SOFTWARE ARCHITECTURES FOR ADA-BASED FLIGHT SIMULATORS

Michael E. Caffey

FlightSafety International
Simulator Systems Division
Broken Arrow, Oklahoma

ABSTRACT

This paper presents information gained during the rapid proto-typing effort for the Rotor Wing Blade Element Simulator Program being conducted by the Simulator Systems Division of FlightSafety International. In this project, a full-flight simulator will be developed for a Bell 212/412 helicopter. Software for the device will be developed in Ada and hosted on a dual-processor Harris Night Hawk Computer System. The software is computationally intensive and includes a 200 Hz finite-element rotor-blade simulation. The goal of the project is to investigate the use of Ada on a production simulator with stringent real-time processing requirements.

This paper will address many of the architectural issues considered in the rapid prototyping phase of the development effort.The discussions will focus on the application of Object Oriented Design (OOD) Techniques in the design of software for critical real-time systems. The discussions will highlight some of the advantages afforded by Object Oriented Architectures as well as some of the key problems encountered when using Object Oriented Design in large real-time applications. In addition, the discussions will address many of the operating system capabilities that will be required in order to make optimum use of Ada and Object Oriented Programming in future simulator systems.

INTRODUCTION

This paper presents information gained through a research and development project being conducted by the Simulator Systems Division of FlightSafety International (FSI). This project, called the Rotor Wing Blade Element Simulator Program, is being performed in order to gain experience in the use of Ada and Software Engineering Techniques in the development of simulator software. The project involves the development of a single Phase-II flight simulator for the Bell 212 and the Bell 412 helicopters. All of the host computer software for this device will be developed in Ada on a dual-processor Harris Night Hawk Computer System. Upon completion the simulator will be installed at Flight Safety's Ft. Worth Training Center where it will be used for initial and recurrent pilot/copilot training.

FSI is conducting the Rotor Wing Blade Element Simulator Program in order to develop expertise in several areas. Though the main objective of the program is to gain experience in the use of Ada, FSI also hopes to develop expertise in : 1) the use of an Ada-based Program Design Language (PDL), 2) development of software to military standards, 3) the application of software engineering techniques, and 4) the use of a Unix-based software development environment. For these reasons FSI has elected to develop software for this device in accordance with DOD-STD-2167A. In addition, an Ada-based PDL will be used during the detailed design phase of the project and DOD-STD documentation will be generated using a document generator that is integrated with the PDL. Also an object-oriented design method is being used for the software design. Software development is being performed on a Harris computer system using a Unix-based operating system.

This paper presents a number of the technical issues considered while developing the architectural model for the simulator software. The discussions will highlight some of the key technical issues that are unique to real-time systems and will describe how these issues were addressed in the design of the Ada software. In addition, the discussions will illustrate the way Object Oriented Design (OOD) Techniques are used in the system. Some of the key characteristics of object oriented designs will be discussed and some of the key differences in an object oriented design and the designs used for FORTRAN simulators will be highlighted. Also some of the operating system features/characteristics that affected the design will be addressed.

The software design presented here was validated through a rapid-prototyping effort and is being used as the basis for the design of the Bell 212/412 simulator software. The results of the prototyping effort will be discussed and a few important points about optimizing Ada software will be highlighted.

Ada AND OBJECT ORIENTED DESIGN

The transition to the Ada programming language is causing simulator manufacturers to re-think the way simulator software should be designed. The flexibility of the Ada language and the software engineering concepts supported by Ada mean that the optimum design of Ada software is sometimes very different than the optimum design of a FORTRAN system. It is generally agreed that Object Oriented Design Techniques are the most suitable approaches for developing Ada software. Object Oriented Design (OOD) encourages good software design practices and in most cases a straightforward transformation is possible from an object oriented design to an implementation in Ada. However there are some deficiencies in OOD.

OOD provides no support for the front-end requirements analysis that must be performed prior to software design. Developers of Ada software must supplement OOD with some suitable method of requirements analysis such as Structured Analysis. In addition, OOD does not take into consideration certain key implementation issues that must be addressed during high-level design. Thus object oriented design approaches are fairly useful in the middle phases of design, but at higher levels of design other techniques may be required.

