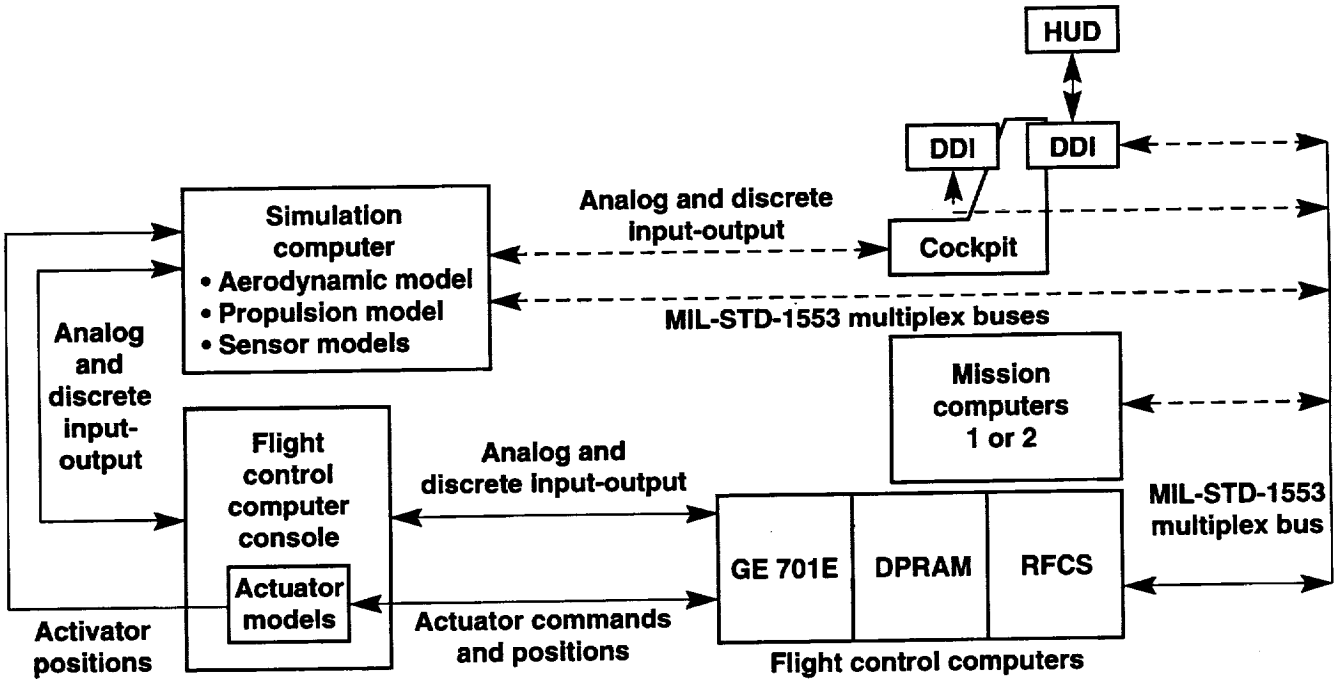

Fig. 7 The High Alpha Research Vehicle hardware-in-the-loop simulation.

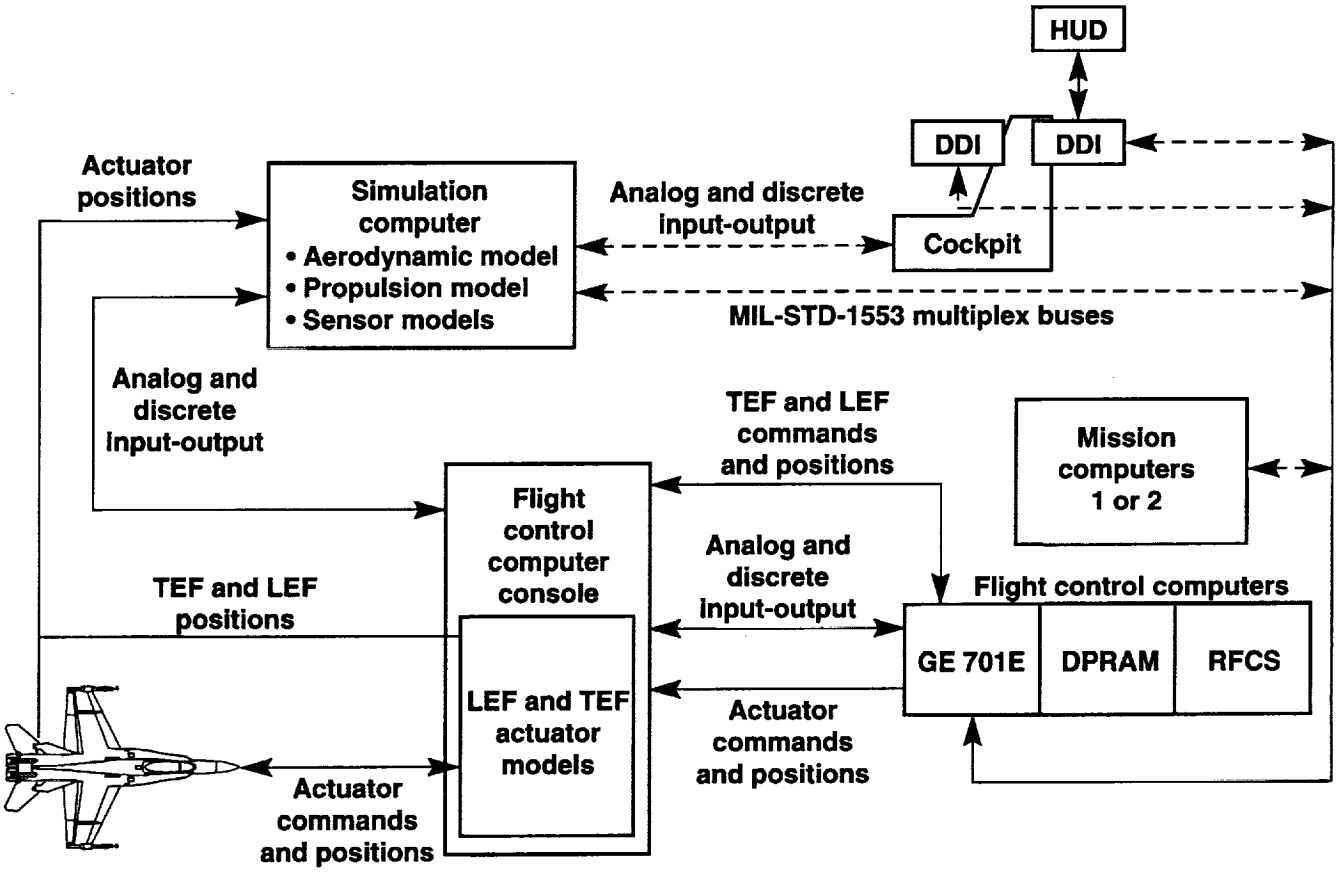

Fig. 8 The High Alpha Research Vehicle ironbird simulation.