THE CHALLENGES OF DEVELOPING A REAL-TIME ENVIRONMENT IN ADA

Walter E. Zink, Sr., and Jill M. Neebe
CAE-Link Corporation
Binghamton, NY

ABSTRACT

With fewer and fewer exceptions, the Department of Defense is requiring Ada to be the sole programming language for all new software-related projects. In addition, these new projects are expected to achieve higher levels of maintainability from a software perspective. Experience shows that these seemingly unobtrusive requirements manifest themselves in a very large variety of unforeseen challenges and implicit requirements. This paper overviews an Ada real-time flight simulation environment based on an implementation for the B-2 Aircrew Training Device (ATD) and the challenges encountered in going from concept to product. Three areas of challenge are involved in building this environment. The first concerns the control of software units distributed across processors and groups of processors. Another area of concern is providing input/output services to all units in Ada, which even the operating system does not readily support. The third area covers selected obstacles encountered in developing a pure Ada implementation of a system to support unit interfaces. The resultant real-time environment represents an effective blend of Ada and traditional techniques.

INTRODUCTION

The Simulation Environment discussed in this paper is a Real-Time Simulation Environment (RTSE) in which the application/subsystem software executes. It is divided into three principal areas: control, software interfaces, and input/output (I/O) (Figure 1). In this context, control consists of implementing the current user's instructions and orchestrating the execution of other simulation software.

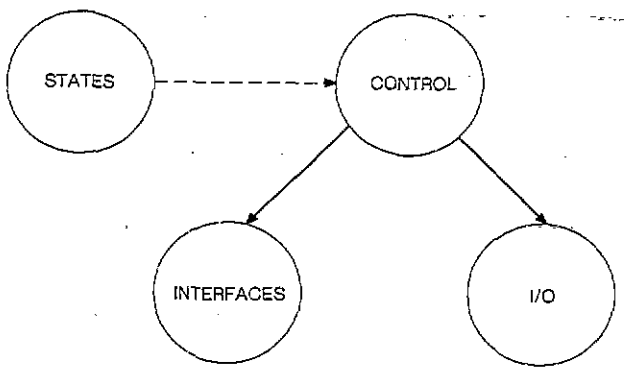

Figure 1 Real-Time Simulation Environment

Since Ada is the language of the RTSE and DOD-STD-2167A is a currently used standard for Ada programs, DOD-STD-2167A phases will be used to chronologically orient the requirements and implementations. Figure 2 outlines these for each product area. The discussion covers System Requirements Analysis to Coding and Computer Software Unit Testing.

While Ada has attractive capabilities not available in other languages, compiler capabilities can dictate IF and HOW various features can be practically implemented. The Ada Joint Program Office, a part of the United States government, sets the criteria and certifies Ada compilers. In the 1985/1986 time frame, Ada compilers were still in their infancy. The Ada Programming Language Military Standard, ANSI/MIL-STD-1815A, was approved in February 1983. Only three compilers had received validation by December 1983, and in 1985 only 14 validated compilers were available on the market.[1] The validation criteria did not require all of the features of Ada needed by a RTSE.

The majority of previous simulation software produced by the company consisted primarily of FORTRAN code; the B-2 ATD was the first "Ada simulator" for the company. Previous design experience was primarily functional decomposition. With Ada we use Object Oriented Design techniques. New procedures needed to be developed and written from an Ada perspective.

Naturally, Ada concepts played an important part in RTSE implementation decisions. As stated in MIL-STD-1815A, "the development of programs is becoming ever more decentralized and distributed. Consequently, the ability to assemble a program from independently produced software components has been a central idea in this [Ada's] design. The concepts of packages, ... and of generic units are directly related to the idea."[2]

2167A PHASES →	SYSTEM REQUIREMENT ANALYSIS/DESIGN	SOFTWARE REQUIREMENTS ANALYSIS	PRELIMINARY DESIGN	DETAILED DESIGN	CODING & COMPUTER SOFTWARE UNIT TESTING
CONTROL	PORTABILITY			MACHINE DEPENDENCIES ISOLATED	
	REUSABILITY		USE GENERICS AND PSEUDO_GENERICS		
	MAINTAINABILITY		USE GENERICS AND PSEUDO_GENERICS		
		PROVIDE A CYCLIC ENVIRONMENT TO RUN USER SOFTWARE	EXECUTIVE DESIGN		
		PROVIDE CENTRAL CONTROL ACROSS PROCESSING ELEMENTS	MASTER/CLUSTER/ SEQUENCER CONCEPT		
		PROVIDE AUTOMATIC LOADING LEVELING	USE OF MULTIPLE-RATE O/S PROCESSES		
		PROVIDE SINGLE-POINT CONTROL		BOOTSTRAP	
		MINIMIZE OVERHEAD ASSOCIATED WITH THE REAL-TIME ENVIRONMENT		O/S PROCESS CONTROL IN CLOCK HANDLER	
INTERFACES	REUSABILITY				
	MAINTAINABILITY	MINIMIZE SYSTEM-WIDE RECOMPILATION OF APPLICATION SOFTWARE		WITH AT BODY LEVEL OF CONNECTION MANAGERS	
		PROVIDE AN AUTOMATIC METHOD TO MANAGE THE INTERFACES, GENERATE THE INTERFACING SOFTWARE AND DOCUMENT THE INTERFACES	SHARED MEMORY MANAGEMENT - INTERFACE MANAGEMENT DATA BASE	DATA BASE ACCESS SOFTWARE INTERFACE GENERATION	
	INFORMATION HIDING				
	DATA ABSTRACTION				
			ENSURE DATA CONSISTENCY FOR INTERFACES	IMPORT/EXPORT CONNECTION MANAGER	
			ENSURE DATA INTEGRITY FOR ALL INTERFACES	DOUBLE BUFFERING OF GLOBAL MEMORY	
		MINIMIZE THE OVERHEAD ASSOCIATED WITH INTER-SOFTWARE COMMUNICATION			
		PROVIDE THE CAPABILITY TO SAVE THE SIMULATOR ENVIRONMENT AND RESTORE THE ENVIRONMENT FOR EITHER A POWER FAILURE OR AN INSTRUCTOR'S REQUEST		SNAP/RESET PRODUCT	
		PROVIDE INTERFACE BETWEEN SOFTWARE COMPONENTS WITHIN THE SAME O/S PROCESS AND BETWEEN O/S PROCESSES EXISTING ON CLUSTER OF PROCESSORS		GLOBAL MEMORY IMPORT/EXPORT CONNECTION MANAGER	

Figure 2 Requirements and Implementations

2167A PHASES →	SYSTEM REQUIREMENT ANALYSIS/DESIGN	SOFTWARE REQUIREMENTS ANALYSIS	PRELIMINARY DESIGN	DETAILED DESIGN	CODING & COMPUTER SOFTWARE UNIT TESTING
I/O	PORTABILITY		USE OF ADA CONSTRUCTS OF TEXT_IO AND DIRECT_IO	MESSAGE_REQUEST OPERATIONS, DIRECT_IO OPERATIONS	
	REUSABILITY		USE GENERICS	REQUEST_DIRECT_IO, REQUEST_CONTIGUOUS_IO	
	MAINTAINABILITY		USE GENERICS; USE ENGLISH-LIKE NAMING PRACTICE	REQUEST_DIRECT_IO, REQUEST_CONTIGUOUS_IO	
	DATA ABSTRACTION		USE PACKAGES	MOST I/O SOFTWARE	
	OUTPUT MESSAGES TO ERROR LOGS, AUDIT LOGS, AND TERMINAL IN SPECIFIC FORMAT			MESSAGE_REQUEST	
		NO-WAIT I/O		INTERFACE STRUCTURE, ALL USE MODULE REQUEST SOFTWARE, ALL I/O PROCESSOR SOFTWARE	
		I/O ACROSS PROCESSORS		INTERFACE STRUCTURE	
		I/O ACROSS CLUSTERS		INTERFACE STRUCTURE	
		I/O FOR OBJECTS GREATER THAN 64K		REQUEST_CONTIGUOUS_IO, CONTIGUOUS_IO OPERATIONS	
		DISK MARKING		REQUEST_DISK_MARKING	

Figure 2 Requirements and Implementations (Cont'd)

We found the packaging concept advantageous from a design perspective. The concepts of limited visibility and information hiding supported our requirement for data abstraction, as did the ability to logically group types and objects with applicable functions and procedures. Packaging provided us with a means to partition the program software. The ability to overload functions and rename adds valuable flexibility.

