IS ADA TOO BIG? A DESIGNER ANSWERS THE CRITICS

Many have criticized the Department of Defense's new computer language, Ada, saying it is too large, too complicated, or too difficult to use. Are they right? And are there some simplifications that could be made to Ada without destroying its usefulness?

BRIAN A. WICHMANN

Ada has been widely criticized for being too large or complicated. Members of the language-design team have all been aware of this and have attempted to trim away the excess. The problem is that in making any significant simplification, one is almost bound to adversely affect one of the applications for which Ada was intended. This article addresses many of the criticisms and suggestions aimed at Ada. An invaluable source for this feedback is the over 5,000 comments on language design collected by the ARPANET system. Ledgard and Singer's article [1] calling for an Ada subset, as well as Lamb's article [4] pointing out the difficulties in such an undertaking, are also valuable.

The language design and requirements document of Ada have been extensively reviewed for a long time. In April 1975, C.A.R. Hoare led one of the first international meetings, which first invited open discussion of the purpose and scope of Ada, in Washington, D.C. The "Strawman" document, which was the earliest formal requirements specification for Ada, was a result of this meeting. Since then, there have been other opportunities for debating the language's strengths and weaknesses. The Department of Defense (DoD) should be praised for this open approach even if it has resulted in a large language not liked by everyone. Clearly, if the DoD is satisfied, the language is—at the most pragmatic level—a success. In Europe, Ada meets the needs not only of the military establishment but of a significant part of the industrial community as well. The telecommunications industry is also interested. In sum, the potential for widespread use of a common software tool is considerable.

Ada's major design goals and its intended area of application should always be taken into account during an evaluation. Ada cannot, for instance, be expected to be a match for BASIC as an interactive language on a personal computer. I address only those criticisms that take into account the complex ends Ada was designed to meet. (A good presentation of these ends can be found in the Ada rationale [6] which gives the reasons for the major design choices.)

Though I was a member of the Ada design team, I speak as an individual—not in any official capacity.

IS SMALL BEAUTIFUL?

The implicit assumption behind much criticism of Ada is—to steal a phrase from economist E.F. Schumacher—that "small is beautiful." Is this an appropriate criterion for programming languages? Certainly, it is less appropriate for languages designed with a large applications area in mind. Of course, there are alternatives to having a single, large language. For instance, a large number of small languages could do the work of a single, large language. The DoD, in fact, now has over 350 languages. Such a multiplicity is probably impractical and certainly uneconomical.

Let's turn now to some of the specific arguments critics propound for simplifying Ada—and my responses to those arguments.

1 *A simplified language would be less controversial*—This assumes that most criticism of Ada is a direct result of its size. In fact, most comments processed on the ARPANET were aimed at problems that could be overcome only by making the language more complex. For example, Ada, unlike Pascal, supports multiple floating-point lengths. Without this feature, some applications would be forced to use assembler language.

2 *If Ada were simpler, implementation would be less expensive*—Ada is certainly expensive to implement. The largest part of an Ada compiler, however, is machine-independent and need only be implemented once. To disseminate knowledge of the machine-independent part, its source text (written in Ada), could be published—as has been done with Pascal and BCPL compilers. If the Ada compiler can help simplify the programmer's job, it will pay for itself very quickly. The initial problems with implementing Ada have been due more to its novelty than to its size. These problems are gradually being overcome. The source text of an Ada compiler is now available; unfortunately, it is not of production quality.

3 *Implementations would be less prone to error*—This depends more upon the quality of the team producing the compiler than on any other factor. However, the Ada Compiler Validation Capability [10] set-up by DoD and consisting of a set of test programs, procedures for running them, and a service for formal validation should ensure that compilers are relatively bug-free provided users insist on purchasing validated compilers.

