ANALYSIS OF PATHS OF TRANSFER TO ADA TECHNOLOGY IN TRAINING SYSTEMS

Don Law and Gary Croucher
Encore Computer Corporation
Ada Development, MS 404
6901 West Sunrise Boulevard
Fort Lauderdale, FL 33313

ABSTRACT

The training and simulation systems of the 1990's will be more complex as total trainers and multiple participant systems mature. The Ada language offers the software engineering fundamentals needed to handle the greater complexity and the life cycle advantages to reduce software costs. To use Ada, training system vendors must decide on a method to transfer to Ada.

There are two basic approaches for transfer to Ada technology. At one end of the spectrum is the "generic" approach, which uses the generic, standardized Ada structures for the implementation. This path promises the benefits of modern software engineering, easier maintenance, and greater portability. This path also requires the cost of quality Ada training, the risk of using a new technology, and possible performance degradation.

At the other end of the spectrum is the "proprietary" approach, which depends on other non-Ada, more traditional support systems for the real-time implementation. This approach promises a more appealing transition since risk of new technology is lowered, but there are tradeoffs such as the predicted higher life cycle costs and the loss of the software engineering advantage offered by Ada.

Neither approach is superior in all cases, but each has its advantages and disadvantages, which are classified and weighed in this paper. Analysis is based on the application speed, efficiency, portability, determinism, software training, and maintainability. A survey of the philosophy of some of the real-time Ada systems currently available on the market is presented. Systems are evaluated based on the cost/benefit areas established in the paper. Developers of Ada real-time training and simulation systems can use these guidelines to plan their approach early in the project to ensure that the requirements will be met in a cost-effective manner.

INTRODUCTION

The Ada language, with its rich set of features and well-defined structure, offers the software engineering fundamentals needed to develop complex real-time applications along with the life cycle advantages needed to reduce software costs. As the training and simulation industry chooses to fund new projects in Ada or convert existing projects to Ada, they must decide on the approach that will be taken to convert existing resources to Ada technology. There are two distinct approaches for transfer to Ada technology: a "generic" approach and a "proprietary" approach. The generic approach restricts project implementation to standardized Ada structures and semantics, while a proprietary approach provides additional non-Ada support typically in the form of runtime procedure calls that support many of the features of the underlying operating system. Neither approach can be considered superior in all cases, so before making any purchases, buyers should carefully consider their situations to determine which methodology is most suited to their individual needs.

This paper analyzes and compares the generic and proprietary approaches to Ada development. A technical analysis is presented for each approach that emphasizes key areas of Ada functionality, such as tasking and interrupt handling, and how efficiently this functionality is implemented. Finally, a comparative analysis is performed that contrasts these two methodologies. The comparison is based on the attributes and measures of software engineering that are considered important when building large real-time systems. Application speed, efficiency, portability, maintainability, and development costs are among these attributes. This analysis is intended to serve as a guideline to assist developers of Ada real-time training and simulation systems in planning an approach early in the project to ensure that project requirements will be met in an efficient and cost-effective manner.