The more reusable a module of code, the lower the actual cost per line of executable code. Generics are one method of avoiding repetitious code. This reduces software testing and configuration management complexity. Since Ada does not support passing of packages as formal generic parameters,we used package renaming and software built from a common template.

Ada provided tasking; our tasking experiences are discussed in relation to the control and I/O sections.

SYSTEM LEVEL DESCRIPTION

States Overview

A state design is the basis of the structure of the Real Time Simulation Environment (RTSE) software. A state is defined as follows:

> A state is a collection of rules within which a functionality can occur. The transition to another state is predicated upon successfully satisfying the state transition rules.

One implication of this definition is that the software executing in a particular state should be tailored to support only that state. This concept is in contrast with earlier implementations wherein the user software supported all states and had to interrogate multiple flags in order to determine appropriate functionality. In the state design used, the high level software functionality is determined off-line rather than during execution.

Another implication is that the definition of a command in one state does not have to be the same in a different state.This implication is observable in the functionality of the Control section. When the Configuration_State is Integrated, the definition of the command "Normal Load" is to load all clusters; however, in the Stand_Alone Configuration_State, the definition is to load only the cluster that has received the command.

The RTSE can be thought of in terms of three levels of states. States are supported directly by the structure of the Control section. The three state areas and their compositions are depicted below.

States	Super States	Configuration States	Element States
State_ Compo- sition	Pre-Synchronous (Boot Up)	Integrated	Initialization Freeze Real Time
	Synchronous	Stand_Alone	Reset
	Post-Synchronous (Shut Down)		Test

The aggregate of pre-synchronous activities is referred to as Boot Up. Post-synchronous activities are referred to as Shut Down. During Boot Up, the entire simulator is powered up, software loaded and started. During Shut Down, the entire system is stopped, cleared, and powered down. The Synchronous Super State is where the majority of the user software is cycled. It is in this Super State that the Configuration_States and Element_States have the most influence on the operation of user software.

The two Configuration_States are Integrated and Stand_Alone. Integrated means all clusters are synchronized and communicating among themselves. This refers not only to user level software, but also to RTSE level software (e.g., Control and I/O). In the Stand_Alone state a cluster functions in isolation from the other clusters.

Element_States primarily affect the execution of the user level software only and do not possess the systemwide influence of the Configuration_States. The Element_States are implemented by the Control Section via Control Records. Since each of the two Configuration_States supports the five Element_ States, there are ten Control Records.

Hardware Overview

The hardware configuration for the simulator can be partitioned into the following basic building components: the Basic Simulator Unit (BSU), the Instructor's Station, a visual system, and five multiple target system computer clusters. Since most of the real-time software executes on these computer clusters, a brief description of that hardware follows (Figure 3).

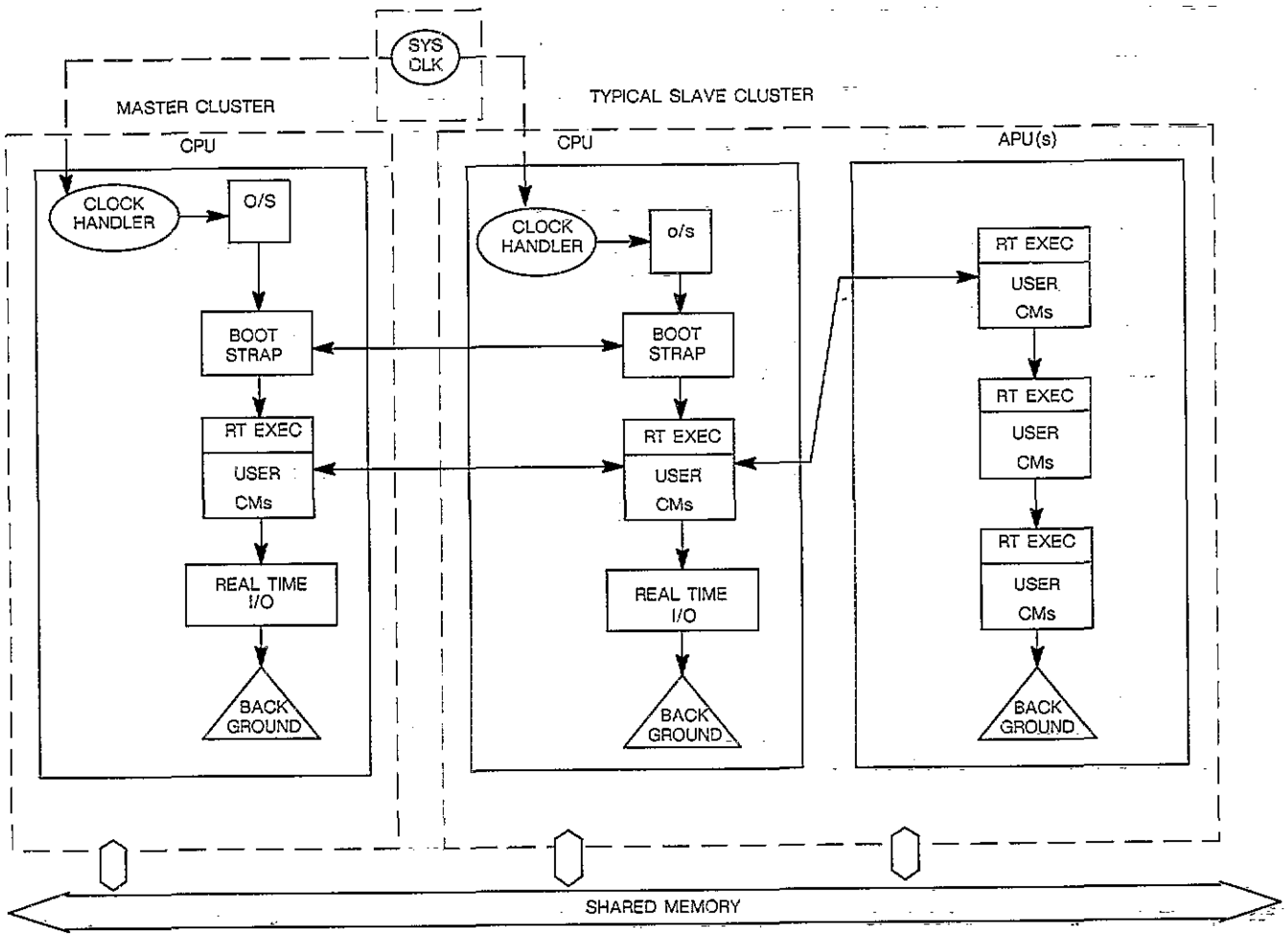

Figure 3 Hardware/Software Architecture Overview

Each cluster contains a Central Processing Unit (CPU), zero to three Auxiliary Processing Units (APU), and 32 megabytes of memory. Intra-cluster (i.e., between a CPU and one or more APUs) communication is performed through a part of memory which has local (to the cluster) visibility only. Inter-cluster (i.e., among multiple clusters) communication is through a special part of the memory that is visible to more than one cluster. In addition to data communication, all clusters are synchronized at the major cycle boundary. The System Clock (Figure 3) issues a hardware interrupt to each cluster once every major cycle, (refer to Figure 4). At this point all clusters will begin execution of the first frame of that cycle together.

