The "Computerization" of Programming: Ada(tm)-Lessons Learned

Dennis D. Struble
Intermetrics, Inc.
733 Concord Ave.
Cambridge, MA 02138

1.0 Introduction

During the past four years, Intermetrics has constructed one of the largest systems yet written in Ada. This system is the Intermetrics Ada compiler. As you might imagine, Intermetrics has learned many lessons during the implementation of its Ada compiler. This paper describes some of these lessons, concentrating on those lessons relevant to large system implementations.

As I considered what lessons to discuss an amusing thought occured to me. Four years ago I gave a briefing at the Johnson Spacecraft Center entitled "Ada: A Management Overview." At that time, I was an ardent Ada proselytizer but one who had never laid hands on an Ada compiler. In that briefing four years ago I made several predictions about what it would be like to manage an Ada project. Having spent the last two years managing an Ada implementation, I thought I ought to determine how accurate my predictions had been. (As you might guess, my predictions turned out to be correct. If they hadn't, there certainly would have been no point in admitting to them in this paper.)

Before I identify these predictions, I'll first describe the characteristics of the Ada compiler implementation project at Intermetrics. Then after listing the predictions I will describe some specific experiences which verify these predictions.

2.0 Project Description

The Intermetrics Ada compiler and linker comprise 400,000 lines of Ada code. The compiler is augmented by a program library manager and by a set of tools which are together another 100,000 lines of Ada. The tool set includes a source lister which optionally includes the generated assembly code, a completeness checker, a body generator, the Byron(tm) design language processor, a debugger, and a set of static and dynamic program analyzers.

Ada(tm) is a registered trademark of the U.S. Government (Ada Joint Program Office).
Byron(tm) is a trademark of Intermetrics, Inc.

Intermetrics is currently completing a total of six compilers under two government contracts and four commercial contracts. The compilers generate code for the IBM 370, the Sperry 1100, and the MIL-STD-1750A instruction sets; this generated code executes in six different run-time environments: IBM MVS(tm), IBM CMS(tm), Amdahl UTS(tm), Sperry 1100, and bare 1750A. The compilers are hosted under four different operating systems: IBM MVS, IBM CMS, Amdahl UTS, Sperry 1100, and VAX VMS(tm).

All of these compilers have been developed in parallel and all of the compilers share the same source code. The source code is maintained under a configuration management system designed specifically to support a multi-hosted and multi-targeted compiler development environment. The development staff at its peak included fifty software engineers.

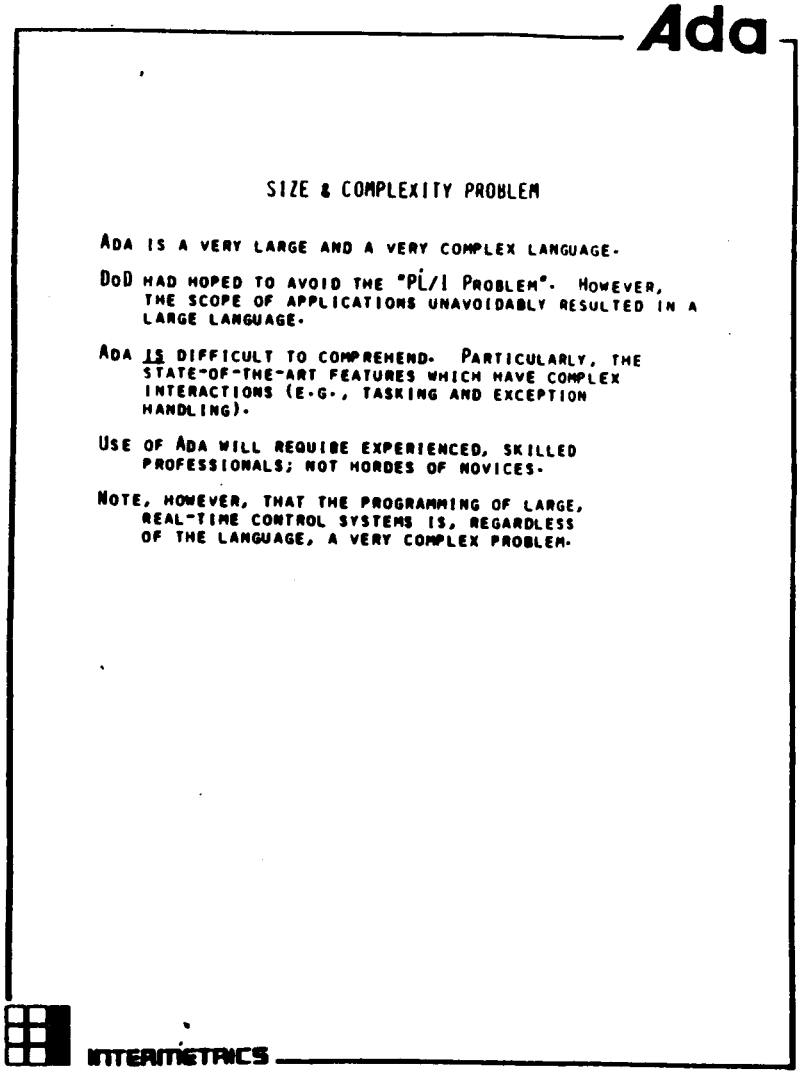
The development environment for the Ada compilers is an IBM 3083 Model BX, running the Amdahl UTS operating system hosted under VM. The production-quality compiler was initially developed using an Ada-subset compiler Intermetrics wrote in Pascal. The production-quality compiler was validated in December 1985 and was bootstrapped through itself in February 1986.

