SOFTWARE ENGINEERING FOR ADA-BASED TRAINING SYSTEMS:

LESSONS LEARNED FROM THE ADA SIMULATOR VALIDATION PROGRAM

Michael Caffey
Marshall Westerby

BURTEK
Tulsa, Oklahoma

ABSTRACT

This paper presents lessons learned from a Government funded research project to investigate the impact of using Ada for flight simulators. The DOD directive requiring the use of Ada for all mission critical software systems will cause a significant change in the way software is designed in the future. In order to study the impact of using Ada and software engineering techniques, the DOD funded the Ada Simulator Validation Program (ASVP). For the ASVP, simulation software previously written in FORTRAN was redesigned using software engineering methods and coded in Ada. The software was fully integrated and tested to evaluate the usefulness of Ada and software engineering practices. This paper presents the lessons learned by Burtek while redeveloping the software on the Ada Simulator Validation Program.

The four main topics that will be addressed are the design approach, computer system support environment, software maintainability, and training. The discussion on the design approach will address the use of object oriented design for designing real-time software, the role rapid prototyping plays in the design process, and the benefits that were derived from the rigorous design effort. The computer system support environment discussion will cover APSE tools required for developing simulation software as well as the need for operating systems that are specifically designed to support Ada. Software maintainability will be addressed, and special emphasis will be placed on the need for maintainability to be the driving factor during design. Tradeoffs between maintainability and efficiency are discussed, and a word of caution is given regarding the haphazard use of Ada features. Finally, a discussion on training will highlight the need for training in software engineering as well as in Ada syntax. The importance of training in the area of application will also be addressed.

INTRODUCTION

This paper presents the lessons learned from a government funded research project to investigate the impact of using Ada and software engineering techniques for developing real-time simulation software. The project, called the Ada Simulator Validation Program (ASVP), was funded by the Aeronautical Systems Division (ASD/YWB) of the United States Air Force. The purpose of the ASVP was to collect information for use in the future procurement of Ada-based training systems. Of primary importance was information on design approaches, computer system support environment, training required for Ada programmers, and the maintainability of simulation software written in Ada.

During the ASVP, simulation software previously developed in FORTRAN for a C-141B Operational Flight Trainer was redeveloped in Ada. This redevelopment was far more than a mere translation from FORTRAN to Ada. The requirements of the ASVP called for the software to be redesigned to implement, as much as possible, the software engineering concepts supported by the Ada language. Thus, the FORTRAN software was used as a source of math models for the simulator systems, however, the software structure used in the FORTRAN implementation was discarded and a new structure reflecting software engineering concepts was developed. The redeveloped software was fully integrated and tested, and a demonstration of the working simulator was performed in December 1987 at Altus Air Force Base. The final product consisted of approximately 65,000 lines of Ada software running on a dual processor Gould 9780 computer system.

The experience on the ASVP indicates that the use of Ada will have a significant effect on the way software is designed in the future. Project schedules, the distribution of manpower, and the Work Breakdown Structure used for tracking charges must all be modified to effectively manage the development of Ada software. In addition, engineers, managers, and quality assurance personnel will all require training in the use of software engineering techniques and how these techniques affect their responsibilities in the software development process. Contractors will be forced to deal with the cost of a significant learning curve on their first Ada project. The real benefits of using Ada will not be realized until late in the first project, or perhaps even the second.

This paper describes some of the main lessons learned by Burtek while developing software on the ASVP. The four topics that will be addressed are the design approach, the computer system support

environment, the maintainability of Ada software, and the training required for Ada programmers. This information is intended to provide engineers, managers, quality assurance personnel, and others with some insight into the critical issues associated with developing software in Ada.

DESIGN APPROACH

The primary emphasis of the ASVP was the evaluation of design approaches/methods that could be used for developing real-time software in Ada. Thus, early in the project, considerable effort was devoted to studying software engineering concepts and the design methods, such as Object Oriented Design (OOD), that support these concepts. During this study Burtek found that classical OOD was difficult to understand and not entirely appropriate for designing complex critical real-time systems. As a result, considerable effort was devoted to clarifying the steps in OOD and enhancing the method for use in designing real-time software. The result was the development of a modified version of OOD that was used to redesign the simulator software.