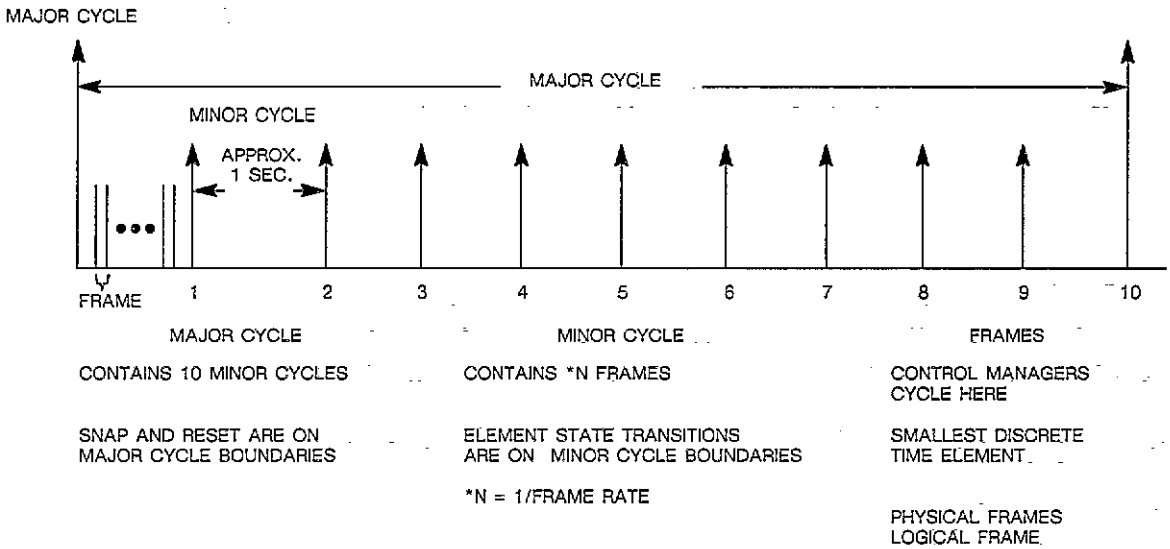

Figure 4 Frames/Minor Cycles/Major Cycles

Software Overview

The software for real-time execution is divided into three levels: Operating System level, RTSE level, and User (i.e., application) level (Figure 5).

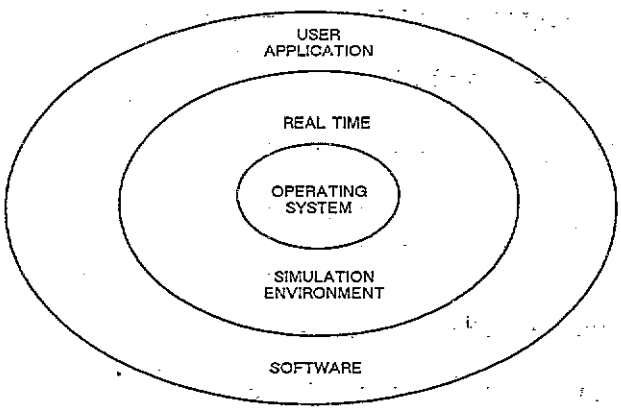

Figure 5 Software Hierarchy

The operating system level software consists of all vendor-supplied O/S services and any application-unique services such as the clock handler, which provides the connection between the hardware framing interrupts and the Control section. The principal activity of the O/S is to handle the scheduling of os_processes. This activity includes scheduling of os_processes on the same processor with different priorities as well as scheduling os_processes on different processors. The os_process with the highest priority is scheduled to run first by the O/S. When that process gives up control, the O/S will schedule the os_process with the next highest priority, and so forth. The RTSE uses this method to achieve auto-load balancing and frame-bound execution at the same time. The other services that the O/S provides are low-level interface software to hardware devices such as terminals and disk drivers.

The RTSE software is the next level of software above the operating system level. This software encapsulates the users' application software, isolates it from hardware constraints, and provides the structure and form necessary to implement the architecture. This level of software consists of the following software products which fall into three groups:

Groups	Control	Interfaces	Input/Output
Products	Executive Bootstrap	Shared Memory Management	Real-Time I/O

In addition, there is support software that is used during off-line development for the Executive, Real-Time I/O, and Shared Memory Management.

The Executive is the central component of the RTSE level of software. Although most subsystems (user software components) within the simulator are assumed to be autonomous, most of the RTSE level software is related to the Executive. The Executive schedules user software as a function of the cluster,

processor, state, and frame. User software is also referred to as Control Managers (CMs). Simulator states define the current functionality. Framing provides discrete intervals in which part of an activity must complete (e.g., integrating distance as a function of airspeed). The state in which the executive is running determines the personality of the entire simulator, even though other software (e.g., Bootstrap, Instructional Station user software) can alter the state of the executive.

The executive comprises three levels of control: master, cluster, and sequencer. Only one Master Executive exists. It is the highest level of control and manages all simulator common activities, such as fielding state change requests while in the integrated state. The Cluster Executives (one per cluster) directly or indirectly schedule all synchronous software within a cluster, provide physical and logical framing for the user software, and provide local management of states. The sequencer executive is the lowest level of executives and does the actual scheduling of the user software.

Bootstrap loads all simulator-related software and provides some control functions/utilities such as memory clear. The functionality of Bootstrap is a direct consequence of the configuration state of the executive.

Real-Time I/O provides the user software with an Ada-like interface for requesting disk and terminal services, and provides for error/audit logging. The functionality of Real-Time I/O is likewise dependent on the configuration state of the executive.

Shared Memory Management consists of two broad functions: a real-time operation and a database management function. The real-time function is to provide inter-user software communication via import/export connection managers and to provide Snap and Reset functionality.

The Real-Time library is a set of standard mathematical functions for integer, float, and long float operations that the user software can reference.

The user software is the last level of software in the simulator. This level consists of simulation-specific software.This software is broken down into subsystems and each of those is further divided into one or more control managers. Each subsystem is encapsulated with an import connection manager (IMP) and an export connection manager (EXP) (Figure 6). These connection managers are a part of the Shared Memory Management software.

CHALLENGES ENCOUNTERED

Control

Requirements and Challenges – The requirements used to design software can be organized into two principal categories: those of a product-specific nature and those of a software engineering nature. Often the requirements from one category will conflict with the optimum method of implementation of a requirement from the other category.