In developing the software for the Bell 212/412 flight simulator, FSI elected to use an object oriented design for the simulator software. Since this type of approach is not possible using FORTRAN considerable effort was devoted to determining how to use an object oriented design for simulator software. In addition, a number of other real-time issues needed to be resolved reguarding the use of Ada in real-time. The sections that follow describe some of the issues that were considered during the architectural design of the software and describe the design approach that has been adopted. The discussions will highlight where object oriented techniques have been used in the software as well as the deviation from OOD that was required at high levels of design. It is intended that this discussion will illustrate the need for software developers to be trained in the use of design methods as well as language syntax.

DESIGN ISSUES

The design approach for the Bell 212/412 simulator software has been driven by several key issues. One of the most significant of these issues is the stringent processing requirements of the rotor-blade dynamics software. The rotor-blade dynamics software for the 212/412 performs a finite-element analysis of the forces acting on each blade of the rotor. In this model, each blade is divided into several segments and the force components acting on each segment of each blade are calculated independently and summed to determine the total force acting on the rotor. Due to the speed of rototation of the blade this model must execute at 200 Hz. The software is computationally intensive and when executed at 200 Hz. places a significant burden on the processor. Thus early in the design phase it was deemed important that a design approach be used that would allow the rotor blade software to be optimized effectively. FSI hoped to be able to optimize the rotor blade software enough to allow all of the high frequency (200 Hz) software to run in a single processor leaving the other processor free for the remaining flight software and the other subsystems.

Another factor that affected the design of the Bell 212/412 software was the need to split the simulation accross two CPU's. An analysis of previous helicopter simulations developed by FlightSafety indicated that at least two processors would be required to host all of the real-time software. Though the use of multiple processors is not unusual in simulation, consideration must be given to the best way to design Ada software to allow tasks to be moved between CPUs. OOD does not consider such issues in any way.

The availability of a frequency-based scheduler (FBS) on the Night Hawk Computer System was another key consideration in design. The frequency-based scheduler allows multiple operating system tasks to be scheduled at fixed frequencies elliminating the need for a classic executive. Early in the design of the prototype software, it was determined that the frequency-based scheduler could add a good deal of flexibility to the software. However the software design needed to be thoroughly evaluated to ensure that the software was not so tied to the frequency-based scheduler that migrating to other hardware platforms would be difficult. Thus considerable time was devoted to determining the best way to use the FBS with the Ada software.

One of the problems encountered when developing real-time software is debugging the software in real-time. Standard symbolic debuggers are not capable of operating in real-time.

These debuggers are useful during the early stages of software testing; however, a real-time monitor is essential for complete testing of real-time systems. The real-time monitor allows the values of variables to be displayed while the software is running in real-time. The user also has the option of assigning values to these variables interactively.

Typically the real-time monitors used in FORTRAN simulations have depended heavily on the use of FORTRAN common blocks. The analogous construct in the Ada language is the specification part of the Ada package. However, due to the compilation dependencies that exist between an Ada package and the programs that depend on the package, the use of large Ada packages as common areas is not ideal in Ada. In addition, the use of OOD encourages engineers to restrict the visibility of data as much as possible. Thus an important consideration in the software design was structuring the software so that a real-time monitor could access sufficient data for testing without the use of massive common blocks.

SOFTWARE ARCHITECTURE

Background

In order to resolve as many design issues as possible prior to the main design team beginnig work, a rapid prototyping effort was conducted. During this phase of the project a design approach was developed for the system and a portion of the simulator software was developed to validate the approach and serve as example code that could be used by other engineers. Since the Night Hawk was not yet in full-scale production at the outset of the program, software was developed on a Harris HCX-9 and ported to a Night Hawk for benchmarking using the frequency-based scheduler. The benchmark results were used to determine the final configuration (number of processors, size of memory,...etc.) of the Night Hawk computer that was to be used for the simulator.

Several different design approaches were considered during the rapid-prototyping effort. These approaches varied in the degree to which they attempted to enforce modularity and object-oriented design techniques. Only the approach deemed most desireable is presented here.