4 *With a simpler language, standardization efforts, which often get bogged down in a legal morass, would be simplified*—There does not seem to be any correlation between the size of a language and the time it takes to standardize it. It took the International Organization for Standardization (ISO) three years to standardize Pascal. But after 20 years, ALGOL-60 is still mired in the standards process. (The ISO process for standardizing Ada has an uncertain time scale.) This uncertainty does not seem to be due to the size of the language. Further, DoD is not permitting any variants or subsets—an approach that has wide support and one that should speed progress toward a standard.

5 *Having a simpler language would reduce the cost of validation*—Most validation costs occur only once—during the development of test programs. Language size has a relatively small effect on these costs. Machine time will, or course, be more expensive for a larger language. Ada validations will cost about six times those of Pascal, which is roughly proportional to the difference in the language sizes.

6 *The tendency toward subsetting would be moderated*—This is certainly true now that subsets abound. But this is a temporary condition attributable to implementation problems that are being overcome. By comparison, COBOL and FORTRAN—the two most popular languages—are notorious for extensions rather than for subsets. Ada's size and capabilities should allow users to resist the temptation to create extensions. The COBOL standard defines over 104,000 different subsets.

Is "small is beautiful" an appropriate criterion for programming languages? Certainly, it is less appropriate for languages designed with a large applications area in mind.

7 *A simplified Ada would be easier to teach*—It is, of course, easier to teach a smaller language. But how can one teach the principles of concurrency with a sequential language? Many academics welcome Ada because they can use it to demonstrate concepts such as abstract data types, concurrency, and error-handling within the context of a single language.

8 *Language forms would be easier to remember*—The general form of Ada is simple: it has 62 reserved words compared to COBOL's 260. Yet, COBOL is the most widely used programming language. An occasional use of the manual is natural in such cases, just as a writer's use of a dictionary is natural.

9 *The development of a conceptual model for understanding Ada would facilitate the learning process*—Many reviewers of Ada have been misled into thinking that the conceptual model for Pascal applies to Ada. This has led them to conclude that Ada is too complex. ALGOL-68, with its notion of orthogonal features, is a powerful language with a strong conceptual model; yet, it has not been very successful. The importance of a conceptual model for a programming language is not clear. After all, there is no simple conceptual model for the English language, yet the British are happy with it and the Americans manage to go wrong only occasionally (or is it the British that go wrong?). Surely one can use simple forms if a complex structure with numerous clauses becomes too cumbersome and obtuse. Thinking clearly about the application

in hand is far more important than the language being used.

10 *The likelihood of errors would be reduced*—The most expensive errors are those not discovered until execution. With Pascal, many of these errors are overcome by checks during compilation. This makes a Pascal compiler larger, but worthwhile. Ada continues this tradition. Errors would be further reduced if Ada had a secure form of separate compilation.

11 *Diagnostic messages would be less confusing*—The messages given by a compiler are so dependent upon the implementation that it is unwise to attribute confusing error messages to the language itself. For instance, many Pascal compilers give "error in factor" for "x := .123;" rather than "0 missing before decimal point."

12 *Users need not learn so much irrelevant information*—Agreed; users need to be selective in learning Ada for the first time. Such selectivity is important in many areas of computing. For instance, one does not expect to learn all the facilities of an operating system before using it for simple jobs. With Ada, there are many areas that can be avoided at first (e.g., tasking, generics, representation clauses). Some criticisms of Ada can be attributed to the lack of a teaching manual. Many users expect to be able to use the reference manual as a teaching manual. Good textbooks on Ada will present the language so that the initial material to be understood is little different from other computer languages.

In Europe, Ada meets the needs not only of the military establishment, but of a significant part of the industrial community as well.

13 *With a simpler language, the development of automated tools would be simpler and cheaper*—A complete tool set for the development and maintenance of Ada programs is envisaged as part of the "Stoneman" plans [8]. Many of these tools will process Ada source text. Such tools, some point out, will be more expensive to develop for Ada than for simpler languages such as Pascal. True, the cost of developing these tools for Ada will be more costly. And true, it is important that these tools be available to users at a realistic price. But Ada promises to be useful in many areas; this should create a large potential market for Ada and Ada tools. Thus, the development costs for Ada tools can be spread over many users, with only a minimal portion of the financial burden falling on each.