During the redevelopment effort, the design team found that the use of Ada and OOD increased the effort required for software design, but reduced the effort required for coding, testing, and integration. The redistribution of effort on the ASVP was primarily due to three factors :

1. The use of OOD encouraged engineers to carefully choose the components of their systems. OOD provided well defined criteria for choosing objects during the design process. As a result, the individual components of the ASVP software were well-bounded and had well defined interfaces. This increased the effort required for design, but significantly reduced the number of problems encountered during test and integration.

2. More effort was required to define the structure of the Ada software than is typically required to define the structure of a FORTRAN system. For the most part, simulation software written in FORTRAN has been architecturally simple. These systems have usually consisted of a relatively small number of large subroutines that communicate through a few common blocks. Very little effort was required to define the software structure.

   To effectively use Ada and generate object-oriented software, considerable effort must be devoted to developing a software structure that enforces software engineering concepts, and also works within the implementation constraints of the computer system. While this increases design time, the software design is more robust and maintainable. Thus less effort is required for test and integration.

3. Ada language features prevented many of the problems experienced during typical FORTRAN projects. These features, such as strong typing checking and strong library checking, prevented many of the mistakes that are frequently made when developing FORTRAN software. This significantly reduced the time required for testing and integration.

In the early stages of the redevelopment effort the project management made a concerted effort to ensure that the engineering team did not begin coding until the software design was complete. Though they successfully prevented the engineers from coding, it was very difficult to convince the engineers that a requirements analysis should be performed before any design is attempted. Requirements analysis is a difficult, tedious task that engineers are reluctant to perform. Nevertheless this task is very important. The requirements should drive the design. The lack of an understandable comprehensive requirements document indicates that the goals of the project are not adequately defined.

During the design phase, the engineering team found that the concepts enforced by a design method are far more important than the actual steps in the method. In order to effectively use OOD, an engineer must clearly understand the characteristics of a "good object." The engineer must be able to visualize the system as a collection of objects and relationships between the objects. Once the engineer understands these concepts the design method provides structure and discipline for the thought process. Without a good understanding of these concepts an engineer can use an object-oriented design method to produce a design that is not object-oriented at all.

Progress was very difficult to measure during the early stages of design. It was often difficult to discern whether the design team was making progress or merely experiencing "analysis paralysis." Requiring documentation for certain steps during each iteration of the design method can provide some means of measuring progress. However, management must realize that the decisions made early in design can have a significant affect on the elegance of the software design. Forcing the design team to hurriedly make these decisions could have an undesirable affect on the quality of the final product and the effort required during the latter phases of the project.

It is important to realize that no one design method will solve every problem. Throughout the design phase on the ASVP, engineers were strongly encouraged to follow the steps in the design method exactly. Although this is generally a good practice, there were times when it was necessary for the engineers to make design decisions that did not naturally follow the application of the design method. The strong emphasis on performing every step of the method sometimes caused engineers to spend considerable time trying to show how these decisions fit the methods steps. In retrospect, the steps in the design method should be viewed as guidelines rather than rules. Engineers should be prepared to justify

any deviation from the design method, however, management and quality assurance must be willing to accept that some design decisions will not perfectly fit the steps in the design method.

During the design phase, prototyping should be performed to verify that the design being generated is feasible. There is a definite conflict between many of the goals of software engineering, such as maintainability and reusability, and the unique requirements of real-time systems. Systems designed to fully enforce software engineering concepts are usually highly modular and architecturally complex. These systems perform extensive run-time checking to verify the correctness of computations and to prevent system malfunctions. In addition the sharing of common data between subsystems, which is common in real-time systems, is usually minimized in favor of parameter passing where the interfaces between the subsystems can be verified during compilation. While these characteristics can improve the maintainability and the integrity of the software, they can also introduce unacceptable run-time overheads and are not always easily implemented in a multiprocessing environment.