Design

The software for the Bell 212/412 simulator is divided into several subsystems. The division used is fairly typical of the way simulator software is usually organized. The subsystems include engines, navigation, fuel, electrics, rotor dynamics,...etc. Though this division of systems is fairly typical the implementation of these systems deviates from the normal FORTRAN implementation in several ways. Each of these subsystems is implemented as a separate operating system task (or process in Unix terminology) that is scheduled at a fixed frequency by the Harris frequency-based scheduler. This is a deviation from the most common approach used in simulation where there is a single real-time task resident on each processsor that is sometimes accompanied by several background tasks.

Communication between each of the simulation tasks is accomplished through a shared memory region. The shared memory region is comprised of several Ada packages referred to as "state packages". There is one state package for each operating system task. The task that owns the state package broadcasts data that is required by other tasks to the state package. The other tasks import (WITH) the state package and read the required data. Each state package contains only information that is required by other subsystems. A diagram of the software design approach is shown in Figure 1. In this diagram each subsystem task is denoted by a circle, and each subsystem state package is denoted by a rectangle. The arrows indicate the direction of data flow between the subsystems and the state packages.

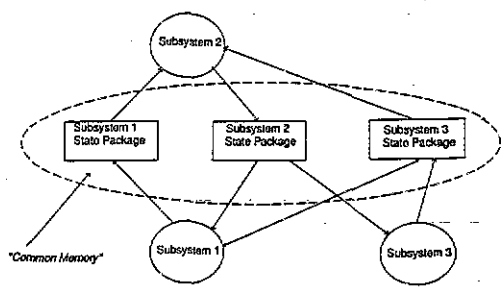

Figure 1. Subsystem Interprocess Communication

Use of the multiple task approach offers several advantages over the typical single task per processor approach to simulation. Among these advantages are :

Increased modularity - The subsystems are completely decoupled from each other.

Flexible configuration management - Use of multiple tasks easily allows multiple versions of a subsystem to be maintained. It also makes it easier to run test versions of one or more subsystems against the configured versions of the other subsystems.

Better fault tolerance - When a single real-time task is used for all of the software on a CPU, a fatal error in one program kills all of the real-time software operating on that processor. By using multiple tasks, only a single subsystem is aborted when a fatal error occurs. This is extremely helpful in the early stages of software testing.

Increased protection of common data - The only data that is common to all of the programs in the system is the data contained in the state packages. Since each of these packages is owned by one subsystem permissions can be associated with each package that allow only the owning task to write to the package. This prevents accidental corruption of shared data by the subsystems that are only supposed to read the data.

The task that is used to implement each subsystem is comprised of several parts : a main program, a subsystem package, and numerous object packages. Figure 2 illustrates the architecture of the subsystems. In this figure, a single rectangle denotes an Ada package that consists of a specification only. A shadowed rectangle denotes a package that is comprised of both a specification and a body. An oval denotes the main program of the task. The arrows in this figure denote a dependency (ie. an Ada "WITH" clause), **NOT** the direction of data flow.

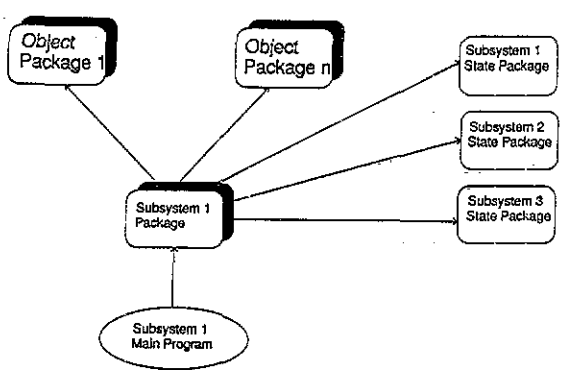

Figure 2. Subsystem Architecture

The main program for each subsystem performs all of the interface to the frequency-based scheduler. This program makes the system calls required to interface to the FBS and contains logic to suspend the execution of all or part of the subsystem when certain freeze conditions are requested.