3.0 The Predictions

Figures 3-1 and 3-2 are extracted from the four-year old briefing I described above. The predictions contained in these figures are self-explanatory. Of these predictions, the ones concerning multi-tasking are, of course, not relevant to our compiler. (Not yet at least; Intermetrics is anxious to modify our compiler to become the first Ada compiler to take advantage of the new generation of multi-micro machines.)

All the other predictions have turned out to be more or less correct. One theme that runs through these predictions is that with the introduction of Ada, the DoD is attempting to take a major step forward in the "computerization of programming." I use the term computerization of programming, rather than "automatic programming" because I believe that for completely new applications, such as Ada compilers and Space Station software, automatic programming will never occur. On the other hand, many of the tasks required in the programming of new systems are amenable to much greater computerization. In particular, Ada requires much more "bookkeeping" to be performed by the compiler than do other languages.

IBM(tm), MVS(tm) and CMS(tm) are trademarks of the International Business Machines Corporation.
UTS(tm) is a trademark of the Amdahl Corporation.
VAX(tm) and VMS(tm) are trademarks of the Digital Equipment Corporation.



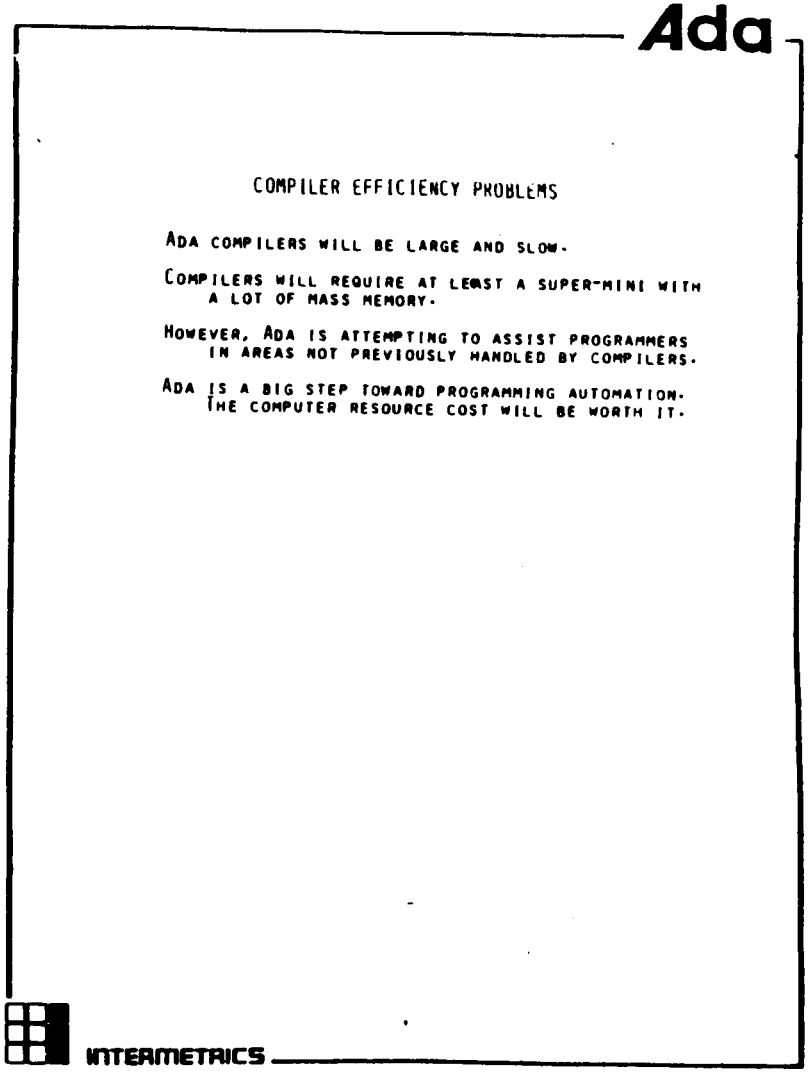
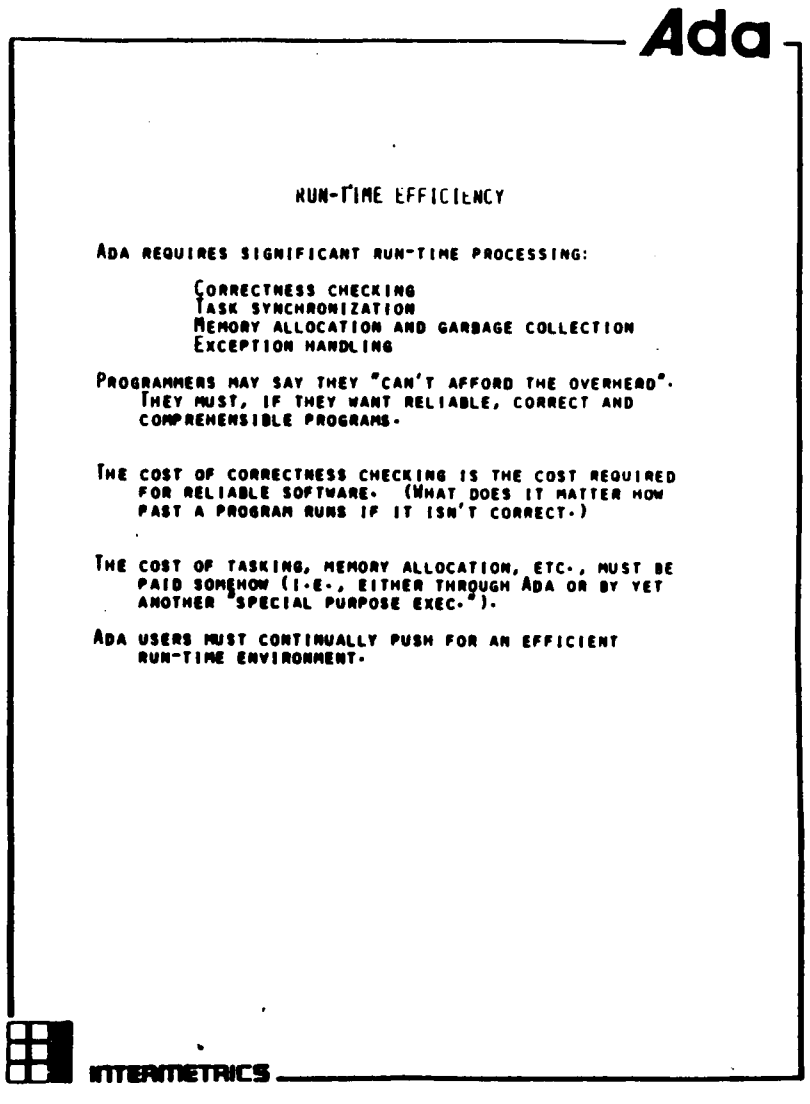


FIGURE 3-1: 1982 PREDICTIONS (1 of 2)



