AIAA 79-1988R

Flight Languages: Ada vs HAL/S

Bruce Knobe*
Intermetrics, Inc., Cambridge, Mass.

HAL/S is a language designed and implemented in the early 1970's to support the development of flight software for the Space Shuttle. It has subsequently been adopted as a NASA standard language for flight applications. Ada is a language designed for the Department of Defense's (DoD) Advanced Research Projects Agency. It is hoped that Ada will eventually become the DoD standard language for embedded software. Both languages provide excellent support for the development of typical aerospace applications. We consider how the differences in perspective, special purpose vs general purpose, and the 10-year difference in time period has affected the language designs.

Introduction

HAL/S is a language developed for NASA to support flight applications; Ada is a language developed for the Department of Defense (DoD) to support embedded applications. Since flight programs are most certainly typical of a large class of embedded algorithms, it is interesting to contrast the approaches taken by these two languages.

Given the similarity in overall objectives, one would expect to find substantial commonality in the two languages. In fact, it is remarkable that two languages could have meaningful differences on so many points. HAL/S is a conservative design of 1969-1970 vintage; Ada is a research level design 10 years later. HAL/S starts out as a real-time algebraic language and then adds a minimal amount of system programming capability. Ada is a system programming language with a minimal amount of numerical and real-time support. HAL/S imposes restrictions to enhance compile time error detection and run time efficiency, while Ada maximizes the language's expressive power and flexibility.

Perhaps the greatest difference, however, lies in their attitude about the value of the pragmatic vs the theoretical viewpoint. HAL/S specifically addresses the problems inherent in writing flight programs. It provides a pragmatic tool for dealing with a large but bounded collection of software problems and development methodologies. Ada attempts to define a language aesthetic and combine this with theoretical concepts in language design to provide a much more general capability. It relies on the user to customize his usage for the particular application area. Writing in HAL/S, some approaches are much simpler to implement than others. Writing in Ada, all things are equally easy (or equally difficult).

When comparing languages, there is a natural tendency to make value judgments. The actual design objectives of these two languages are, in fact, sufficiently dissimilar that such an assessment can only be made in the context of a particular project. Thus, an attempt has been made to describe associated language features and discuss their strengths and weaknesses, without making a choice as to which is superior.

It should be understood that both of these languages are quite large. The Preliminary Ada Reference Manual[1] contains 131 very technical pages which really must be read in the context of the 248 page Rationale for the Design of the Ada Programming Language.[2] The HAL/S Language Specification[3] requires 228 pages to achieve a level of explanation intermediate between the Ada reference manual and the combined Ada documents. These sizes can be compared with the 32 pages required by the Pascal Report[4] or the 44 pages required by the very legalistic draft Pascal standard.[5]

This paper does not attempt to treat these two languages in any depth. Only those areas with some particular interest to flight applications will be discussed.

Lexical Considerations

In most languages, lexical considerations are fairly mundane. Even here, however, we find substantive differences in approach. Ada takes a fairly standard attitude. The source text is a linear stream of ASCII characters and, following the Algol-60 tradition, there are very few reserved words.

HAL/S programs are also expressible as a linear stream of ASCII characters, but a standard two-dimensional notation is defined and HAL/S compilers generate two-dimensional listings regardless of the source format. Thus, a HAL/S listing would appear as:

$$A_I = SQRT(B_I^2 + C_I^2)$$

as opposed to the Ada

```
A(I): = SQRT(B(I)**2 + C(I)**2)
```

Programmers rarely write programs in the two-dimensional notation and, consequently, some of them complain that there are difficulties in relating the listing to the source text. Readers of HAL/S programs are unanimous in their acceptance of two-dimensional listings.

HAL/S reserves all keywords in the language. The Ada programmer can redefine the predefined type name INTEGER—not so in HAL/S. In general, the HAL/S reader is somewhat more secure that he understands what he is reading. The HAL/S writer, on the other hand, must deal with a rather long list of reserved words which include some very natural variable names (e.g., LINE).

Types

Types is one of the two really big areas of disagreement between Ada and HAL/S, the other being real-time support.