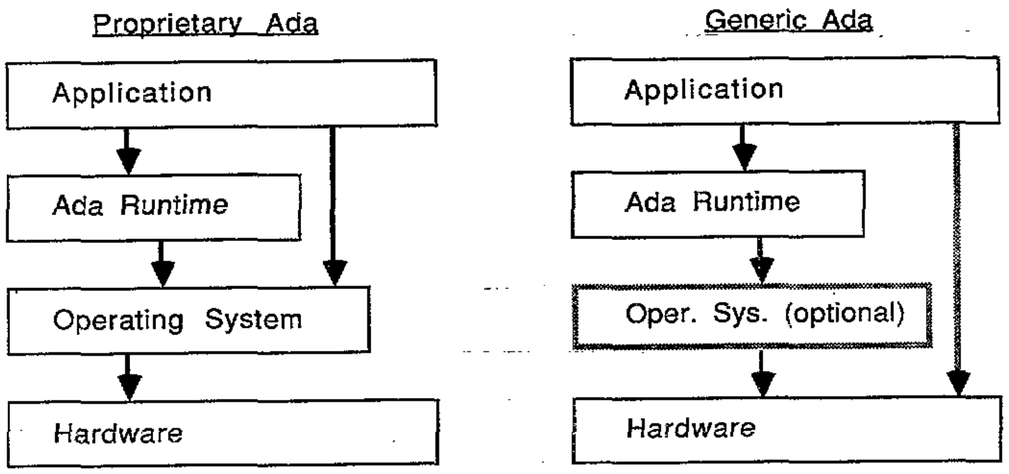

Figure 1. Typical Proprietary vs. Generic Implementation Heirarchy

GENERIC ADA ENVIRONMENTS

A generic Ada environment provides only generic, standardized Ada support for development of target applications. Typically, the generic Ada runtime will directly control the target hardware. This approach is commonly known as a bare machine implementation of Ada, since no underlying operating system exists. Of course, a purely generic implementation of Ada can exist above an underlying operating system, as long as no underlying proprietary functionality is visible to the application. Generally, this type of Ada development environment provides strong independent support for critical areas of Ada functionality. For the purposes of this discussion, we will assume that a generic Ada environment has no underlying operating system, and that all operating system functionality is performed by the Ada runtime.

Device I/O

When no underlying operating system exists, all of the driver support for performing device I/O must be implemented as part of the Ada runtime. This is one of many factors that contribute to an excessively large generic Ada runtime. This approach can actually prove to be more efficient, however, as runtime procedures can be specifically designed to meet the needs of Ada I/O. On bare machine systems, the additional size required is compensated by the memory that is freed up by removing the operating system.

Interrupt Handling

The generic Ada mechanism for handling interrupts is defined in Chapter 13 of the Language Reference Manual. It is integrated with the Ada rendezvous mechanism used for inter-task communication. Interrupts behave like the rendezvous, except that there is no software calling task. The tasks that are to handle the interrupt are abstracted from the normal flow of execution. When the user is developing the interrupt task, he may design it as a completely separate execution entity. Visibility to other parts of the program is governed by normal Ada rules, just as if the interrupt task were a procedure in the application.

There is only one provision for connecting an interrupt to an Ada task. The interrupt must be connected to an entry within that task using an interrupt entry representation specification. A number in the specification of the task specifies which interrupt in the underlying system is to be associated with that task entry.

One drawback with the generic mechanism is that interrupts must be processed as a "scheduling event," which means that the Ada scheduling rules must be followed when the interrupt occurs. This prevents the "fast interrupt" concept of processing the interrupt without entering the scheduler.

The generic mechanism offers the advantage of simplicity in that the interrupt scheduling behaves exactly like the inter-process scheduling. This contributes to a more consistent system design. The tradeoff to that is that all interrupts are subject to the same behavior. The imposition of the Ada scheduling rules on interrupts can make "fast interrupts" (no scheduling evaluation) difficult to implement.

If an application requires unusual interrupt support, such as extremely fast interrupts, then generic Ada may not be a sufficient platform (but see the alternatives discusses below). If the traditional interrupt concept is all that is required (that is, the connection of an interrupt to a particular logical sequence of code) and the interrupt design is complex, then the generic Ada solution is a good choice. It provides the powerful semantics of the language to manage a complex interrupt design.

Exception Handling

The exception mechanism in Ada is one of the most complex parts of the language to implement. A good generic Ada implementation will map all error conditions on the target into Ada exceptions.

Traditional operating systems have relatively weak exception handling capabilities. Often, the task is aborted altogether. Some systems provide an error receiver capability, but usually the error handling code does not have visibility to where the exception was actually raised. The generic Ada mechanism is quite sophisticated by comparison. Error handling can be done more gracefully within the language. Exception handlers may be localized to any particular area of code down to the individual statement level. Exceptions are user definable, and may be propagated up the call chain.