The design team made a concerted effort to fully enforce software engineering concepts during the design phase of the ASVP. The Engineers attempted to ignore implementation details and focus their attention on identifying design abstractions (objects) and characterizing the relationships (actions) that exist between These abstractions. Figure 1 illustrates this concept. The object called the Hydraulic System performs the action of pressurizing the Flight Controls. The Engines System performs the action of powering the Hydraulic System. Implementation details for the system were not to be considered during the design.

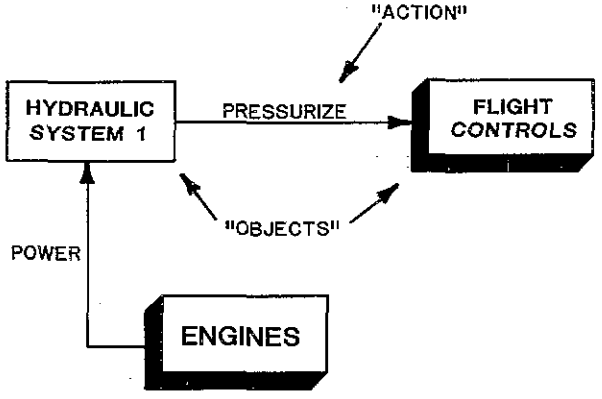

Figure 1. Software Design

A relatively straightforward and elegant transformation can be performed to convert this type of design to code. However, during the implementation phase, the team found that, due to real-time considerations and the constraints of a multiprocessor computer system, the straightforward elegant implementation of the design was not suitable. An additional software link was required for the interface between the components. Figure 2 illustrates the use of this lin . Prototyping during the design phase could have highlighted the problems with the design approach. The design, and the associated documentation, could have been tailored to provide a smooth transition from design to code.

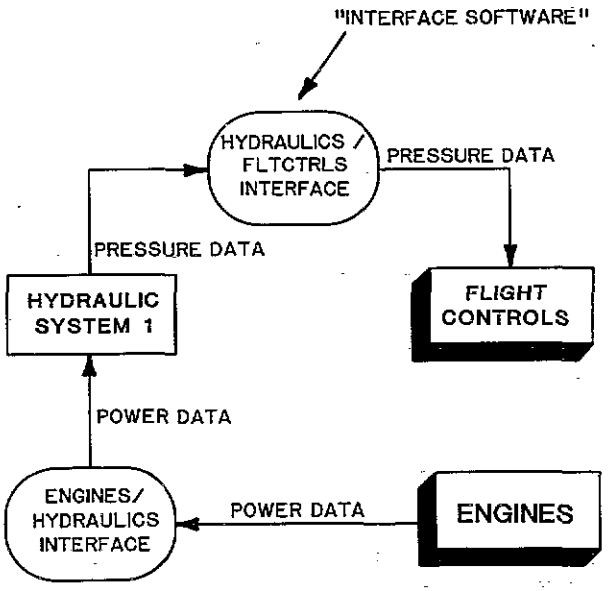

Figure 2. Software Implementation

The goal of the prototyping activity should be to develop a design approach that enforces software engineering concepts, while also working satisfactorily within the implementation constraints of the computer system to satisfy the real-time requirements of the device. Figure 3 illustrates the role prototyping should play in the design process. Through prototyping, the use of the design method may be tailored to yield a design that is easily transformed into code. If prototyping is not performed, the transformation from design to code may not be straightforward.