SCALING DOWN

DoD requirements are the main reason for Ada's large size. The Steelman requirements [2] are demanding; designers working on the project took these requirements seriously because of their wide support among Ada's potential users. If reducing the facilities of the language forces applications to the assembler language, then this must be weighed against the convenience of simplification. In my view, there are substantial benefits to having as much of the computing community as possible using a common, well-standardized language. The benefits of large FORTRAN libraries and of thousands of well-trained FORTRAN and COBOL programmers speak for themselves.

We now turn to some proposals for removing specific features from Ada. Many of these come from Ledgard and Singer [1], while others are from the ARPANET files and elsewhere. I have classified these into three categories: 1) changes that would be disastrous for Ada; 2) changes that are possible but of questionable value; and 3) changes that are practical and worthwhile.

CHANGES THAT WOULD BE DISASTROUS FOR ADA

Eliminating the following features would seriously damage Ada. Such changes contravene the requirements of the language. If implemented, these changes would make Ada into a different language entirely.

Exceptions. For Ada to be suitable for real-time applications, Ada programs must be able to recover from errors. Removing exceptions from Ada will not remove them from the applications.

Generics (Templates for Module Construction). In principle, one could manage without generics by textual expansion of source code (i.e., a macro). But, this would undermine reliability (by requiring *n* copies of an algorithm), conflict with the principles of information hiding, and add significantly to maintenance costs. The need for generics and the power they provide is illustrated by the input-output packages that would not otherwise be credible. Further on, some simplifications to generics are considered.

Tasking (Concurrent Programming). The subset of Ada without the syntax for controlling parallel tasks is not significantly smaller than full-size Ada. The removal of tasks would not make compilers that much smaller, so it would be of little value to users of sequential Ada. The subset would be useless in many of the application domains for which Ada is intended.

Initialization of variables in a declarative part. Use of undefined or uninitialized variables are a common programming bug. Ada reduces the risk of this error by making initialization easier. This feature makes Ada programs easier to follow than Pascal programs.

Default initial values for types. There are two distinct instances of this in Ada: default initialization of an access type value to null and initial values by default for a record type. Both are essential for security. In preliminary Ada, the use of an uninitialized value raised the NO_VALUE_ERROR exception. This very common mistake must be minimized by a language, but the NO_VALUE_ERROR is too expensive to implement. (A few machines have hardware that allows detection of this error with tolerable costs. But since this is not the rule, a standard language cannot insist upon detection. In fact, Ada *does* allow the detection.) Consequently, the two above-mentioned forms of default initialization are available. The insecurity of pointers in Pascal is well-known, and Ada is superior in this respect. The default for record types is essential for private types in which a package provides a type in a secure manner.

User overloading of operators. Proposals to remove this feature make little sense to me. FORTRAN-77 has complex numbers with infix operations, that is, overloading of the operators (+, −, ×, etc.). In Ada, overloading is already present with the predefined operators. Ada allows users to define types similar to the predefined ones. By this means, it eliminates the need for a complex type or for a matrix type.

Limited, private types in packages. Limited, private types in Ada are types, defined by the programmer, whose properties are tightly controlled. The critics suggest such types can be implemented by access types, i.e., pointers. This is not true—the meaning of various operations would be different: Equality would be equality of the access type, i.e., pointing to the same object, whereas often the natural semantic is that of equality of values. Sometimes, a programmer even wants equality to be defined differently, as with rational numbers, where ½ = 2/4, etc. Again, the user definition of equality might be perverse, but it at least gives a Boolean result and is the complement of not equals. In sum, the deletion of this feature would not allow data types to have properties natural to the application.

Restriction of generic parameters to names only. This proposal would reduce generics to macro-substitution. The check for the legality of a generic body would have to wait until an instantiation was given. This would in turn mean that a user of a generic package would be confronted with details of the body if the wrong parameters were given for an instantiation. What this scenario demonstrates is that foolproof generic packages could not be written in Ada. In fact, substantial improvements have been made in generics for revised Ada so the generic body can be properly checked before an instantiation is given. Early detection of errors is an important goal of Ada, and would be inhibited by this restriction.