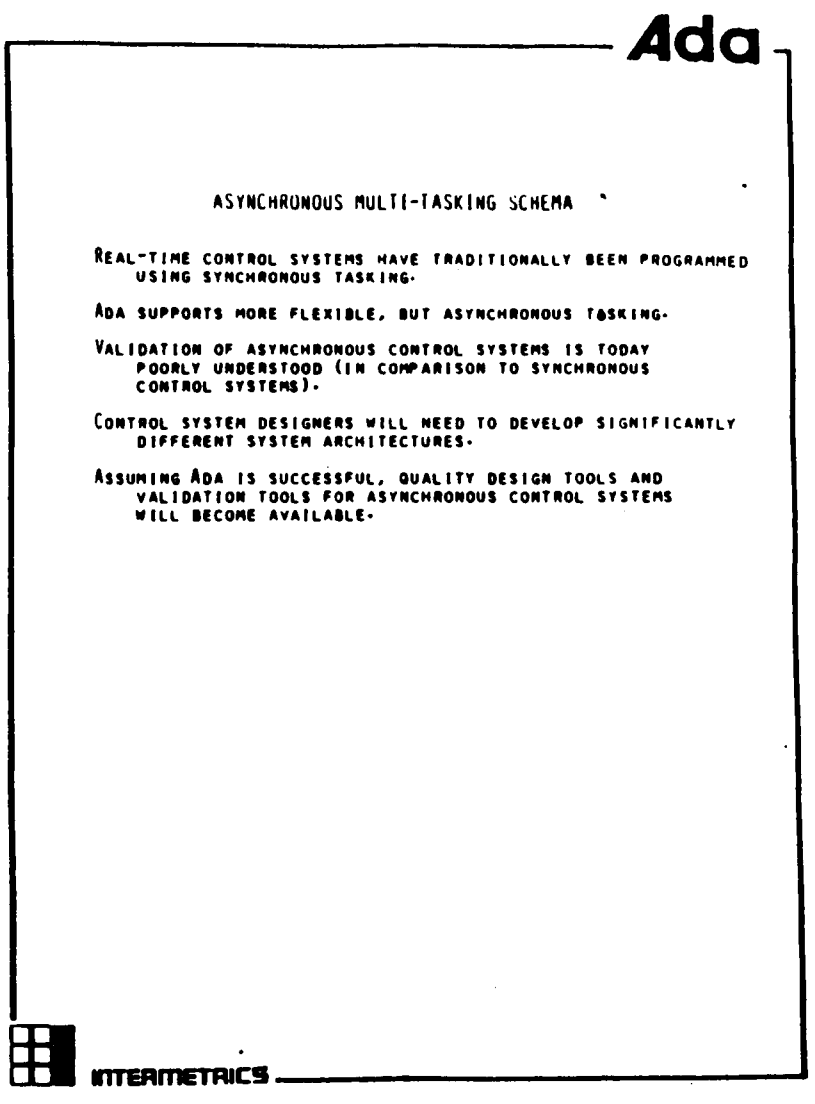


FIGURE 3-2: 1982 PREDICTIONS (2 of 2)

A more significant computerization of programming arises because Ada fosters, if not requires, a database management approach to the handling of software. That is, each Ada package should be treated as a valuable, complex, and evolving piece of data; database management facilities and procedures should be provided that are commensurate with the value and complexity of this data.

As Intermetrics has further computerized its software implementation procedures through the use of Ada, Intermetrics has learned several lessons which confirm those four-year-old predictions, as well as some lessons that could not have been anticipated four years ago. These lessons are described below.

4.0 Ada-Lessons Learned

The lessons Intermetrics has learned may be split into the following categories: Ada Training, Ada Tools, and Ada Language Use.

4.1 Ada Training

One of the predictions states that the use of Ada would required well-educated software engineers. Implied by this prediction is a possible short-fall in software engineers trained in Ada and trained in the software engineering principles that Ada encourages.

In fact, availability of trained Ada engineers has not been a problem at Intermetrics. This is because the Intermetrics Software Systems Group employees computer scientists who specialize in support software. Most of our new employees already know Ada and already know the system design principals associated with Ada software engineering.

Ironically, in some cases this broad knowledge of modern language technology has actually caused problems. Some engineers who have worked with university-developed, state-of-the-art languages expect Ada to behave the same way. Many of these state-of-the-art languages emphasize expressability, perhaps at the expense of run-time efficiency, whereas run-time efficiency was a key criteria in the design of Ada (and has been a key criteria in the development of the Intermetrics Ada compilers.)

An example of the problems caused by an orientation to state-of-the-art languages arises from the CLU programming style which advocates regular use of "signals" to return status from subprograms. Several new Intermetrics employees have assumed that in a corresponding way, exceptions should be used in Ada programming to return subprogram completion status. In fact, Ada exceptions are intended for truly "exceptional"

circumstances. Efficient Ada compilers attempt to generate code in such a way that exceptions require no processing time unless, and until, the exception is signalled. However, when the exception is signalled, substantially more processing is required than simply returning an output parameter. Thus, use of Ada exceptions is not analogous to use of CLU signals. Through coding standards and code reviews, Intermetrics educates its programmers into efficient use Ada programming.