The generic Ada exception system opens up a new realm of design possibilities for real-time systems. Error conditions can be handled in the same context that they were caused.

One difficult issue that must be addressed in real-time systems is what to do with asynchronous error conditions, such as a memory parity fault. At the time that the error occurs, any task in the system could be executing. Furthermore, any call chain of procedures and functions could be in progress when the conditions occur. If graceful recovery from such conditions is a requirement for the application, then adequate exception handlers must be installed in the code. These handlers need to exist in any task that may be in execution when the fault occurs. Since the addition of exception handlers is likely to be extensive, the overhead of these handlers must be examined. It is possible for a vendor to implement Ada exceptions such that no additional execution overhead is required when an exception handler is added to a piece of code. This should be considered when selecting an Ada target environment.

Another complication with asynchronous error conditions is the possibility of such a condition occurring while no tasks are executing, that is, while the system is idle. In such a case, there is no scope to deliver the exception to, so the implementation must perform some target-specific operation.

Concurrency

The unit of concurrency in Ada is the task, and the only way for applications to support concurrent code segments is to write these code segments as task structures. All scheduling mechanics, including scheduling called tasks to run and queueing calls to task entry points, is handled by the rendezvous mechanism of the Ada runtime and are well-defined by the language. There may easily be more logical elements of execution than physical processors, allowing better abstraction in the system design.

Multi-Processing

Since the generic runtime is self-contained and has control of the hardware, a distributed Ada environment can be implemented by the vendor as an extension to the existing runtime. All functionality required to support additional processors will be contained totally within the runtime, since the runtime alone controls the hardware.

This extended or distributed runtime will also be responsible for determining how the Ada application will be divided up among the participating processors. It is conceivable that the runtime could choose to partition the executable application on any or all Ada constructs. Typically, however, designers of generic environments have opted to partition the application on task boundaries. Partitioning on task boundaries tends to produce a correspondence between Ada's unit of concurrency, the task, and the underlying system's unit of concurrency, the processor.

Vendors of this type of distributed system often introduce additional "proprietary" features that optionally provide the user with explicit control over how and where tasks are distributed. For example, the Encore ARTE (Ada Real Time Executive) provides a runtime call to anchor a task to a particular processor. This feature is useful for tasks that may require specific hardware found only on one particular processor. Application developers can still develop a truly generic, portable application by choosing not to utilize these additional proprietary supports. If the runtime is not supplied with any proprietary information, the mechanics of task distribution transparent to the user.

PROPRIETARY ADA ENVIRONMENTS

A proprietary Ada environment uses operating system functions for real-time support instead of the aforementioned generic environment. Proprietary environments usually provide extensive support of the functionality of the underlying operating system as well as the required standard Ada language constructs. Typically, support for the proprietary features of the operating system are accessible to the application through interface procedures provided by the Ada runtime. These runtime interface procedures often do little more than call an underlying operating system function to perform the required task.

Device I/O

Device I/O is controlled by driver programs in the operating system. I/O operations can be executed either by making the appropriate Ada runtime call, which in turn calls the operating system, or by calling the operating system directly, if it is directly accessible from the application. While this approach helps to minimize the size of the Ada runtime, it tends to generalize the specific needs of Ada I/O, and consequently, may not be as efficient as it could be if it were dedicated to Ada I/O operations.

Interrupt Handling

In real-time systems, interrupts provide the asynchronous interface from the outside world. The interrupt mechanism of target computers will vary widely from vendor to vendor. A vendor will offer a software interface that is designed with the particular target hardware in mind. In addition to hardware variations, different vendors will have different market requirements, which leads to different offerings of functionality. The scope of this discussion will be constrained to the ability for a section of user code to gain control of execution when an interrupt arrives, and then to resume execution back at the point of the interrupt.