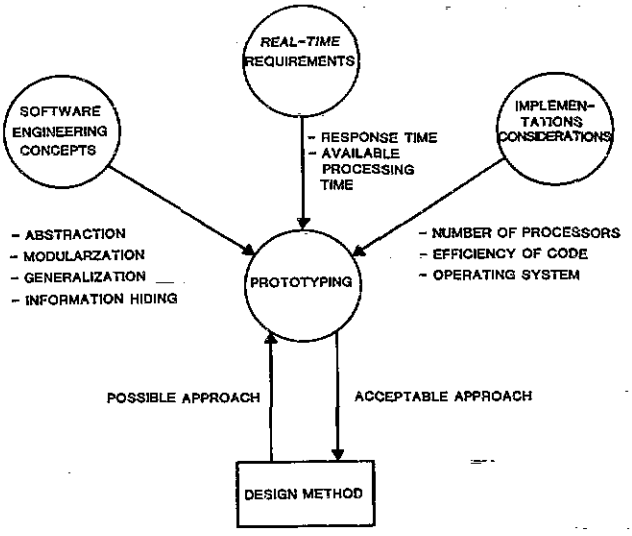

Figure 3. Prototyping During Design

The emphasis placed on software engineering during the ASVP proved to be very worthwhile. By far the most important benefit of the rigorous design process was a significant reduction in the time required for integration and test compared to the time required for these activities on most FORTRAN projects. Eleven weeks were originally scheduled for HSI based on the effort required to perform HSI on similar FORTRAN projects. Actual integration was accomplished in nine weeks by, on the average, three engineers on-site. The engineers had access to the simulator eight hours per day on weekdays, and twenty-four hours per day on weekends.

The early completion of HSI left time for the addition of two more simulator systems; a limited ground handling system, and the visual system. These systems were not originally scheduled to be redeveloped in Ada. Complete test and integration of the visual system was accomplished in three days, and integration of the ground handling system was accomplished in two days. Figure 4 shows the family tree of the software that was developed for the ASVP.

Compared to the FORTRAN software, the software developed on the ASVP was more maintainable and more robust. Problems were easily located due to the dedicated interface between the systems. In addition, the much dreaded "ripple effect" of software modifications was minimized due to the modular structure of the system.

COMPUTER SYSTEM SUPPORT ENVIRONMENT

During the redevelopment effort, the engineering team learned that the degree to which Ada programming support tools are coordinated and integrated in a development environment greatly affects the ease with which a software system is developed, tested, and maintained. An environment with a well integrated and fully documented tool set allows the engineer to be productive and efficient. Other factors that affect the usability of tools are completeness of documentation, ease of use, and speed of the tool. These three items determine the extent to which a tool will be used during the software life-cycle. Therefore, the entire set of tools supplied by a vendor may not be put to full use simply due to the quality and coordination of the tools.

An Ada Programming Support Environment (APSE) may contain a wide variety of tools to complement any aspect of the software development process. These tools may range from requirements analysis tools to automated test tools. Although the study of tools was not a main thrust of the ASVP, we were able to evaluate a standard environment made available to us by Gould under MPX. The Gould APSE consisted of the following tools:

- Editor. The Gould Programmable Editor was used throughout the development and integration of the software. This editor