Traditionally, languages have provided some collection of built-in types and a mechanism for building aggregates from those objects, the two aggregates being arrays from FORTRAN and records from COBOL. Somewhat more modern languages (e.g., Algol-68 and Pascal) recognized that

Presented as Paper 79-1988 at the AIAA/IEEE/ACM/NASA 2nd Computers in Aerospace Conference, Los Angeles, Calif., Oct. 22-24, 1979; submitted Nov. 15, 1979; revision received June 26, 1980.

*Manager, Programming Languages and Compiler Department, Compiler and Support Software Division.

these aggregates are essentially new programmer-defined types and have provided explicit capabilities for the programmer to define and use his own types. Yet more advanced languages (e.g., CLU and ALPHARD) have identified programmer-defined types as *the* essential ingredient in language design and have introduced the notion of abstract data types. HAL/S and Ada lie at opposite extremes of the spectrum on this issue.

HAL/S takes the traditional viewpoint that an application area requires some small number of data types. The language has all those data types built in and offers exceptional support for them. Ada takes the viewpoint that each program requires its own data types and therefore provides a low level of support to a smaller number of data types, while providing a very substantial facility for programmers to define their own data types. One could argue that combining the two approaches would yield the best of both worlds, but both designs wisely realize that such an approach would yield an unwieldy mess.

Built-in Types

The built-in types in Ada are integer, real, fixed, boolean, and character. I would prefer not to consider boolean or character as predefined enumeration types in this discussion. The programmer can build arrays and records of these objects, but there is little built-in support for doing anything with these objects. In addition to the Ada types, HAL/S contains bit strings, varying length character strings, matrices, and vectors. An Ada programmer can build all of these objects, but the built-in approach offers several advantages:

1) Standardization—all HAL/S programmers use these things the same way.

2) Documentation—the compiler recognizes uses of different kinds of data and annotates the listing appropriately (e.g., bars are printed over all vectors).

3) Efficiency—the compiler can generate optimal code for these constructs because it knows about them specifically. This is increasingly important on vector architectures.

4) Cost—the implementation must be done once per compiler rather than once per project.

5) Convenience—the basic data structures are there, one does not have to build them.

HAL/S also provides unusual support for arrays. Any place an object can be used in HAL/S, an array of such objects can be used instead. This eliminates a large number of one-line loops and enhances efficiency and readability.

With respect to types, HAL/S can be viewed as the logical conclusion of the traditional line of development customized for flight applications. There were no serious problems with the language definition since there was a very solid base on which to build. The types provided have proved adequate because the application area is well understood. The flight software for the Space Shuttle, perhaps the largest flight program attempted to date, is coded in HAL/S without running up against limited data types.

The argument against built-in data types is, of course, inflexibility. How about complex numbers? How about sparse matrices?

There are three major components to the Ada-type mechanism: 1) the ability to define types, 2) the data abstraction mechanism, and 3) overloading.

Programmer-Defined Types

The basic type mechanism in Ada is patterned after Pascal. The Pascal system works well for the problem area it addresses but has several important problems which were excellently analyzed by Haberman.[6] These problems are of substantial import when dealing with large programs built by hundreds of programmers. The Ada designers understood and addressed all of these problems, including providing a definition of type equivalence, an issue not addressed by the Pascal report. The penalty they paid was replacing the very simple type mechanism of Pascal with a much more complicated facility involving types, derived types, and subtypes. In order to limit the burden placed on the Ada writer, context rules are used as part of the type determination algorithm.

Simply having a way to define types does not necessarily buy anything. The Ada facility immediately provides the programmer increased security. The enumeration type provides more security than named integer constants and the ability to define different types with the same underlying properties extends the compiler's ability to detect errors using the type mechanism.

Data Abstraction

Once a type defining mechanism exists, it is possible to define the type MATRIX, for example, and a collection of operators on MATRIXes. This is possible in simple Pascal. The problem is that Pascal has no way to package the type and operators. Ada provides a PACKAGE which can encapsulate the definition of the type and its operators. The PACKAGE can control the amount of information exported about a type or a variable by defining and declaring identifiers in any of three different places. The PACKAGE facility does not provide any new power. It provides added security (and perhaps provability) by drawing a syntactic wall around a collection of identifier definitions, thus limiting access to these identifiers.