The user must specify, usually at run time, the location of the code that is to do the interrupt processing. One possibility is to specify a procedure or function that is to be executed when an interrupt arrives. This can done using the Ada `procedure'address` syntax, by specifying the symbolic name of the procedure in a runtime call, or by some set of commands to the compilation system. When the interrupt occurs, the procedure is entered for interrupt processing. When the procedure exits, then the interrupted task resumes.

Another possibility is for a particular process to be scheduled when an interrupt arrives. In this scenario, a separate Ada program is developed for the purpose of receiving the interrupt. The program starts and then notifies the underlying proprietary support system that it is ready to take the interrupt, causing the process to suspend. When the interrupt arrives, it is then resumed at that point, processes the interrupt, and schedules the interrupted process to run while it suspends. While the second approach is still a proprietary approach, it is more like the generic Ada philosophy of interrupts. It isolates the entity that will handle the interrupt from the entity that is executing when the interrupt occurs.

The major advantage to the proprietary use of interrupts is the availability of special features not common to all target systems. The underlying system may offer particular actions on an interrupt (such as starting the next I/O operation). The user may be able write his own handler for extremely fast interrupt processing. If the interrupt handler is developed in assembly language, then the interrupt can be processed without even doing a complete context switch.

Exception Handling

The needs of exception (error condition) support vary widely depending on the application program. Some real-time systems have very stringent exception handling demands, while other systems may not consider exceptions to be critical.

Error conditions peculiar to the target hardware may have proprietary support that is very helpful to some systems. For example, there may be an extremely time critical period in which data is captured from the outside world. A very small portion of that data may be corrupted with noise, which could cause arithmetic exceptions during the data capture. The use of generic Ada exception handlers may require too much execution overhead to be ready for the next element of data. Typically, the application will need only to note that the exception occurred, but not take the time to process it. Many proprietary real-time systems offer this capability, even though this capability is not standard Ada.

The most time-critical real-time systems often have very stringent time requirements for exception handling, yet very simple functionality requirements (as in the preceding example). Yet the generic Ada exception mechanism tends to be just the opposite, even in high-performance Ada systems on the market today: the exception handling usually is time consuming, but offers extensive capabilities for the handling and propagating of the exception.

With proprietary error handling, the actions available to the user for the occurrence of an exception are usually very limited. At best, the equivalent of an interrupt handler can be established for the exception. Users choosing proprietary environments need to ensure that the target system supports the exception handling needs of the application, particularly if Ada design constraints are in place.

Concurrency

Concurrency can be provided through one or more of several different methods, depending on the particular Ada environment that is being used. The standard Ada unit of concurrency (the task) is supported along with whatever means the underlying operating system employs to activate independent units of concurrency (normally referred to as tasks or processes). Once again, the runtime mechanics to activate an Ada task may consist of little more than a call to a process activation routine in the operating system.

Multi-Processing

In a proprietary environment, parallel execution of tasks is usually done using multiple processes executing multiple Ada programs. The application is divided into several different functional units which may proceed independently of each other. These units are then assigned to a particular processor in the system.

An advantage to this approach is that already existing non-Ada programs can be easily incorporated into the application. Supporting systems often provide a shared memory facility between different processes in the system. If an existing application is being ported to Ada, this approach allows one functional piece of code to be ported at a time.

Another advantage is that process scheduling facilities may be available that are not directly offered by the Ada language.

A disadvantage to choosing the proprietary approach over the generic approach is that the division of functional units among the processors is not abstracted from the logic of the application. Moving a component from one processor to another may prove to be a redesign effort. Another loss is the checking of shared data by the compiler. If all tasks are in the same program using the generic approach (distributed over multiple processors), then the correct use of shared data (data typing) is checked by the compiler.

COMPARATIVE ANALYSIS

Speed

Execution speed of the target application on generic and proprietary environments is dependent primarily on the areas of Ada functionality most frequently stressed in the application. Error (exception) handling tends to be faster and more versatile under a proprietary environment. Using a proprietary approach, the complicated, nested methods of handling standard Ada exceptions can be bypassed at the operating system level by time-critical applications that need to process frequent error conditions at a high rate of speed. Similarly, interrupt handling, as discussed earlier, is also faster and more versatile under a proprietary implementation.