The actual code used to implement each subsystem is contained in the subsystem package that is imported (WITHed) by the main program. This package contains the declarations of variables to be used in the models for that subsystem and a set of subprograms that implement the models. These subprograms are called by the main program for each subsystem each time that the main program is triggered by the FBS. The subprograms for each subsystem can read data computed by other subsystems from the subsystem state packages. In addition, the subprograms can broadcast data to the subsystem state package for their subsystem. Note that by encapsulating the code used to implement the subsystem models in a package, the machine dependent code contained in the main program (the interface to the FBS) is separated from the code that is not machine dependent. Thus all of the code that implements the simulator models for the subsystems can be easily ported to another hardware platform that uses a different method of scheduling programs. Separating machine dependent code from code that is not machine dependent has been a key consideration in the software design.

To further increase the modularity of the subsystems, many of the components of the software are implemented as Ada private types. The packages used to implement these private types are referred to as "object packages" since each of these packages implements a single private type which may be viewed as an object. The object packages are imported by the main subsystem package and are used extensively in the software to promote modularity and re-use of code in other subsystems and on future projects. To a large extent, the programs contained in the subsystem packages simply put together programs contained in the object packages.

This use of private types simplifies design of each subsystem a great deal. Much of the software used on simulators can be divided into objects fairly easily. By implementing each of these objects as private types the engineer can focus his attention on developing a number of small simple programs instead of a few monolithic programs. In addition, the scope of changes to any part of the software is well defined when using private types, thus maintenance of the software is

simplified. Also through using private types, closely related data is encapsulated in a single data structure improving the understandability of the code. In addition, high-level design/decomposition of the software is simplified since each subsystem can be viewed as a collection of simple well defined "black boxes". The well defined interfaces to these "black boxes" simplify the development of math models and code for the components.

In order to permit a real-time monitor to be used for testing of the 212/412 software careful consideration was given to the location of all variables declared in the software. The only variables which are readily accessible to the real-time monitor are those for which memory is statically allocated. This consists primarily of variables declared in package specifications and package bodies. Variables declared within subprograms exist only for the life of the subprogram. In order to protect data from inadvertent corruption and still allow access to key data by the real-time monitor data that must be viewed in real-time is declared in the bodies of the subsystem packages. This prevents any one subsystem from accidently corrupting data that belongs to another subsystem while still allowing the data to be viewed in real-time.

Comparison with Classical OOD

The use of shared data and multiple operating system tasks represents a deviation from classical OOD where Ada tasking is generally used for concurrent processes. Classical OOD does not address operating system constraints in any way. This is a deficiency in most design methodologies that is most clearly realized when these methodologies are used to design a large "real-world" system. There are certain critical implementation issues that must be addressed during the early phases of design, yet most methodologies fail miserably in addressing these issues. However, at lower levels of design methodologies can be useful and OOD was used at lower levels in the design of the prototype software.

For example, at the subsystem level, the Bell 212/412 software reflects a relatively classical approach to OOD. The subsystem packages are objects that are being implemented as state machines. The state machine in turn contains a number of objects that are implemented as private types. It is important to note the differnces in this design and the design normally used in FORTRAN simulators.

In this design each subsystem has its own state. The state of the subsystem is determined by the state of the variables declared in the body of the subsystem packages. These variables can only be altered by calling one of the programs declared in the subsystem package. Other subsystems have no access to this data thus the data cannot be accidently corrupted by another subsystem.

The most common FORTRAN system that is analogous to this design is acheived through the use of a local common block for a set of programs. This is illustrated in Figure 3. This is different from the Ada implementation because the subsystem (the collection of programs) does not have its own state. The programs must store data in the common block and retrieve it in order to function. There is not a single compilation unit that encompasses the entire subsystem as there is in the Ada version. The common data used in the FORTRAN version is volatile. Other programs can access the common data and accidently (or maliciously) corrupt it. The accidental corruption of data is one of the main problems in most large FORTRAN systems. Ada provides the potential to prevent many of these kinds of problems.