Overloading

Languages limited to built-in types typically define a collection of overloaded operators, that is, a collection of operators defined for several different kinds of operands. In HAL/S, the multiply operator is defined for:

integer	integer
integer	real
real	integer
real	vector
vector	real
⋮	

The list is almost endless (and gets four times as long if we consider single- and double-precision operands). In fact, a programmer in such a language does not worry about these details. He knows the properties of multiplication, that HAL/S provides automatic coercion between real and integer, and that HAL/S provides all the natural arithmetic operations on integers, reals, matrices, and vectors. The Ada programmer building a matrix/vector package would like to be able to provide the same kind of naturalness. This means that Ada must allow the programmer to write definitions of function and operator symbols. The specific definition applicable in a particular context is determined from the type and number of operands, or type of result desired in a particular context. This facility seems sufficiently powerful to define the entire built-in type and operator structure of HAL/S—a formidable task indeed. Ironically, however, the facility for overloading weakens the type-checking structure at its most useful point, checking across procedure calls. This weakness derives from the fact that a procedure call made with incorrectly typed operands may not be detected as an error. Rather, a procedure completely different from the intended one may be invoked.

The advantage of the advanced-type facility of Ada is quite clear—flexibility. Ada enables the expert to build layers of a virtual machine, custom crafted to his particular needs. The disadvantages are more worries than facts. The Ada-type facility is very much at the research level. There is no body of experience using this kind of facility. The Ada facility is quite complicated. Early statements by the Ada design team indicated that type resolution might require six passes; in fact, it

requires no more than three. Certainly a human reader can have great difficulty interpreting some Ada programs. Only experience will show whether the potential problems are real; whether Ada is appropriate to projects employing hundreds of relatively unsophisticated programmers and whether acceptably efficient implementations can be achieved. Ten years ago, two major areas of programming language research were dying down. The proponents of structured control structures had won a clear victory and all subsequent languages benefited from the research. The proponents of syntactic extension had failed to convince the community of the practicality and implementability of their ideas, and today one hears very little about what was once the great hope of advanced language design. Data abstractions appear to be a prime tool supporting the most popular program development methodologies, but only experience will tell us whether these ideas are useful in practice.

Storage Allocation

Automatic Allocation

Ever since Algol-60, programming languages have allowed programmers to declare variables which are *automatically* allocated on procedure entry and automatically deallocated on procedure exit. Both HAL/S and Ada use implicit stacks to support automatic allocation. Once a stack discipline has been adopted, it can, in principle, support recursion and varying length objects with little extra work. These facilities have substantial impact on security in a flight environment, where it is intolerable to run out of space at execution time.

Storage allocation poses some interesting problems in a real-time multitasking environment. The obvious way to maintain a stack is to simply give it all the available space and hope for the best. In a multitasking environment, each task must have its own stack. Furthermore, in order to enable tasks to scope in data from their parents, each stack must point back to its parent's stack. Implementing these "cactus stacks" on classical linear memory architecture requires either assigning a fixed amount of space for each task or making the procedure entry and exit code very inefficient. Memory architectures with mapping hardware can support cactus stacks by giving each task its own map and performing a remap of memory on each task swap.

Fixed Sizes vs Variable Sizes

Storage allocation is a central concern in the HAL/S design. A clear decision was made that all storage requirements be known at load time. This decision forces the following language restrictions: 1) recursion is disallowed; 2) the sizes of all objects must be known at compile time; and 3) the number of tasks must be known at load time.

The gains that accrue from this decision are: 1) a program can never run out of storage in flight; and 2) the amount of space required by each task is computable and, consequently, the correct amount of space for each stack can be allocated at load time. This enables efficient entry/exit code and also supports telemetry processors and other programs which must asynchronously probe the state of the machine.

Ada makes none of these restrictions. Presumably, flight programs will take precautions to prevent running out of space in flight. Those precautions might entail adopting the HAL/S restrictions or might depend upon an analysis that computes upper bounds on the space required, the number of tasks, etc. This is one more example of the same difference of philosophy. HAL/S builds in a strategy guaranteed to work but limiting the programmer's alternatives. Ada provides greater flexibility but requires the programmer to provide his own guarantee. The HAL/S restrictions allow the building of a reasonably simple linker to allocate space. The Ada facility requires sophisticated optimization in either the compiler or some subsequent integration tool to detect those cases where simple, efficient stack manipulation procedures can replace the less efficient general case algorithms.