A generic approach will generally out-perform its proprietary counterpart for systems utilizing a large number of concurrent tasks. Since task scheduling mechanics for a generic system are contained entirely in the runtime, optimizations can be made to this code without adversely affecting other system software (such as an operating system). In most generically oriented systems currently on the market, the vendors have implemented major optimizations. Table 1 is a comparison of two Encore Ada development environments. The first is proprietary, executing on top of the MPX-32 operating system. The second is the Ada Real Time Executive (ARTE), a generic environment. The compilers (and hence, the code that is generated) for each environment is identical. The difference in the load modules is determined by the runtime link.

Test Description	Task Location	Avg. Time (MPX)	Avg. Time (ARTE)	PIWG Test Name	% DIFF (ARTE-MPX)
One Task One entry	In a Procedure	234.4	140.0	T1, T2	-40.1%
One Task Two Entries (in a Select)	In Main	386.5	277.3	T4, T6	-28.3%
Two Tasks One Entry Each	In a Package	232.8	145.1	T3, T5	-37.7%
One Task One Entry	In Main	157.6	89.2	T7	-43.4%
Two Tasks One param. passed by rend.	In Main	625.0	526.1	T8	-15.8%

Table1. Standard PIWG execution results for MPX -32Ada (a proprietary environment) and ARTE (a generic environment).

Notice that rendezvous times are significantly faster on a generic, optimized implementation of task scheduling.

If application speed is vital to project success, then the problem domain of the target application should be evaluated carefully before selecting a particular approach. Applications containing a high volume of concurrent tasks will tend to perform better under a generic environment, while programs that rely heavily on interrupt handling capabilities or must handle a large number of error conditions as part of its primary control flow will probably perform better under a proprietary environment.

Efficiency

The Ada language is a more complex language than most languages used in training systems. When making a transition to Ada, the user must be aware that in some cases more resources will be required in the system. For example, more memory is typically required for Ada. One reason is that there are "automatic" data structures internal to the language that will be allocated to track the elaboration of each package used in the system. This additional memory is not required by Fortran or C.

There are other parts of Ada that require more resources because of the added functionality of the language. The runtime library will necessarily be larger due to the fact that it must support the operating-system-like features of the language. A bare machine Ada approach helps to alleviate this problem by replacing the operating system with the Ada runtime, freeing up memory that would be required by the operating system.

Although Ada is often seen as a language that requires more computer resources for everything, that is not always the case. The Ada language can be compiled into very efficient machine code. Because of the richness of the language, programmers have the opportunity to give a large amount of information to the compiler, such as constraints of variables, whether or not parameters and variables will be modified, and so on. Good compilers and optimizers can use this information to produce better quality code than can be produced from less detailed high order languages.

If memory requirements on the target system is an issue, then the developer should look for an Ada compilation system which only includes routines that are actually called in the program to be loaded. This would allow the user to eliminate the tasking part of the runtime from programs which do not use any Ada tasking constructs, for example. When proprietary alternatives are chosen rather than the generic Ada counterparts, then it is desirable that the support for the Ada that is not used is not loaded with the final program.

The efficiency issues of proprietary real-time support versus Ada are difficult to quantify because the efficiency varies so much over different vendors. Some proprietary subsystems may be more efficient because they do not support the complex structures of Ada (*in exception handling, for example*). *Yet other* proprietary subsystems may be less efficient because they must support a wide variety of needs for different languages (in I/O, for example).

Portability

Application portability is the ability to move application source code from one Ada system to another, recompile, relink, and execute the resulting application with a minimum of effort. Portability can be achieved using either a generic or proprietary approach simply by restricting application source code to use only standardized Ada constructs. A generic environment is most likely to ensure Ada portability, since a generic environment supports only standard Ada structures, which are standardized across different platforms.