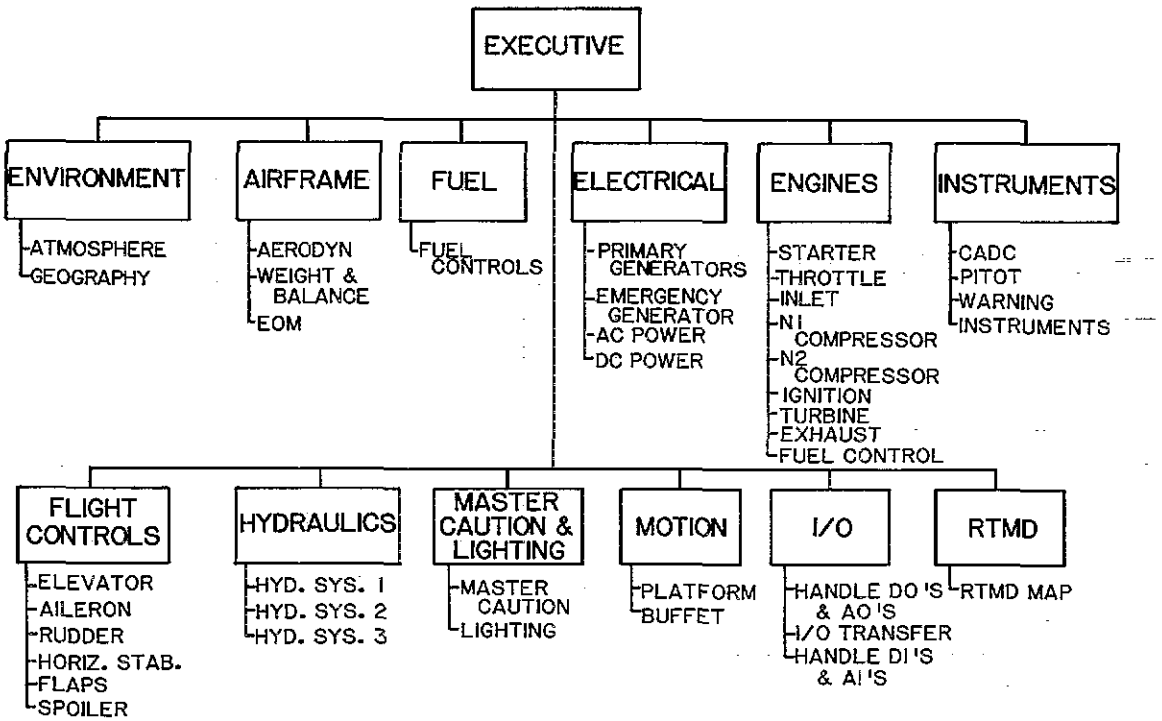

Figure 4. Software Family Tree

could be reprogrammed to fit the user's requirements. For example, a single key could be programmed to produce a frequently used package name (IO_MCL, ELE_IO, etc). The capability to customize the editor greatly enhanced its usability.

- Syntax Checker. The syntax checker was very useful during the early coding phases of the ASVP. It allowed the user to quickly check for syntax errors without leaving the editor. (The alternative was to exit the editor and invoke the compiler which would have taken considerably longer.)

- Library Management Tools. These tools allow the user to perform a wide range of operations on the Ada libraries. These tools enabled the user to look at several characteristics of an Ada library (size, components, dependencies, etc.) as well as create and delete libraries and their components. In addition, these tools allowed the user to move a compilation units from one library to another.

- Pretty Printer. This tool was only minimally useful on the ASVP. Had we been required to produce documentation and hold critical reviews of Ada code, it would have been beneficial to have all source files printed out in the same format. It is important for the pretty printer to allow the user to easily customize the format of the output as one format will not be satisfactory for every application.

- Compiler and Linker. The Aplex Compiler (Rev. 1.0) and Aplex Prelinker were used to process all of the Ada source code generated. We found that the speed of the compiler was adequate throughout the entire program. The linker, however, was not as efficient as we would have liked.

During the development of the software system it became evident that several additional tools would have been useful:

- Automatic Recompilation Tool. This tool should determine what units need to be recompiled and the order of the compilation. As a software system grows, the compilation dependencies between units do not always remain obvious. The modular architecture that results from using Ada complicates the compilation problem by increasing the number of compilation units within each system. A tool which handled this problem automatically would reduce the time an engineer would need to spend tracking dependencies. On ASVP, indirect command files were established to provide automatic recompilation. This system was workable, but an automatic recompilation tool would have been far better.

- Library Management Tools. A tool is needed in this area that allows the user to compress an Ada library after numerous recompilations have occurred. During software development, Ada libraries grew larger and larger after each recompilation of a unit into the library. The only way to recapture any space in the library was to delete the library, recreate it, and manually recompile all Ada units into the new library.

- Automated Configuration Management Tool. Configuration management of source files is not a new problem with Ada. Due to the many compilation units produced by Ada architectures, the number of source files that must be maintained increases. A tool which automatically keeps track of version and revision numbers, program structure trees, and past configurations would greatly ease the programmers job. Relying upon manual insertion of revision numbers does not ensure that accurate records will be kept regarding date, name, and reason for change.

- Source Line Debugger. Because of the modular structure that can be achieved with Ada, it is easy to test many small stand-alone units. Testing these units with a source level debugger is an easy way to verify math models and algorithms. Burtek ported much of the simulation code to a MicroVax computer in order to make use of the VAX debugger. (At the time there was no source line debugger available for use on the Gould system.)

- Real-Time Monitor. Despite the short integration time required by well-developed Ada code, there is still a need for a real-time monitor. This tool is an essential part of any sizeable real-time time software system. Burtek made extensive use of a real-time monitor developed in-house. This tool allowed for monitoring and changing of values during real-time execution.