There are two other kinds of storage allocation typically supplied by programming languages: 1) dynamic or user-controlled allocation, and 2) static or compile time allocation.

Dynamic Allocation

Dynamically acquired data is invaluable when dealing with complex graph structures. It is an important facility in most system programming languages. Dynamic allocation implies that the required amount of storage is not known at load time and consequently is not supported by HAL/S. Ada does include dynamic allocation and furthermore includes it in a form supported by garbage collection to recover space after it has been used. Whether such a facility can practically be used in a flight environment is still an open question. It seems very difficult to avoid using dynamic allocation in Ada as pointer variables can point only to dynamic variables.

Static Allocation

Static storage allocation provides increased efficiency on some computer architectures. It enables telemetry processors and debuggers to access data easily and efficiently without any support from the running programs. HAL/S supports static allocation, both in separately access-controlled global blocks and also at the individual procedure's scope level. Ada does not really provide static storage, although the outermost scope of data visible to all procedures could be allocated and initialized statically.

Expressions

Both languages support fairly typical expressions. Because of the much larger number of built-in types, HAL/S provides many more operators than Ada.

Control Flow

The important debates in the area of control flow have been settled for a long time now. Both languages have IF statements, RETURN statements, CASE statements, EXIT statements, and various flavors of loops. Both languages have GOTOs and it is unlikely that anybody will ever need them in either language. The HAL/S statements provide somewhat more power than the Ada facility at the price of a slightly messier syntax.

Procedures

Both languages provide procedures and functions with all the usual properties. The parameter passing mechanisms are significantly different, however, and deserve some consideration.

HAL/S has INPUT parameters and ASSIGN parameters. INPUT parameters cannot be modified by the subroutine while ASSIGN parameters are expected to be modified. ASSIGN parameters must be passed by reference while INPUT parameters may be passed either by reference or by copy at the compiler's option. The one nicety in HAL/S is that the programmer is required to distinguish ASSIGN parameters both at the point of call and at the routine's declaration. Thus, if a procedure is changed so that it now modifies one of its arguments, all callers must be notified of the change.

Ada provides IN parameters, OUT parameters, and IN OUT parameters. The original design documents made it clear that parameter passing would be implemented by copying. This strategy would have required that large objects be dynamically allocated and then explicit pointers to the objects be passed rather than the objects themselves. Thus, few, if any, programs would have been able to avoid substantial use of pointers and dynamic storage. There has been considerable comment about this philosophy and it appears that the language does not *require* such an implementation.

A rather unique feature of Ada is its ability to name the parameters in the call. Thus,

ACTIVATE(PROCESS: = X, AFTER: = NOPROCESS)

is a perfectly legal procedure call. Furthermore, it is possible to omit parameters in the call when the procedure specifies defaults. This latter facility of default parameters seems very convenient when calling procedures with long calling sequences, but incurs the price of further weakening the type checking facility across procedure calls. An accidentally omitted parameter may well go unnoticed by the compiler. This represents a line of thought consistent with the overloading approach which provides very nice expressive capability at the price of slightly weakening the type checks across procedure calls. When using this notation, the Ada programmer is required to specify IN, OUT, or IN OUT, at the point of call, by writing :=, =:, or :=:. Thus, it is possible for the Ada programmer to achieve additional security against one kind of change in called procedures by using the optional, more verbose procedure-call syntax.

Modules and Separate Compilation

Both of these languages are intended for supporting large programming projects, and consequently support separate compilation with full type checking.

The HAL/S system is quite simple, patterned after Jovial compools and FORTRAN-named common blocks. Any name (other than that of a procedure) which is to be visible to more than one compilation unit must be put into a COMPOOL. All COMPOOLs and all separately compiled program units are potentially visible to all compilation units. In practice, any given compilation unit only INCLUDEs a small number of other units and HAL/S compilers provide an extralingual facility to allow project management to restrict access to separately compiled units or to particular names in COMPOOLs. Although the COMPOOL facility allows a programmer to build a pool of data accessed by some limited number of routines, the language provides no syntactic encapsulation around these units and no syntactic mechanism for building hierarchies of such units. The experience in the Space Shuttle program has been that data which logically need not appear in a global COMPOOL often migrates there anyhow to provide convenient access for the telemetry processors. Thus, even when the data could be moved further into the program hierarchy, it has wound up at the most global level.