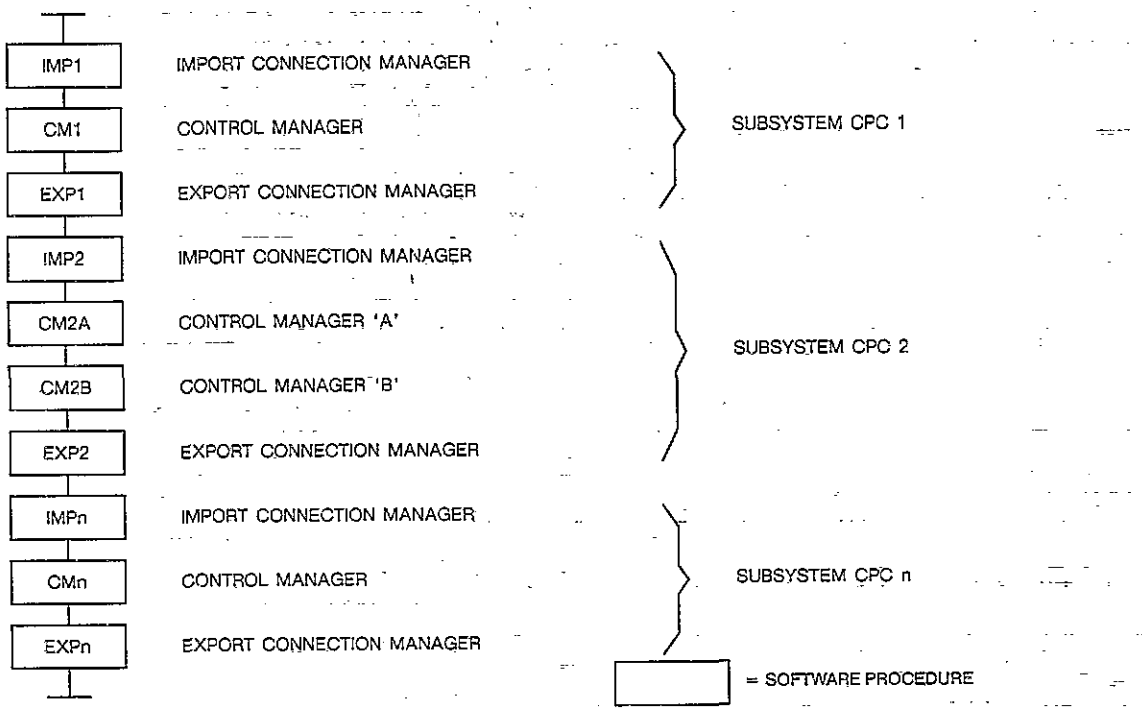

Figure 6 Typical Subsystem Execution Sequence

The product-specific requirements used to define the control portion of the real-time environment are fairly similar to those of previous simulators. They are as follows:

a. Provide a cyclic environment in which to run user software.

b. Provide central control across processing elements.

c. Minimize the overhead associated with the real-time simulation environment.

d. Provide automatic load leveling.

e. Provide single point control.

f. Provide a state-driven environment to envelope the user software.

The "Software Engineering" requirements are as follows:

a. Make the product portable.

b. Use as many reusable components as possible.

c. Make the product easy to maintain.

d. Maximize data and control abstraction.

The following is a discussion of each product-specific requirement and the software engineering requirements which affect it. The requirement for a cyclic environment is essentially a practical decision. There are basically two ways to handle asynchronous events. The first method is to run a cyclic process fast enough to service the event without a noticeable effect on event response time (such as changing a switch position, observing when a light comes on, or some other hardware event). The other way is to establish a system of interrupts for the asynchronous events and interrupt handlers to interface with the event processing software. All of this is then wrapped in an Ada tasking construct. There are three reasons for not choosing the Ada tasking/interrupt approach. First, high context switch times were witnessed during our benchmark testing. Second, risk was involved in mapping multiple hardware interrupts to user level software. And third, some iterative processes still had to be calculated cyclically.

The requirement for central control across all processing elements is a departure from previous endeavors where control is more regional in scope. Unlike the previous requirement, this one lends itself well to various software engineering attributes such as packaging, data abstraction, and information hiding. The requirement is implemented via the three-tier approach to the executive. When the simulator is integrated, the master executive manipulates data that is common to all clusters (such as shutdown state changes). This data is in a separate package which only the master executive writes. Each cluster executive then reads the system level data and controls its cluster accordingly. Similarly, the cluster executive manages data which is common to all sequencer executives in the cluster. This concept is taken one step further by sequence executives which manage all environment-related data used by the application software running under them. This concept of partitioning data into separate packages takes advantage of two of Ada's built-in software engineering concepts: data abstraction and information hiding. The requirement to provide central control is not limited to the management of common environment data. The cluster executives control the execution of all sequencer executives in their cluster. The original implementation was accomplished by:

a. Setting the time_to_executive flags

b. Monitoring Frame_complete flags

c. Controlling subordinate sequencer executives, executive os_processes, via an Ada interface to the operating system.

These activities use standard Ada programming constructs. The Ada interface to the O/S dependent software is localized to a single package to enhance transportability. However, controlling an os_process via an Ada interface to the O/S proves to be an exorbitantly time-intensive activity. Depending on the number of APUs to be rescheduled and the number of lower rate CPU executives, 25 to 35 percent of the cluster executive's frame time would be consumed by this activity. This directly conflicts with the requirement to minimize overhead. The resulting decision placed the functionality of subordinate sequencer executive's control into each cluster's clock handler. This decision reduces the activity's execution time by up to sixfold! Even though the clock handler is an assembly language program, all of its data (i.e., os_process names, etc.) is still managed by the RTSE, and thus retains much of the original Ada software engineering attributes.

The requirement to provide automatic load leveling is met by the use of multiple executive os_processes running on the CPU and APUs. Although the actual code implementing the First In First Out (FIFO) queues on the APU is O/S dependent, the system design and structure is portable since most computer operating systems provide a mechanism to accomplish this activity. Since the O/S-dependent code is isolated in one package, it can be conveniently replaced by the appropriate interface software when transporting the design to another system. The disappointing reality is that at the time

these load leveling decisions were being made, current implementations of Ada tasking did not support this requirement. In theory, Ada tasking should support a load leveling activity for a single processor. Our initial benchmark test results of Ada tasking in the 1986 time frame did not support the Ada tasking implementation we had expected. We observed that the entire os_process paused if even a single Ada task within that os_process paused. We had expected the pause to cause a context switch to another Ada task. In addition, the Ada task context switch took approximately as long as an os_process context switch. It was anticipated that Ada tasking would be much quicker since it was strictly part of the application software and did not depend on an operating system interface. Ada's failure to recognize multiprocessor cluster is a fundamental deficiency in an otherwise remarkable language.

The requirement for single-point control is considered to be relatively uncontroversial. In earlier simulators, with fewer computer hardware components, the process of bringing up each cluster individually was relatively simple. However, this simulator was considerably more complex and the process would have been very detailed and involved. Thus, this requirement was reasonable and logical. For hardware reasons, the design (i.e., the bootstrap product) to accommodate this requirement uses multiple, nearly identical os_processes across the simulator. These os_processes differ only by the actual data package which contains the cluster's unique personality. In light of the requirement to make our products more reusable, the use of a generic main procedure would have been the preferred implementation. However, Ada does not permit the passing of a package as a generic formal parameter. Therefore, the pseudo-generic technique of package renaming was used. The product software was identical in all cases, thus adding to the maintainability. The local version of each cluster-specific package is referenced throughout the bootstrap product. Each version of this software contains a rename of a cluster-specific package to a local package name. Since the cluster-specific package is not in the direct withing structure of the main, the main is the same in all clusters and meets the criteria of reusable software. However, a separate library or sublibrary must be maintained for each cluster of this software because of the renaming of the cluster-specific data package to the local package name. This approach poses no problems in the design and development phase; however, the use of multiple libraries/sublibraries complicates load production due to library management, compilation, and linking issues.