Portability should not be an issue when deciding on a particular approach unless there is a possibility that the application will one day be ported to a different target system. If there is a reasonable certainty that the application, throughout its life cycle, will execute on only a single or restricted number of targets, then the *additional functionality of a proprietary operating* system may prove to be a superior choice. However, the portability of Ada systems is one of the most desirable attributes.

Determinism

The degree of deterministic behavior of a real-time system is not easily decided by whether it uses a generic or proprietary approach. There are many factors to be considered that are beyond the scope of this paper. However, there are a few items that should be considered if the determinism of the target system is important.

In a generic Ada environment, a priority preemptive scheduler is usually a requirement. In addition, features such as check suppresses, rendezvous times (optimal and typical), maximum interrupt blocking times, method of exception processing, etc. must be considered. These are some of the critical elements needed to build a deterministic system.

In a proprietary environment, the determinism of the system depends on the functions supported by the underlying system and how they behave in real-time. Space does not permit their discussion here.

Software Training

When the transition is made to Ada from some other language, training of the software staff will normally be required. Obviously, rudimentary training in the Ada language will be necessary. In addition, training for the real-time aspects of the target system is required. If the transition is being made on the same target platform, then the software development team may already be familiar with the proprietary real-time features of the target system. If the proprietary approach is chosen, then extensive training on Ada real time systems is not required.

Taking advantage of the proprietary expertise already developed will pay off in the short term, but may not prove to be worthwhile in the long term. This benefit must be weighed against the cost in the areas of life cycle costs.

If the development team is not already familiar with the target proprietary system, then the generic Ada approach will probably require less training. Since the real-time features are integrated into the language, it is easier to understand how to use them once the language is understood. Unfortunately, there is not much training available for real-time Ada systems today. To make matters worse, there are not many engineers available in the industry who understand the real-time Ada issues. As Ada vendor products mature, more systems will be developed using them, which will cause the experience in the field to grow. In the 1990's, the demand for quality real-time Ada training will grow and more training will become available.

Maintainability

The primary reason for the use of Ada in real-time systems is the reduction in life-cycle costs. Proprietary real-time interfaces are not standardized, even across different target systems from the same vendor. Neither is the interface guaranteed across different revisions of the same system. If a target system is upgraded or changed to be a system from a different vendor, then all of the code that uses the old interfaces will have to be modified. Even worse, any subsystem in the application that depended on some functionality of the original system that is not present in the new system may have to be completely redesigned. The proprietary approach to these systems partially defeats the maintainability advantages of Ada and can lead to very high life-cycle costs.

When using the generic Ada features, the maintenance costs are reduced in several ways. First, if the system is upgraded, the new system, even if from a different vendor, will have the same Ada interface for real-time features. Note that this will not be completely compatibility because the language standard defers some features to be implementation dependent. Secondly, if a development staff turnaround occurs, it is more likely that new staff members with Ada experience will have generic Ada familiarity than to have familiarity with a particular target proprietary system.

Convertibility

The term "convertibility" refers to the ease of which applications written in a different language can be converted to Ada using the given approach. This is only a consideration for applications that are being converted, not for new applications being developed.

At first thought, it would seem that the proprietary approach lends itself to greater convertibility. Naturally, this would only be true if the underlying proprietary support were available on the new target system. This may often not be the case. Another prerequisite is that the Ada run-time environment provides adequate access to the proprietary interfaces. This assumption is frequently made, but may often not be a valid assumption. Many Ada vendors actually purchase their Ada product from a third party, which may not have an interest in the proprietary interfaces.

Even if none of these considerations pose a problem, the user has another decision in making the transition to Ada. Using the proprietary approach may lead to a much faster transition initially, but this benefit may be lost over time as life cycle costs remain high. The user should consider going back to the design of the existing application and attempting to accurately capture the original design in Ada code. Not only does this capture the benefits of the generic approach already mentioned, but it also leads to a cleaner implementation, avoiding the "Ada-tran" syndrome.