CHANGES THAT ARE POSSIBLE BUT OF QUESTIONABLE VALUE

Removal of the following features would only marginally change Ada. Most of these proposed changes are really a matter of taste. But for a majority of them, there are some clear disadvantages. The result, though, is still a usable language.

Derived types (i.e., new types derived from existing ones). There has been a lively debate within the Ada community on this one. The definition of derived types in revised Ada was unsatisfactory, but ANSI Ada [5] has rectified this. The derived-type mechanism (or something essentially like it) is needed for integer and floating-point types. Also, the stepwise refinement of data types is enhanced by derived types. The alternatives are not to use the refinement process or to use unsafe conversions, neither of which is very attractive.

Fixed point. Fixed-point facilities are needed for some real-time data analysis systems for which floating-point facilities are too expensive or too slow. The IEC floating-point standard [9], together with chips like the Intel 8087, make the need for fixed-point facilities less essential. However, since the fixed-point option can be ignored in such cases, it need not get in a user's way.

Many comments critical of Ada were aimed at problems that could be overcome only by making the language more complex.

More than a single exception. A more realistic proposal than the deletion of all exceptions is to allow just one. This is in line with the view that a program unit has simply "failed" rather than that it has failed for a particular reason. The difficulty with this approach is that, because the reason is unknown, recovery is either insecure or too drastic. As an example, one can safely handle NUMERIC_ERROR in Ada because the situations where it arises are well-defined, i.e., overflow of a computation. If only one exception were available, the input-output system would have to be redesigned. It is not easy to see how the redesigned package could be simple to use and yet allow convenient error recovery. In short, this is a dangerous proposal for real-time applications.

Number declarations. Number declarations are a means of introducing numerical values without a corresponding type, as with a literal. Isolation of "magical" numbers is an important programming principle. Logically, such values are often introduced to *define* the types in the program. Hence, to require that they be typed is to frustrate the logic. In detail, if a literal of 100_000 is used in a program and should be named, what type should it have? It cannot be INTEGER on some machines (on which the largest INTEGER may be 32767), while on others LONG_INTEGER would be overkill. Also, having given it a type, every use would be likely to require an explicit-type conversion. Number declarations provide a useful means of compile-time parameterization with numeric values.

The binary notation definition of real types. Ada is the first programming language that has defined the properties that real types must have. The definition, which covers floating point and fixed point, uses a binary notation. Whatever one's view of the specific definition, it seems inappropriate to criticize the size of a language because it is precise. These definitions do not appear to be overly restrictive; there has been very little objection to the method used. One can now prove that algorithms using reals are formally correct.

Positional aggregates. Strictly speaking, there wouldn't seem to be a need for both positional and named aggregates, i.e., values for both array and records. However, named aggregates are very verbose and positional aggregates are uninformative. If named aggregates are retained exclusively, then named parameters should be the only form for subprogram calls. Neither single choice seems appropriate. An important design aim with Ada is readability, and readability is essential for the successful understanding and maintenance of programs. Table-driven techniques can be easily followed with positional aggregates.

Logical operators over Boolean arrays. These are provided in Ada to allow for the efficient implementation of the Pascal concept of a set. The alternative would be to define a package for set operations. While this might make the language marginally simpler, it would not improve matters overall because the description of such a package is likely to make the manual longer.

Relational operators over discrete arrays. Pascal permits relational operators between strings. Ada makes a similar choice but at the more logical level of discrete arrays.

Short circuit expressions (i.e., avoiding complete evaluation of Boolean expressions). An annoying problem in Pascal, which is avoided in Ada, occurs when the complete evaluation of a Boolean expression would cause an error.

The development of an Ada package industry should make it possible to accommodate the experienced programmer, while not overwhelming the novice.