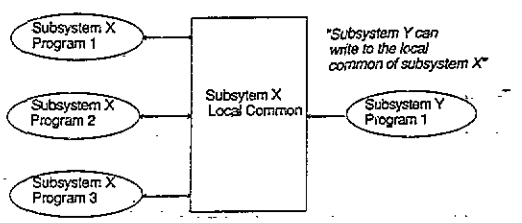

Figure 3. FORTRAN Implementation of a Subsystem

Within each subsystem many of the data structures used are implemented as Ada private types. These data structures can only be manipulated using the programs contained in the packages where the private type is declared (ie. only the programs in the object packages can manipulate the data structures). This prevents programs from manipulating the data structure in an inapropriate way. This use of private types maps directly to object oriented design techniques in which a system is divided into a collection of objects (the data structures) and the operations (the functions and procedures) that must act on the objects. FORTRAN does not provide the rich set of data structures that are possible with Ada, and it provides no way of limiting the way any of its data structures are manipulated. Thus implementing an object oriented design in FORTRAN is very difficult if not impossible.

PROTOTYPE

In order to validate the design approach for the software the high frequency portion of the rotor blade simulation was developed using this design approach. Due to the high frequency of this model it was determined that this would be the most time critical portion of the simulation. Ultimately the intent is to optimize the high-frequency rotor software enough to run all of it in a single CPU leaving the second CPU free to host the lower frequency flight modules and the other simulator subsystems.

The prototype software was benchmarked on a Harris Night Hawk with several combinations of options used to optimize the execution speed of the system. In the initial tests, no options were used to optimize the code. Then various compiler options were used to optimize the speed of the software. Using an Object Oriented architecture tends to produce code that performs a large number of subprogram calls and performs a great deal of parameter passing. For this reason, pragma INLINE was used in optimizing the prototype software. Though this option is now supported by most compilers, care should be exercised to ensure that subprogram calls are not used excessively if the compiler to be used does not support INLINE. For this reason, object-oriented architectures may not be appropriate for systems inwhich the compiler does not support INLINE. In addition, compiler writers need to work to optimize subprogram calls and parameter passing as much as possible. These are crucial to the development of systems using object-oriented techniques.

Another option that was used in optimizing the prototype software was a compiler option that specified that the bodies of generic units were not to be shared. Some compilers attempt to share the bodies of instantiated generic units. The sharing of generic bodies conserves memory but it also introduces a run-time overhead each time that the generic unit is executed. This type of overhead is not acceptable in very time-critical situations. In addition, pragma SUPRESS was used to reduce the ammount of run-time checking that was being performed.

During the benchmarking it was found that the use of these features along with the use of an optimizing compiler made a considerable difference in the execution time of the benchmark. In total, the time required for executing the prototype model was reduced almost fifty percent from the initial run, which used no compiler optimization and no Ada pragmas for optimization, to the final run which used the highest level of optimization available on the Harris compiler and used Ada pragmas extensively to in-line subprogram calls and suppress run-time checks.. The initial (unoptimized) version of the prototype was far too slow for our purposes. However, after optimizing, the execution time of the prototype was at a very acceptable level.

SUMMARY/CONCLUSIONS

Through the rapid prototyping effort for the Rotor Wing Blade Element Simulator Program FSI has developed a design model that can be used for simulator software written in Ada. This model differs from the design normally used in FORTRAN simulators by its use of multiple real-time tasks and its use of object oriented design techniques. These differences improve the fault tolerance of the software, provide greater flexibilty in managing and maintaining multiple configurations of portions of the software, and improve the level of data protection that is enforced in the software. In addition, the approach promotes a highly modular software design and provides good support for reusing portions of the software.

A major lesson learned in the prototyping effort was that a clear understanding of design methods and good software design practices is essential to take full advantage of the capabilities of the Ada language.

ABOUT THE AUTHOR

Mr. Michael Caffey is a Principal Engineer at FlightSafety International specializing in operating systems and the design of Ada software. Mr. Caffey is currently working as System Architect on the Blade Element Simulator Program performing research and prototyping on software architectures that may be used in real-time Ada systems. Through other research and development efforts Mr. Caffey has gained experience in the use of software engineering techniques and Ada in real-time simulator systems. Mr. Caffey holds a Bachelor of Science Degree in Electrical Engineering from the University of Tulsa.