A drawback to the generic approach is that it will tend to yield an Ada program that is not easily converted to a different language, since the Ada constructs are generally more flexible than traditional proprietary support systems.

AN ALTERNATIVE SOLUTION

It would be incomplete not to mention an effort underway to alleviate some problems discussed so far. The Ada Run-Time Environment Working Group (ARTEWG) of the SigAda of the Association for Computing Machinery has proposed a solution to having to make the tradeoffs discussed in this paper. The solution is to provide a set of interfaces to the underlying system which are not part of the Ada language, but implemented as procedure calls using the standard Ada package interface. The key is to standardize this set of interfaces to preserve some of the advantages of the generic approach that would otherwise be lost when using the proprietary approach, such as portability.

The ARTEWG has published a "Catalog of Interface Features and Options for the Ada Runtime Environment" (1) which contains a standardized specification of support routines needed in a real-time system. The purpose of the document is to provide users with a quick way to implement portable, maintainable, real-time Ada systems without having to confront some of the problems with and restrictions of the language (mentioned above) and without having to use an extended Ada. The specification defines interfaces for such support as cyclic scheduling, "fast" interrupt handlers, dynamic task priorities, and so on, none of which are defined by the Ada language.

Although the proposal is somewhat controversial, it is nevertheless a good idea. The language itself must be very well defined and regulated. As a result, issues specific to real-time systems are not standardized by the language, because the language serves other realms of computing as well. A separate standard should be adopted to standardize support needed only in real-time Ada systems.

CONCLUSIONS

This paper is a guideline to the real-time training systems developer. The paper not only answers some questions, but also presents which questions need to be asked by the implementor to the vendor. These questions are guidelines and are not a comprehensive list of things that need to be considered by the implementor.

The Ada language has made great progress in the maturing process, but maturation is not yet complete in the industry. It seems that the real-time features of the language are among the most immature. This is probably caused by the fact that they are not required for validation, while most other features are required for validation. Thus the vast majority of the resources of Ada vendors has been concentrated on issues other than the real-time features. Presently, compilation systems are maturing, which shifts more of the industry activity to areas such as real-time support and runtime functionality. (Other areas of rapid growth in the Ada industry included tool development and refining of standards.)

When implementors of real-time systems are selecting an Ada environment, they must be careful to determine all the runtime needs for the system. Because of the newness of Ada technology, there is still quite a bit of confusion about the terminology, especially in the marketing literature. A good way to accurately determine if the support needed is available in a particular system is to provide functional benchmarks to the vendors.

The generic approach is in general the favored choice for real-time Ada systems in the 1990's. While there are certainly tradeoffs involved, most drawbacks mentioned in this paper will be overcome by the continuing capacity enhancements of target systems. Other factors such as larger memory systems, better design and debugging tools, and better training in real-time Ada design will help insert Ada technology into the simulation and training industry.

In the near future, the proprietary approach will continue to be popular. There are certain niches, especially in smaller systems, where the advantages of the generic Ada approach are just not significant, while tradeoffs may prove to be quite costly. In such cases the proprietary approach is the best choice.

REFERENCES

(1) Ada Runtime Environment Working Group, "A Catalog of Interface Features and Options for the Ada Runtime Environment." Release 2.0, December, 1987.

ABOUT THE AUTHORS

Both Don Law and Gary Croucher are senior members of the technical staff at Encore Computer Corporation in Fort Lauderdale, Florida. They have been working on the Ada Real-Time Executive project for the past two years. Both authors hold a Bachelor of Science in Computer Science and Mathematics from Furman University.

Mr. Croucher is the project leader for the ARTE project. He holds an MS in Computer and Information Sciences from the University of Florida, College of Engineering. He has worked at Encore for five years and with Apple Computers for two years prior to that.

Don Law has been working at Encore since graduating from college five years ago. He has worked on several Ada projects including a prototype of the Common APSE Interface Set (CAIS). He has been a contributing member of the BMA project team since its onset three years ago. He is currently the project leader of the Ada compiler and debugger group at Encore.