This real-time environment had to support a simulator that was roughly an order of magnitude larger than previous ones, and the old software control methods were not sufficient. The requirement to be a "state-driven" environment eliminated some of the size-related issues and provided a mechanism to actively support control abstraction. This state-driven requirement addresses the overhead issue as well as size and control abstraction. As previously stated, the software was tailored to support only those activities related to the state in which it runs. At first, it was feared that this approach would greatly increase the amount of the user application software because there would be many copies of nearly redundant code. However, Ada packaging can be used to maximize utility of a state-driven executive, if the software is organized by the functionality it must support in any given state. There are three steps involved in this process. The first is to organize (via packaging) the software of a particular operation (e.g., an engine model) into the smallest identifiable operations, known as primitives. The second step is to organize the primitives into groups that support the required functionality in a given state. For example, all primitives that are involved in integrating quantities would be included in the group of primitives that execute in real time and would not be included in the freeze group of primitives. These groups are simply Ada procedures referred to as Control Managers (CMs). The third step is to identify to the executive the applicable execution state for the CMs.

This approach has a twofold advantage. First, the decision as to when software executes is made off-line during development and does not contribute to the real-time overhead. The second advantage is that only software applicable to the current state executes. For example, a popular method of summing a value in real time is to calculate a delta value, scale that value, and add it to the running total. In previously used approaches, the scaling constant would have two values: one for real time and another for freeze. The real-time scaling constant would equal the inverse of the frame rate and the freezing constants would equal zero. The software would always execute and simply change scaling constants to implement real-time or freeze. This is not the case in the state-driven approach. In our state-driven approach, the summing procedure would only run in the real-time state and would not be called in the freeze state. The states in which a procedure is to run is listed in the Configuration Control File (CCF).

The system definition for the entire simulator is contained in the CCF. This file correlates the user

software (Control Managers – CM), states in which each CM will execute, and which executive os_process contains the CM.

Implementation – As previously mentioned, the executive steps through the three superstates: pre-synchronous, synchronous, and post-synchronous. During the pre-synchronous superstate, each executive performs the user initialization and the executive initialization. The user initialization procedure provides a standard interface to the Executive for the user software. Within the initialization superstate, all necessary user-defined procedures are called to perform initialization for the users. Once a particular Executive completes its initialization activities, it waits for all other Executives to complete their initialization before entering the synchronous superstate. The waiting is accomplished via rescheduling for APU executives, suspends for CPU sequencer executives, and a clock interrupt for master/cluster executives.

After entering the synchronous super state, the following functions are performed by the Executive:

- Timing;
- Frame and Cycle Boundary Updates;
- Scheduling of OS Processes and User Procedures.

If the timing of frames has been requested, the real-time clock is read at the beginning and end of the frame and statistics are kept.

The master/cluster Executive keeps track of physical frames, cycles, and changing of states. Once the frame and cycle boundary logic is complete, the master/cluster executive manages all sequencer executives within the cluster. Each sequencer executive calculates its own logical frame number. Then the sequencer executive calls the user procedure (CMs) based on the state and logical frame number. If requested, the sequencer executive times user procedures and maintains the statistics.

Interfaces

Requirements – On previous simulators, information was shared through a construct known as a global data pool. These constructs were subject to data inconsistency, unnecessary global visibility, and data corruption.These factors drove the development of the requirements for the interface software.

For interface software, we observed very little conflict between the product-specific requirements and software engineering requirements, because this product is system level in nature. The product-specific requirements are as follows:

a. Provide interface between software components within the same os_process and between os_processes existing on a cluster of processors.

b. Provide the capability to save the simulator environment and restore the environment for either a power failure or an instructor's request.

c. Minimize the overhead associated with inter-software communication.

d. Provide an automatic method to manage the interfaces, to generate the interfacing software, and to document the interface.

The software engineering requirements are as follows:

a. Ensure data integrity for all interfaces.

b. Ensure data consistency for all interfaces.

c. Minimize systemwide recompilation of application software.

d. Actively support data abstraction of the interface objects.

e. Actively support information hiding of the interface.

f. Design the product to be maintainable and reusable.

The requirement to provide interfaces between software components within the same os_process and between os_processes existing on a cluster of processors has a far-reaching effect on the RTSE design. Due to the large size of this simulator, it was immediately apparent that this simulation would span many os_processes and processors. A method to handle the interfaces had to be established. The first task was to identify hardware which supported the software resource needs. The options were to find either a single memory in the 256 to 512 megabyte range or an arrangement of interconnected memories which function as a single memory. In conjunction with the hardware requirement, we needed a compatible validated Ada compiler to be paired with it. In our early 1980's study, the coupling of these two requirements effectively reduced the field of viable vendors to one.

The vendor-supplied interface mechanism is at the Ada package level and uses the construct of a "Task Common" area (TCOM). Multiple Ada packages can be linked to a particular TCOM without any coding changes to the Ada packages. This preserves the original data abstraction designed into those packages. All os_processes linked with a par-

ticular TCOM could have visibility to any data if they withed the applicable package on the TCOM. This method preserves Ada's visibility rules. Through this mechanism, software units can interface with other software units anywhere in the system using normal Ada constructs. This method is used by all three areas of the RTSE.

The interface software requirements tend to reinforce one another, with the requirement for low overhead being the only exception. This is a distinct contrast from the control software requirements. Although the interface requirements are basically symbiotic, many challenges arose during the implementation, most stemming from the sheer size of the simulator software.

The RTSE maintainability requirement encapsulates the entire product and was coupled with the requirement to automate the generation, management, and documentation of interface connections. The Interface Management Data Base (IMDB) and the supporting software were the result. The IMDB and supporting software generates interface packages through an automated, menu-driven process. Our system currently manages approximately 10,000 interfaces. It also automates the process of ensuring data consistency between the interfacing components. Since this automated process is the only method for generating or changing interfaces, no informal modifications can be made to the interface software, hence protecting the integrity of the interfaces. Once the IMDB and associated software have been through testing, generated packages need not be tested. The reusability of the IMDB and associated software in this fashion supports maintainability of future interfaces or changed interfaces providing effectively pre-tested packages to the user.

Although understood by most application software engineers, the concept of the FORTRAN data pool and its symbol dictionary was simply unfeasible for a number of reasons. The first reason is that the number of interfaces is many times that of previous simulators. Any attempt to manually manage that unwieldy number of interfaces was impractical. The Ada packaging capability, coupled with the requirement for information hiding, assisted in the solving of the interface problem. Packaging under Ada allows related objects to be grouped together in packages small enough to be meaningful and managed at all levels of integration. Data integrity is greatly enhanced by use of Ada packaging. Unintentional corruption of data is practically impossible with Ada because one must consciously "with" a package in order to alter any of its contents. These packages are generated from the IMDB and the Configuration Control File (CCF).

Another challenge was the view, entertained by some, that the benefits of packaging and information hiding did not fully compensate for the loss of flexibility to change data especially in a test environment. In certain dynamic software tests (e.g., Air Vehicle modeling), large numbers of objects are manipulated. However, this situation was anticipated and a separate product (CoASTE – Coherent Automated Simulation Test Environment) outside the realm of RTSE was provided to support this activity using primarily standard Ada constructs.

The views on the level of abstraction ranged from permitting only three types — Integer, Float, and Boolean — to "if Ada supports it, then use it." The final decision was to support data abstraction to the maximum extent possible with only a few restrictions. The principal restriction limits the full path name of an object to 112 characters in length.

The final challenge was the overhead issue. An approach which ensures data integrity and consistency may use more overhead than one that does not provide those services. The overhead consists of execution time and memory allocation. For a software project this large, it was felt that data integrity and consistency could not be sacrificed in favor of a low overhead requirement. To minimize the impact of increased overhead, several things were done. To lessen the impact on any individual, the overhead of ensuring data consistency and integrity was distributed over both data importers and exporters. Activities such as constraint and range checking could be employed during product development and validation, and then turned off (i.e., compile without them) to meet the spare time requirement. The use of a double buffering scheme was implemented to protect data integrity and ensure data consistency.