The HAL/S COMPOOL facility is clearly something added on to the language to support separate compilation. The Ada approach is much more integrated. An Ada program is a single hierarchically nested object. The separate compilation facility completely supports this notion. That means that a deeply nested procedure can be compiled separately and still inherit the same scoped in names it would have had if it had been compiled in place. The key to this facility is the PACKAGE system described earlier. The PACKAGE system allows declaration of data and procedure interfaces separately from the procedure body. The system also includes facilities for explicitly importing names so that a deeply nested procedure is not flooded with irrelevant names. The Ada system has some important advantages. It is integrated with the rest of the language and can be used both in separate compilations and monolithic compilations. It provides very substantial support for modular programming. The system also has some disadvantages. It assumes that all data relationships are hierarchical; that is, it requires that any data shared by A and B be in a scope containing both A and B. This scope may well contain many other procedures that want to share data with one or both of A and B but which should not see this shared data. Although the hierarchical access to data provides a good match to most classical applications programs, it may prove to be something of a straitjacket to impose this relationship 100% uniformly on very large real-time programs. The net effect may be to promote all interesting data to the most global level. As with many of the advanced features of Ada, the separate compilation facility is quite complicated. Whether programmers will learn to use it effectively must be seen. It is not nearly as simple as writing down nested declarations in Algol-60 or Pascal.

Multitasking and Real Time

The difference in the real-time facilities of these two languages is truly amazing. They seem to contain no common point of reference whatsoever.

The HAL/S system, as usual, picks out a particular world view and provides very substantial support for it. The Ada system, equally characteristically, provides the tools to do it yourself. HAL/S has chosen a very definite real-time model and supports it to the hilt. If you do not like the model, you may have real trouble. Ada picks an extremely general solution. It is likely that this formulation has never been used in a flight program and probably has never been used in a real-time program. Certainly, many of the classical formulations can be built with the Ada facilities, but the programmer must do the building and face the issue of achieving acceptable levels of efficiency.

The principal capability provided by Ada is the notion of a rendezvous. A rendezvous is initiated by one task calling and another task ACCEPTing. An ACCEPT statement is much like a procedure, except that it appears embedded in the code of a task. When the task reaches the ACCEPT statement, it waits for a caller. Callers can be queued waiting to be ACCEPTed. A multiway wait may be programmed by embedding several ACCEPT statements in a SELECT statement which waits until one of its ACCEPTs is called and then executes it.

Both languages contain the concept of a task (or process). In each case there is a primeval process which is started in some undefined way and which then can start up other tasks on one or more machines which can start up yet others, etc. In both cases there is some underlying runtime system which actually starts tasks, operates queues, and so forth. To start up a task, an operating task must request that the task be enqueued. The scheduler in the runtime system decides which tasks will actually run at any given moment.

Scheduling

When an Ada program initiates a task, the task is enqueued immediately with the unchangeable priority specified with the task's declaration. If the desired start-up time for the task is some later time, the initiator must DELAY until the right time. If the task is to be run cyclically, the initiator must sit in a loop synchronizing with the subtask, DELAYing by appropriate amounts, and then enqueuing the task again. It seems quite likely that relatively simple cyclic processes like guidance, navigation, and control applications might have tasks whose sole function is to sit in a loop which enqueues the task that actually does the work.

The HAL/S SCHEDULE statement is at the complete opposite extreme. A task can be enqueued immediately, at some specified time, after some specified delay, or after some real-time event. The priority of the new task is specified in the statement. Whether the new task depends upon its initiator's continued existence can be specified. If the task is to be run cyclically, a repeat cycle length can be specified in absolute time or in time after the last cycle finished and conditions for terminating the cycle can be specified in absolute time or in terms of real-time events. It is quite clear that HAL/S has made very definite decisions about the way real-time programs should be organized. There are other incompatible models which are difficult or impossible to realize. On the other hand, the HAL/S statements allow the programmer to state in an easily read fashion a fairly complicated but very common scheduling philosophy. Furthermore, the compiler

and runtime system can optimize the scheduling to minimize the number of context swaps. Building this kind of scheduling out of the Ada primitives can be done fairly easily, but the resulting program will require many more calls on the runtime system and perhaps some additional tasks. Since the context swaps required to do this extra work are expensive on present hardware, there are serious efficiency issues.