Reverse loops (i.e., loops counting down). Pascal has reverse loops, and so does Ada. The elimination of reverse loops would not enhance Ada's usefulness. Certainly reverse loops are not a common requirement, but the circumlocution needed when they are not available is unattractive. Even minimal Basic allows a step of −1.

Blocks. A local variable is needed for a **for** loop. Look carefully at the convoluted semantics of the **for** loop in ISO Pascal [3] for an illustration of what happens when blocks are not allowed. A parameterless procedure is not suitable because the text is not in the position needed.

Named parameters. Ada has a facility for using the *names* of parameters when setting their values for subprogram calls. Almost all operating systems have this facility to permit the convenient call of, say, a compiler with 20 parameters (most of which have default values). (Contrast this with a language having only named aggregates.) This facility, however, has always been controversial: It was not a Steelman requirement. But with readability a key goal of Ada, key word parameters, with defaults, help significantly. If Ada is successful, packages will be developed for a wide range of applications. Nonexpert use of such packages is aided considerably by key word parameters.

Default parameter values. Ada allows input parameters to subprograms to have a default value that is taken if no value is given in the call. Those who advocate the deletion of this feature argue that overloading can be used to obtain the same result. Logically, this is true, but the corresponding program text is longer, obscure, and less likely to be efficient. As noted above, such defaults aid nonexpert use of packages.

Machine-code insertions. Implementations are not required to provide for these. In many cases, though, it is convenient not to use an assembler language at all but instead one or two inline machine instructions. With such a facility, an Ada system could produce a more efficient linking system not necessarily compatible with the assembler language.

Separate compilation for subprograms. In Ada, separate compilation applies to either a package or a subprogram. The idea of basing separate compilation entirely upon packages is good, but has problems. For example, a main program is a subprogram and not a package. If that is changed, then the main execution of a program is the initialization code of a package that is hidden (!) in the body of the package. Also, conventional linkage to other languages is via subprograms, not packages.

Separate compilation for subunits. Subunits are units nested inside the main units and used for separate compilation. Subunits have two key advantages: First, because they have a unique name within a unit rather than the whole library, subunits avoid the unique identifier problem. Most current libraries have one namespace. Thus, with large systems, one is forced to use meaningless identifiers. Subunits minimize this problem. Second, with top-down program development, subunits are the natural device for separate compilation. If subunits are not available, then the Ada source text structure will no longer reflect the development process—a significant drawback.

CHANGES THAT ARE PRACTICAL AND WORTHWHILE

Eliminating the following features from Ada would, in my view, be worth the inconvenience.

Unnamed array types. In Ada, one can declare an array variable in one of two ways: by using an array type or a special construct. Why was this special construct included in Ada? It was believed users were accustomed to declaring an array without an array-type name. Without this special construct, it was believed, Ada acceptance would be more difficult. I would prefer to eliminate this special construct, even though others may find it useful.

When ***conditions in exit statements.*** The alternative coding is quite reasonable (even natural). One would therefore write

```
if FOUND then
    exit MAIN_CYCLE;
end if;
```

rather than

```
exit MAIN_CYCLE when FOUND
```

Goto statements. One must face the fact that it will be hard to sell Ada, to convert programs from other languages, or to automatically generate Ada programs (for a finite state machine, for instance) without this feature. Fortunately, an Ada goto cannot jump out of a subprogram (as a Pascal goto can). Thus, the worst aspect of this feature is avoided.

Entry families. This facility permits queues of concurrent tasks to wait on one function so that the programmer can control the order of processing in the queue. This, however, can usually be handled instead by an additional task. The semantic is quite complex and the justification uncertain until more experience is gained with the basic Ada rendezvous mechanism. These liabilities make this facility a reasonable candidate for deletion.

CONCLUSION

Ada can be made simpler by reducing its facilities. It is far from clear, however, that the resulting language will be as useful to the user community especially in the long run.