The Design – To accommodate the many design requirements, one off-line and two real-time products were designed. The off-line product was designed to manage and document the software component interfaces as part of an integrated load build system. The actual automatically generated (autogenned) interfacing software and the software to do the capture and restore comprises the real-time software. The collection of these products is known as Shared Memory Management (SMM).

The off-line software incorporates the use of an Oracle database to manage and document the interfaces and to build the interface software. This part of SMM consists of three sections: the Interface Management Data Base (IMDB), Data Base Access (DBA), and Software Interface Generation (SIG).

The DBA section consists of all software that is needed to insert or extract database information. DBA is written entirely in Ada and uses the Oracle provided interface to the database. Access to the IMDB for entering and deleting information consists of two levels of activities, Administrative Tool Package (ATP) and the Element Tool Package (ETP). Paper forms are used to describe the software interfaces changes and are processed via ATP and ETP on the IMDB.

The ATP permits the IMDB Administrator to insert, update, and delete IMDB entities via the IMDB ATP menus. It is the IMDB Administrator who controls all direct IMDB change activities (via the IMDB information forms and the element auto-entry text files), while both IMDB Administrator and users (ETP) are granted privileges to generate IMDB reports.

Also, the ATP will provide the IMDB Administrator with the ability to extract information from the IMDB and generate connection manager packages (Import and Export/State), shared data packages (Shared Memory), and various reports.

The user requests IMDB changes through the ETP. While the menu choices are similar to those of the ATP, the user's changes do not go directly into the IMDB as ATP changes do. User changes are stored in an auto-entry text file. The IMDB Administrator then reviews the changes, and if approved, processes them into the IMDB. The ETP allows each user to generate various reports from the IMDB. It is this part of the SMM IMDB process (i.e., IMDB information forms and element auto-entry text files) that is solely under control of the users.

The users must define all export objects relative to each control manager in the IMDB. A logical group consists of an importer, one or more control managers, and an exporter. Logical groups (Figure 6) are defined in the Configuration Control File (CCF).

The last part of SMM DBA is the extraction function. Each import CM, export CM, and applicable state data is associated with a logical group. An information list for each is extracted from the IMDB based on the definition of the logical group as defined in the CCF. These information lists serve as input to the Software Interface Generation (SIG) process.

The Software Interface Generation (SIG) process will generate shared data packages (i.e., Shared Memory) and connection manager packages (i.e., Export/State CM packages and Import CM packages). During software generation of connection managers packages and shared data packages, all non-interfaced export and/or import objects will be filtered out. The information for this process is contained in the information lists provided by the DBA extraction function. The resulting software interface product consists of CMs and related shared memory packages.

The connection managers provide inter-simulation communication and execute in real time as part of the executive in the same fashion as the control managers.

The connection managers include code to import and export inter-simulation communication variables (shared objects). They also include code to transfer state data between local data area and a shared data area for the snap/reset process. Also included in the connection manager is software to initialize the associated "shared memory" to values derived from the IMDB. Since connection managers are built software, each connection manager has the same format.

Each export connection manager has a single shared data package associated with it. The shared data package contains a type package defining a record containing all of the variables to be exported and a current side of memory pointer. This record is a "side of memory". An array of two of these records is then used to provide two "sides of memory" so that one "side of memory" can be active (i.e., exported to) while the other "side of memory" remains constant. Therefore, the snap process reads valid, consistent (with respect to time) data from the other "side of memory", thus providing doubled buffering of the data.

A second package, the "shared memory" package, contains an object of the array type defined in the type package. Each side of shared memory is separated into export data and state data. State data is information needed for reset/restore. Shared memory uses buffering techniques, where necessary, to manage data consistency among sets of export data being stored. In such cases, multiple buffers of export data are maintained, avoiding simultaneous reads and writes to a single area (Figure 7). The third and final package contains the exporter, the initialization for the "shared memory", the state data export, and the state data import.

The import side of the connection manager contains only one package. This package has an import procedure that transfers "shared objects" from one or more "shared memories" to the user's software local data area.

The import procedure will use user-defined filtering techniques (e.g., interpolation and extrapolation), where necessary, to manage data consisten-

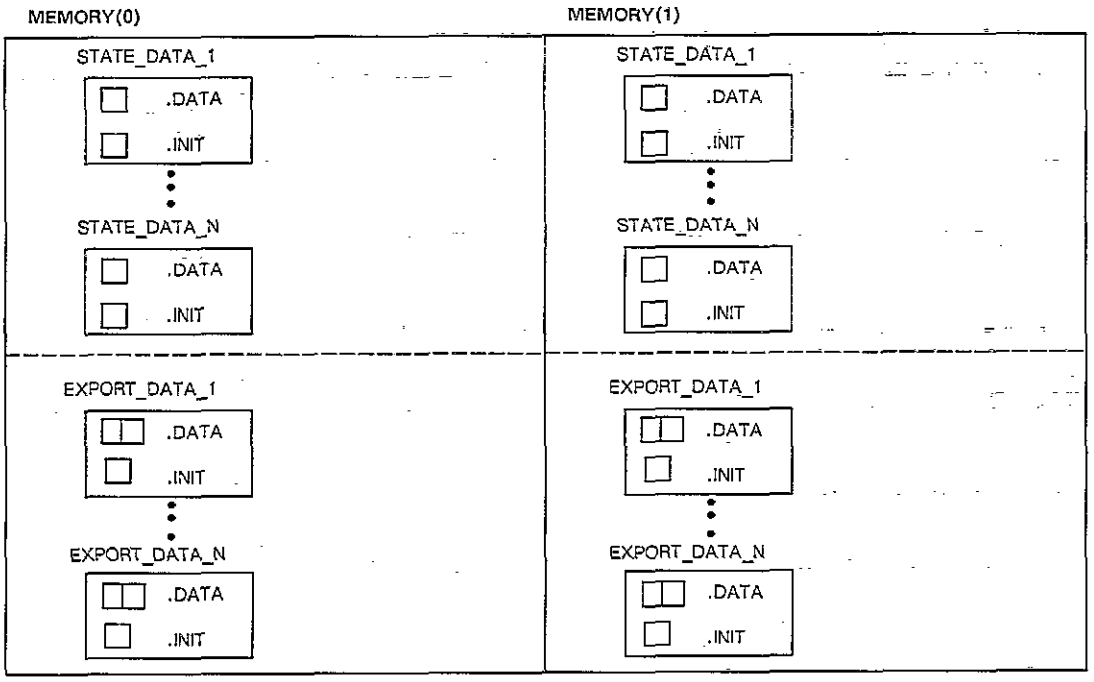

Figure 7 Shared Memory Diagram

cy among sets of import data moved from the shared memory areas.

The following is a description of the snap/reset software used to implement the requirement to save the simulator environment. During a power fail snap or an instructor requested snap, the current side of shared memory pointer is toggled. During a snap, one side of memory will be written (i.e., saved) to disk for later use in the reset/restore process, while the other side of memory becomes active to support importing/exporting. The side of memory being saved to disk will not be updated until the save is complete. A reset request to a previous point in time is the reverse of the above process.

Real-Time I/O

Requirements – The I/O requirements are divided into two groups: Software Engineering requirements and Product Specific requirements. Each group of requirements will be listed, followed by detailed information on each Product Specific requirement. Since the Software Engineering requirements generally support or conflict with the Product Specific requirements, these relationships will be covered in the Product Specific requirements discussion. Software Engineering requirements which are not fully addressed in the other sections are grouped at the end.