Synchronization

Both systems allow a task to wait for a specified amount of elapsed time. The HAL/S wait statement also allows waiting until some specified time or for a real-time event. This latter function is performed by the Ada ACCEPT facility, the difference being that the ACCEPT facility forces a rendezvous. If the real-time event occurs before the wait happens, some task has to perform a call of an ACCEPT statement and then stall until the ACCEPT is executed. Once again, a trivial subtask will probably be spawned to perform this operation.

Interprocess Communication

Communication between tasks is a rather difficult issue in language design. There are two common solutions: passing explicit messages, and sharing global variables.

The Ada ACCEPT facility looks just like a procedure call; thus, explicit transfer of information between synchronized processes is very natural. If the message sender does not want to wait for the receiver, it can spawn a subtask to pass the message. HAL/S has an extremely primitive facility which allows a process to send a boolean signal. Other processes can test the value of these signals and/or include them in real-time statements such as SCHEDULE and WAIT.

Most languages which support real-time have a specific facility for guaranteeing exclusive access to shared data. For instance, several tasks might want to read the navigation data, but the navigation process must have exclusive access while it is updating the values. In HAL/S, this facility is provided through LOCK groups and UPDATE blocks. When a data item is declared, it can be put in a LOCK group. All access to locked data must be made in syntactically delimited UPDATE blocks. At the entrance to an UPDATE block, the compiler automatically locks the semaphores associated with the LOCK groups manipulated inside the block. At exit from the block, the semaphores are automatically unlocked. An optimization mechanism allows several tasks to read a LOCK group concurrently. The language rules limit UPDATE blocks in order to quarantee that tasks cannot deadlock over access to UPDATE blocks and that locked data can never be accessed outside of UPDATE blocks.

In the Ada rationale, the use of shared data is discouraged and the Ada language provides no special mechanism for dealing with this issue. The user can, of course, build his own. HAL/S UPDATE blocks can be trivially simulated by the use of semaphores, but an Ada compiler cannot guarantee that all accesses are properly protected by semaphores and care must be taken to avoid deadlock when several LOCK groups are accessed in the same block. A much simpler and safer way of dealing with shared data is to set up a task which contains the shared data and a set of ACCEPT statements to perform all the required operations on the data. Many of these ACCEPT statements may simply copy to and from the shared data. Nevertheless, each appears to require two task swaps. An interesting optimization[7] enables the elimination of the scheduling overhead of the task swaps, but substantial environment switching may still be required. Even after optimization, the code executes two unusually expensive procedure calls and some copying. Furthermore, conceptually simple operations may require a surprisingly large amount of code when performed on shared data. One can certainly build primitive monitors in Ada, although it seems unlikely that a guaranteed safe system could be built allowing a monitor to be re-entered while one of its actions is blocked. At best, any safe system will require the introduction of additional tasks and two apparent task swaps every time shared data must be accessed.

Talking About Time

One of the interesting differences between the two languages is their reverence for time. HAL/S includes time in several constructs and even provides a facility for dealing with time in implementations supporting only fixed point reals. Ada refers to time only in the DELAY statement and the CLOCK attribute (built-in function). There is, for instance, no *reliable* way to make something happen at some single specified future time or at some specified cyclical interval.

Exceptions

Both languages support exception (error) handling. In both cases, if a handler is not locally available, the search goes back to the caller rather than out to the statically enclosing scope. The Ada syntax is somewhat more readable than that of HAL/S, although the two are equal in semantic power with one difference. The Ada system specifically requires that once an exception has been detected, the execution of the enclosing program unit must be replaced by a handler. There is no way to ignore an error or to fix up an error. Based on the horrors perpetrated by PL/I ON conditions, this seems like a very good principle. On the other hand, there do seem to be some places where fixups are reasonable. On floating point underflow, for instance, many situations simply call for setting the result to zero. When values get out of the expected range, but are still in a computationally acceptable range, it is nice to produce a warning message and continue rather than abort the process. The HAL/S language allows fixups only in those cases where they are defined by the implementation. An individual program cannot perform fixups, but it can signal the error to a separate task and then continue.