There are simplifications which make good sense, but there are many more that only seem to make good sense. I have tried to differentiate among these, and in so doing have created three categories: changes that would be disastrous for Ada; changes that are possible but of questionable value; and changes that are both practical and worthwhile. Changes at the second level, the "possible" changes, are naturally the most controversial. If every one of these changes were made, however, Ada compilers would be only 20 percent smaller with a corresponding reduction in the size of the manual. Would this be of much benefit to the long-term user of Ada?

The advantages of a single, highly standardized language should far outweigh the inconvenience presented to individual users by Ada's complexity. For any given user, of course, the language contains unneeded features; the objective is to accommodate the experienced programmer with essential tasking for more demanding applications, while not overwhelming the novice. The development of an Ada package industry should bring this objective more within reach.

All indications are that Ada compilers will be bigger and slower than Pascal compilers. Fortunately, single-chip microprocessors are now becoming available with addressing beyond 16 bits. These should facilitate the development of inexpensive systems to support Ada compilers. All software houses can afford such systems given adequate Ada compilers and programming support environments.

In ten years, we will know how Ada should have been designed. It is the decisions we made in the original design phase, and the alterations we make today, though, that really count. From my perspective, there are no significant simplifications that can be made to Ada that would not materially reduce its application area. Those who believe that Ada contains a good subset should provide a textbook setting forth that subset. Such a text would be invaluable to the various industries about to start using Ada, for it could serve as a kind of introduction to the language and as a way of holding down retraining costs. Of no less importance is that it could help to establish a solid foundation for a more thorough knowledge of this formidable programming language.

Acknowledgments While the views expressed in this article are mine, I received valuable comments on an earlier draft from the following people: J.D. Ichbiah, H. Huenke, R.W. Meijer, M.W. Rogers and R.S. Scowen. *Communications'* editor-in-chief, P.J. Denning, and referees also contributed significantly to the revision.

REFERENCES

1. Ledgard, H.F., and Singer, A. Scaling down Ada (*or* (Towards a standard Ada subset). *Commun. ACM 25*, 2, (Feb. 1982), 121–125. Also in *The Ada Programming Language: A Tutorial* Sabina H. Saib and Robert E. Fritz (Eds.), IEEE Catalog No. EHO 202-2 (1983).
2. "Steelman" (Department of Defense requirements for high-order order computer programming languages). Department of Defense. (June 1978).
3. British Standards Institution. Specification for the computer programming language Pascal. BS6192. (Also International Standard Organization 7185) BSI (1982).
4. Lamb, D.A. Subsets. *Ada Letters*, Nov./Dec. 1982, pp. 14–15.
5. Reference Manual for the Ada programming language. ANSI/MIL-STD 1815A, February 1983.
6. Ichbiah, J.D., Barnes, J.G.P., Heliard, J.C., Krieg-Brueckner, B., Roubine, O., and Wichmann, B.A. Rationale for the Design of the ADA programming language. *SIGPLAN Notices 14*, 6, Part B, June 1979.
7. Dewar, R.B.K., Fisher, G.A., Schonberg, E., Froehlich, R., Bryant, S., Goss C.F., and Burke, M. The NYU Ada Translator and Interpreter. *SIGPLAN Notices 15*, 11 (1980), 123–127.
8. "Stoneman" Requirements for Ada Programming Support Environments. Department of Defense, Feb. 1980.
9. International Electrotechnical Commission *Binary Floating Point Arithmetic for Microprocessor Systems.* Publication 559, International Electrotechnical Commission, Geneva, 1982. (This is slightly different from the current IEEE proposal.)
10. Goodenough, J.B. Ada compiler validation capability. *SIGPLAN Notices 15*, 11 (Nov. 1980), 1–8.

CR Categories and Subject Descriptors: D.3.0 [Programming Languages]: General—*standards*
General Terms: Languages Standardization
Additional Key Words and Phrases: Ada, language size, success of a language

Received 8/83; accepted 9/83

Author's Present Address: Brian A. Wichmann, Division of Information Technology and Computing, National Physical Laboratory, Teddington, Middlesex TW11 0LW, United Kingdom