The Product-Specific requirements are as follows:

a. No-Wait I/O

b. I/O Across Processors

c. I/O Across Clusters

d. Disk I/O : direct_io

e. Disk I/O for objects greater than 64K

f. Output messages to error logs, audit logs, and terminal in specific format

g. Disk Marking

The Software Engineering requirements include the following:

a. Portability

b. Reusability (Internal and External)

c. Maintainability

d. Data Abstraction

Challenges/Conflict Resolution – The following challenges/conflicts had to be met/resolved:

a. No-Wait I/O

A real-time environment requires no-wait I/O. Our real-time environment's integrity could not withstand the delays of Ada-provided I/O constructs. No-wait file I/O in our context does not mean that the actual I/O happens at the instant requested but rather that the requesting software need not wait around for the I/O to complete before it can go on with its other processing. It is the difference between waiting 20 minutes at Piz-

za Hut for your pizza to be made and calling ahead so that the only time you spend is on the phone to place your order and walking in to pick it up.

Originally we believed that Ada tasking would be the key to the no-wait I/O requirement. When we tried to implement it, tasking was slow. While a particular task waited for a rendezvous, the entire os_process would come to a halt.

A system of interfaces was devised to serve as a communication block between I/O-requesting and I/O-processing software. The interface structure, made up of an interface record array and buffers, serves as a queue that requests can be immediately stored in so the user can continue processing. When the I/O-processing (I/OP) has time to execute, it reads a request off the interface record array, reads the associated data from the appropriate buffers, and completes the request. The interface structure holds all information that is needed to process the request and the request's status.

b. I/O Across Processors

The RTSE is required to support I/O regardless of what processor the requesting code is running on. The system's solution to APU I/O would cause the APU tasks to execute in an unpredictable fashion.

Our solution lies is in the use of the interface structure. All I/O requests are placed in an interface record regardless of where the task is running. Specifically, an APU task's I/O request is placed in an interface record and the task continues without influencing its execution order. The I/O processing software always runs on a CPU where execution order is strictly maintained.

c. I/O Across Clusters

While each cluster has its own disk, the customer required that disk writes occur on only one disk when running integrated. Our O/S / Ada does not support I/O across clusters. RTSE solved this dilemma using a system of slave clusters and a single master cluster. Each cluster, whether master or slave, has its own interface structure, I/O Processor, and disk. Recall that a cluster is made up of a CPU and one or more APUs (Figure 3). The clusters designated as slave have an I/O Processor to execute all requests stored in the local area of their interface record (Figure 8). Requests in the local area are performed on the disk associated with that cluster. The master cluster has access to the master portion of all the interface record arrays. By storing all output requests when integrated in the master portion of the interface record array, all write requests will be performed on the master disk when integrated, just as required.

d. Disk I/O : direct_io

An effort was made to make the I/O requesting software structures and formats Ada-like; this was done to provide the user with a familiar format. The result was the generic Request_direct_io package. The no-wait I/O request software mimics direct_io with similar procedure and parameter names being used. When the request is processed by the "I/O Processor" software, it uses Ada's direct_io, making this service software transportable to other hardware/compiler combinations.

e. I/O for Objects Greater than 64 K

Since request_direct_io is an extension of direct_io for the RTSE, it carries with it direct_io implementation limitations. While in the preliminary design phase, we considered breaking up the data into sizes direct_io could handle but there was an unacceptable cost attached in the form of overhead time and space, and lost Data Abstraction.

Since machine-specific direct_io object size limits already removed an element of portability *and standard Ada constructs were not suitable* for the situation, software was developed that interfaced with the operating system services. Contiguous_io was the result.

To the user the choice between request_direct_io or request_contiguous_io is based on the size of the object. While contiguous I/O allows faster retrieval due to its file structure, it is used primarily because it allows unlimited object size and only one stable copy of the object exists at any one time. Direct_io's implementation keeps one or more copies of the object in system data buffers and multiple copies in an array of user buffers. As the impact of this has systemwide effects, guidelines based on the size of the objects were establish to indicate the type of I/O to use.

f. Output Messages to Error Logs, Audit Logs, and Terminal in Specific Format

Besides two-way communication with files, there was a need to for a write-only utility. This utility is strictly for the output of strings, hence need not be a generic. Any Ada object can be translated into a string by use of the image attribute.[2] The message_request procedure for the user and I/O Processor's message_request operations compose this implementation.

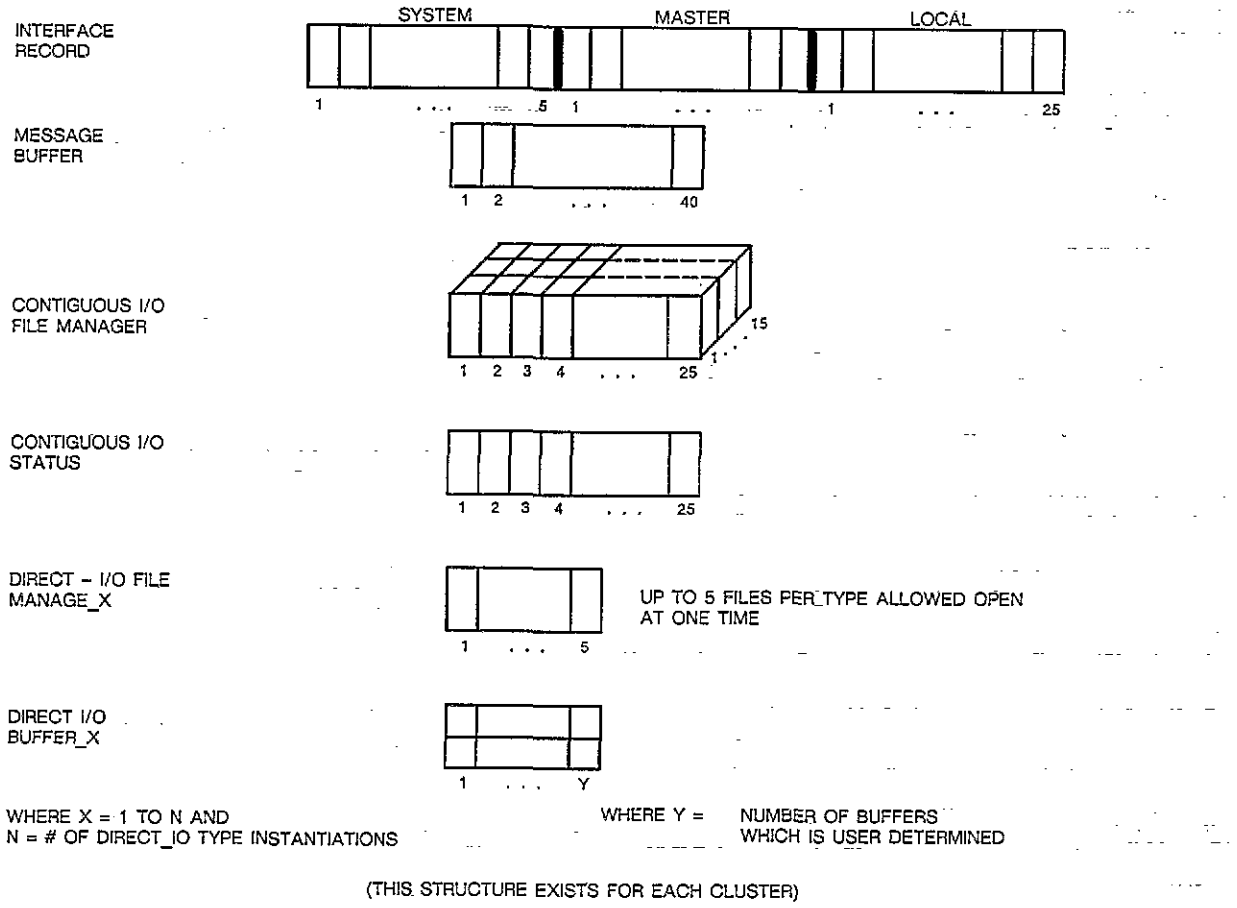

Figure 8 I/O : Interface Structure

g. Disk Marking

Since the O/S did not provide the user a service to request and status disk marking, that too became a RTSE I/O requirement. Portability was not possible given the fact that our Ada version did not provide an interface and there were no O/S services to indicate or modify the status of a disk available to the user. The devised interface used a pragma interface to assembly code: a machine-dependent implementation.

h. Additional Software Engineering Requirements

To make the software portable for use on other hardware or with other compilers, we made an effort to use the standard Ada constructs of text_io and direct_io while avoiding system-dependent tools. Reusability was achieved through creation of generics which were instantiated multiple times within the simulation software and perhaps can be used on future projects. Generics and packaging added to the maintainability by limiting the scope of changes.