Macros

The area of macro definitions presents an interesting reversal of roles. The HAL/S facility provides a general parameterized macro capability, while the Ada facility attempts to limit the kinds of macros which can be written. The importance of this kind of facility should not be underestimated. The Space Shuttle code, for instance, contains over 32,000 characters of macro definition.

Ada has two macro style features. RENAME allows the programmer to provide a second name for an already declared variable. The GENERIC facility is much more sophisticated. GENERICs allow the programmer to build prototypes of program units and then instantiate multiple copies of this unit with different values of the parameters. For instance, a PACKAGE could be built to manipulate stacks of type T, where T is a parameter. Using this GENERIC definition, actual PACKAGEs can be instantiated for stacks of integers, stacks of reals, stacks of parser tokens, etc. There are various limitations on GENERICs to make them compilable and reliable. Finally, GENERICs follow the Ada scope rules, so imported names use the declarations available at the point of GENERIC definition rather than those at the point of GENERIC instantiation.

The HAL/S facility provides simple parameterized macros. Macro expansion is a simple textual substitution process with no limitation on what can be generated. For instance, three different macros might be written to obtain the same value: 1) call a function; 2) compute the function directly; and 3) find the value in a precomputed table.

The user of such a macro has no reason to know how the value is computed when he writes the invocation, and the object code he gets will be identical to the code he would have gotten if he expanded the macro himself.

Input/Output

Both languages have facilities for generating formatted I/O, but neither of these facilities is aimed at flight systems. Flight system I/O must be performed by project-dependent library routines.

Built-in Libraries

HAL/S defines a large collection of libraries (e.g., SIN, SINH, ARCSIN). These names are reserved and the functions can be used in compile-time computations. The advantage of such a library is that it promotes standardization, improves cost sharing between multiple projects using the same compiler, and provides more readable constant definitions. The disadvantage is that it ties up a lot of names and puts a substantial extra load on the compiler writer. The Ada requirements document[8] expressly disallowed such libraries.

It should be noted that NASA's configuration control philosophy for HAL/S expressly recognizes subsets of the language and defines a "valid subset." Although the subject is currently being debated, the Ada requirements document states that subsets will not be allowed. This difference in configuration control philosophy may well explain the differences in attitudes about language-defined libraries.

Conclusion

Ada and HAL/S represent dramatically different approaches to a problem. The basic philosophical difference is the viewpoint on generality. Ada must support a very large and varied community; therefore, it has opted to maximize generality. HAL/S must serve a much smaller community and consequently chose to maximize reliability and ease of implementation at the price of providing a language with more restrictions than many common general-purpose languages. The optimal choice of language depends upon the particular application to be programmed, the level of efficiency required, and the programming methodology to be employed.

References

[1] Ichbiah, J.D., Heliard, J.C., Roubine, O., Barnes, J.G.P., Krieg-Brueckner, B., and Wichman, B.A., "Preliminary ADA Reference Manual," *SIGPLAN Notices*, Vol. 14, June 1979, Part A.

[2] Ichbiah, J.D., Heliard, J.C., Roubine, O., Barnes, J.G.P., Krieg-Brueckner, B., and Wichman, B.A., "Rationale for the Design of the ADA Programming Language," *SIGPLAN Notices*, Vol. 14, June 1979, Part B.

[3] Newbold, P., et al., *HAL/S Language Reference Manual*, version IR-61-9, Intermetrics, Inc., Cambridge, Mass., Sept. 1976.

[4] Jensen, K., and Wirth, N., *Pascal User Manual and Report*, Springer Verlag, New York, 1974.

[5] Addyman, A., et al., *Specification for Computer Programming Language Pascal*, New York, Feb. 25, 1979.

[6] Haberman, A.N., "Critical Comments on the Programming Language Pascal," *Acta Informatica*, Vol. 3, 1973, pp. 47-57..

[7] Haberman, A.N., and Nassi, I.R., "Efficient Implementation of Ada Tasks," Carnegie Mellon University, Dept. of Computer Science, Pittsburgh Pa., CMU-CS-80-103, Jan. 1980.

[8] Department of Defense, *STEELMAN Requirements for High Order Computer Programming Languages*, June 1978.