In real-time applications, the operating system can have a significant impact on the design alternatives that are available to the designer. Ada systems are very different than most older software systems, and therefore make different demands on the operating system. To make optimum use of Ada in real-time, operating systems will need to be tailored to handle Ada tasking and the architectural complexity of Ada-based software systems.

MAINTAINABILITY

Maintainability is defined as the ease with which software can be changed or modified over the entire life-cycle of the system. Software must be

designed to be maintainable. Simply coding in Ada will not ensure that the end product will lend itself to easy maintenance.

Maintainability must be considered at the earliest stages of design and should be the main thrust throughout the design process. Qualities that will make the software maintainable should be defined early in the design process and communicated to the design team. Some of these qualities include: small "object-oriented" modules, close mapping of the modules to actual aircraft components, and dedicated interfaces between the various systems.

In addition, Ada features must be used in a straightforward fashion in order for software to be maintainable. General guidelines should be established early in the program regarding the use of Ada features to encourage consistency in the design. The haphazard use of Ada features will not produce a maintainable or efficient product.

Care must be taken in considering the tradeoffs involved in the selection of Ada constructs to ensure run time efficiency and a usable end product. Benchmarking should be performed to evaluate the the performance of various Ada constructs. This information is helpful in resolving possible tradeoffs between maintainability and efficiency. When a tradeoff exists, the more maintainable approach should be taken provided the run-time penalties are not excessive. If it becomes necessary to "speed-up" the code, it is fairly easy to make maintainable code fast. It is rarely easy to make efficient code maintainable.

It was found on the ASVP that time devoted to designing maintainable software reduced the time required for testing and integration. In addition, when some additional systems (the visual system and a limited ground handling model) were added following integration , the software was installed and tested in a very short period of time (three days for the visual, two days for the ground handling).

Special considerations should be given to compilation dependencies when designing Ada software. Compilation dependencies should be minimized as much as possible. To accomplish this the system designers must have a good understanding of the visibility rules in Ada.

Ada compilers need to provide pragmas such as INLINE and SUPPRESS to improve the execution speed of code that has been designed to be maintainable.

Control of types is an important consideration in code maintainability. It became obvious in the early stages of development on ASVP that the number and organization of types would need to be controlled. A suggested approach would be to have a high level type package containing the basic types required for the project, such as the number of engines on the aircraft, number of hydraulic systems, etc., and packages at the system level that contain system unique types.

The experience on the ASVP indicates that it is possible to produce code that is both maintainable and efficient with sufficient front-end planning and organization.

TRAINING

Training on the ASVP was accomplished primarily through in-house training courses conducted by a consultant. Prior to starting the ASVP the entire design team had completed an Ada programming course, however none of the team members had received any formal training in software engineering. Thus, at the outset of the project, software engineering training sessions were conducted several times a week. Through the first few months several lessons were learned about training in software engineering methods.

Early in the project it was found that teaching Ada syntax is merely the starting point in training an engineer to design Ada software. Software engineering concepts and design methods are much more difficult to grasp than the language syntax. Merely being a language expert does not qualify an individual to design Ada software.

It is very important not to be too abstract when teaching software engineering methods. The tendency in the early phases of the ASVP was to study software engineering methods only at very high abstract levels. Implementations of the abstract designs generated using methods such as OOD were rarely ever discussed. It is much easier for engineers to understand and accept a software engineering method whenever the implementation of the design is understood.

Training in the area of application is much more worthwhile than training from textbook examples. Actual use of the method in the engineers area of expertise will prove to the engineer that the method is worthwhile. It is often difficult to read simple textbook examples and understand how these methods can be used on larger more complex systems.

SUMMARY

The results of the ASVP indicate that Ada and software engineering can provide a feasible means for producing an efficient real-time application. In addition, other important benefits were realized. Software designs were significantly improved resulting in a more maintainable software package. More time spent producing a structured design was found to have the benefit of reducing the time required for code and integration. The time saved in the later phases of the program offset the additional time spent on design and produced an overall time savings. Integration and testing of the ASVP was accomplished in much less time than was required for integrating and testing of comparable FORTRAN systems.