Another way to improve maintainability is to increase software readability. We encouraged the use of enumeration types. We implemented an English-like naming practice for types, objects, and packages. We required that acronyms and abbreviations be defined in the prologue of each unit in which they appear.

The quest for data abstraction influenced the allocation of objects and functions into packages and the grouping of data into records. For contiguous_io we provided a procedure which provides the exact memory location of an object rather than allowing the user to calculate this information. Good data abstraction is attained by not allowing options to go around this structured method. By forcing the user of contiguous_io to use this function to determine the address of an object in memory, a degree of control and a bit of calculation is automated, removing a location for potential user errors. An error of sending the wrong address could be fatal to the executing software if an address for the executable code rather than the desired object is written over. Re-

moving this possibility can only be a boon to the I/O system.

Implementation – The no-wait adaptation of I/O was coined Real-Time I/O (RTIO). To meet the requirements, the RTSE was to supply the user with four I/O related abilities: no-wait disk I/O, no-wait disk I/O for types greater than 64 K in size, message I/O, and disk marking. The user refers to these services by the following names: request_direct_io, request_contiguous_io, message_request, and disk_marking. Collectively, we refer to the software the user interfaces with as the "User Module". The User Module software fills the appropriate interface structure arrays and buffers.

For example, when a user requests a message, the message is stored in an available spot in the message_buffer and an interface record is set up with the corresponding message_buffer index and type of message request stored. Pending is stored as the status in the interface record. The user who made the request can then continue processing while the request waits in the interface record queue for the I/OP.

The interface structure is used in conjunction with all I/O requests, providing a separate buffer segment in the interface_record for system, master, and local information (Figure 8). Additionally there is one Message_buffer, one Contiguous_io_file_manager, and one Contiguous_io_status buffer per cluster. Conversely, there exists one direct_io_file_manager and one direct_io_buffer for each type instantiation of direct_io. This number directly correlates to the number of request_direct_io instantiations and is user determined.

The key to the interface between the User Module software and the I/OP software is the interface record, which is made up of three parts: system, master, and local. Each is a circular fill queue. The system interface record is reserved to report internal RTIO errors. I/O requests to the master cluster are queued in the master section,and I/O requests for the local I/O Processor are queued in the local portion.

The I/OP functioning is based on how we had hoped Ada tasking would work. A simplified structure follows. The I/OP consists of a big loop which steps through the interface record array checking the statuses. Using a case statement which keys off the status, and another which keys off the type of request when the status is pending, the I/OP executes the desired I/O requests as appropriate. As an I/OP is on the CPU processor of each cluster, it can be linked at an appropriate priority so as to not interfere with the execution of the other tasks, hence run invisibly to the simulation software.

To circumvent the limitation of excluding packages as allowable formal generic parameters, the principle of the pseudo-generic was used for the I/O Processor (I/OP). I/OP software refers to a predefined set of interface arrays, objects, and procedures while each cluster and user determine their own unique names. This pseudo-generic principle comes into effect as a series of renames causes the unique packages to be renamed to the predefined set that the I/OP software recognizes.

SUMMARY

The development of the Real-Time Simulation Environment for the B-2 ATD was a challenging undertaking even though many other environments had been developed before. Many felt that the challenges encountered were a direct result of this being an Ada project. Although the lack of experience in Ada on an actual simulation project was a contributing factor, we have judged the following factors to be of greater influence in meeting our challenges.

The "early adopters syndrome"[1] most likely had the greatest impact on our development effort. This syndrome is characterized by the availability of only a few validated Ada compilers and even fewer mature Ada compilers. Our compiler was immature at the start of the program and has gone through rapid changes and multiple revisions. We too have had to adapt to these changes, report unfavorable situations not detected by the validation tests, and accept the workarounds provided until the next revision was available. While we believe that these inconveniences were common to most compiler vendor/customer teams at the time, they are unique to immature compilers.

Some unique Ada features were handled differently than were originally anticipated. The Ada tasking model provided had significantly impacted the design of the control and I/O sections of the RTSE. If the essence of Ada tasking can be accomplished via an interface to O/S services, then it is possible for a compiler to do the same or better. The compiler that more fully exploits the intent of Ada tasking and functions in a reasonable amount of time will have a significant advantage over its competitors in the simulation market.

The size of the B-2 ATD had a significant role in determining the direction of the development process. However, we believe that the structure inherent in Ada had a positive influence on the development process. This is especially true now as the project and compilers are maturing. This view is

consistent with the life cycle cost studies done on Ada and other languages.

The failure of Ada to support the passing of packages as formal generic parameters had a negative impact on the development of the control and I/O sections of the RTSE. Several legitimate ways were found to approximate the effect. As mentioned earlier, the methods were package renaming and the use of built software from standardized templates. However, the use of packages as formal generic parameters would have made this task easier and more maintainable.

Since Ada does not recognize multiple os_processes, many schemes have been developed to provide communication between os_processes. Although our experiences do not include an exhaustive list, the method provided by our current vendor is by far the best system we have seen. This method provides for the linking of Ada packages into a common identifiable area. This area can then be linked against when linking os_processes. These os_processes can then utilize the data in those packages. Other systems we have investigated only recognize the sharing of individual objects between os_processes. Even with renaming of objects, it can be considerably more difficult to map packages through discrete objects. It is our opinion that compiler vendors who want to compete for large multiprocess Ada projects will have to support inter-process communication at the Ada package level.

We feel that the advantages of an Ada-based system greatly outweighed the challenges imposed by Ada and the related "early adopters syndrome". As we look forward to new projects and possible rehosting on different hardware, we have already observed benefits of Ada.

REFERENCES

1. "Ada Issues Awareness Seminar" by Reifer Consultants, Inc., 25550 Hawthorne Boulevard, Suite 208/Torrance, CA 90505
2. ANSI/MIL-STD-1815A - 1983, "Ada Programming Language"

ABOUT THE AUTHORS

Walter E. Zink is currently a Section Head – System Engineering at CAE-Link Corporation, where he has held management and technical positions since 1986. His current effort is devoted primarily to system-level software design of the real-time environment that supports the Ada architecture in use at Link. Prior to that, Mr. Zink was employed by the General Electric Company for twelve years, where he held various technical and management positions. His principal activities there were in the areas of Computer Based Instruction and Artificial Intelligence/Expert Systems. Mr. Zink holds a B.S. degree in physics from Harding University.

Jill Neebe is a Systems Engineer at CAE-Link Corporation, where she has been employed since 1987. Her current effort is devoted primarily to system-level testing of real-time Ada software. Prior to this, Ms. Neebe worked in Software Quality Assurance developing and implementing instructions for quality assurance in a DOD-STD-2167A environment. Ms. Neebe holds a B.S. degree in mathematics from Virginia Polytechnic Institute and State University.