4.2 Tool-Use Lessons

In using high-order languages, Intermetrics often has found that the quality of the compiler is more important than the quality of the language. Certainly in the initial years of Ada use, this will be the case. Three characteristics of Ada tool usage are discussed below: the importance of the library manager, the unfortunate variability among Ada compilers, and the substantial computing resources required by Ada tools.

4.2.1 A Sophisticated Library Manager is Critical

During the parallel construction of the six compilers, all of which share the same Ada program library, the necessity for a database management approach to Ada software configuration management became clear. It is the Ada program library manager that provides this database management. This database manager must provide the following services:

> Separate development areas for projects and sub-projects along with a facility to share formally "released" packages among projects and sub-projects.
>
> Management of variants of subsystems, where these variants support rehosting or retargeting the overall system.
>
> Formal configuration management of successive versions of subsystems.
>
> An interactive facility that can answer queries concerning the status of packages in the library as well as queries concerning dependencies among packages.
>
> An interactive facility which supports constructing a system by choosing specific variants and versions for each sub-system.

4.2.2 All Ada's are not the same

During the developement of its Ada compilers, Intermetrics has used three different Ada compilers and attempted to use a fourth. The three successfully used compilers are the two Intermetrics compilers and the DEC (tm) compiler. (One Intermetrics compiler and the DEC compiler are validated compilers.) Not surprisingly, these compilers do exhibit enough variation that rehosting a large system from one compiler to another is a substantial undertaking. Some of the major differences Intermetrics encountered are listed below.

Three classes of differences were experienced: functional, capcity, and performance. Two functional differences were noteworthy; the first arises because Ada does not specify a default elaboration order. Thus, unless pragma elaborate is used exhaustively to explicitly order the complete elaboration, a complex system may elaborate correctly using one compiler and yet fail to elaborate using another.

The more troublesome functional problem involved the different handling of un-initialized records. It is, of course, incorrect to rely on un-initialized variables. Nevertheless, it is common in large systems developed using a compiler that does initialize all variables to zero by default, that this large system will work correctly even though some variables are not explicitly initialized. When such a large system is rehosted to a compiler with a different default initialization, it becomes extremely costly to identify the un-initialized objects.

At times potential customers have asked us to rehost our compiler front-end and Byron tool set to systems already having an Ada compiler. In one case we were unable to respond to the request because the existing compiler did not have the capacity necessary to compile the largest units in the Intermetrics compiler. (Generally, the Intermetrics compilation units are from ten to several hundred lines; however, there are a few very large packages in the compiler. These packages include the parser tables, the code-generator tables, and the DIANA access package.)

The most serious difference we encountered was the speed of our compiler as compiled by different compilers. We, of course, expected variation in the code quality among the different compilers; when we forecast the speed of our compiler on the VAX as compiled by the DEC Ada compiler, we took into account the difference in code quality and difference in machine speed. Nevertheless, our I/O-intensive, host-interface package, which conforms to the CAIS file model, ran much more slowly on the VAX than anticipated. We eventually identified Ada file open and close operations as the cause of this anomaly. The lesson is that for extrapolating the performance of a systems-level Ada program, a simple comparison of code-quality is not sufficient.

There are straightforward procedures which may be used to avoid these compiler variability problems. Foremost is the identification of those aspects of Ada which may vary from compiler to compiler and establishment of coding standards addressing these variations. If you know in advance that your system will be rehosted to several compilers, investment in a standards checker will definitely pay off.

For a large project such as the Space Station which will have the resources to modify its compilers, it would be appropriate to enhance each compiler to flag possible sources of incompatibilities and to generate code that conforms with the anomalies of other compilers. For example, Intermetrics is considering adding a DEC-Ada compatibility option to the Intermetrics compiler so that we may minimize the recurring cost of rehosting the Intermetrics compiler to the VAX.

4.2.3 For Ada, Don't Underestimate the Computes!

Sure enough, Ada compilers have turned out to be big and to be slow. Despite what some may hope, an Ada compiler will always be slower than an equivalent Pascal or C compiler; it's a simple issue of algorithmic complexity. Again, Ada is attempting to computerize software engineering substantially more than have previous languages; this computerization requires substantial computing resources.

4.2.3.1 Compile and Link Speed

All potential Ada users are aware that average compilation speed is a critical compiler characteristic. Nevertheless, in addition to the average lines-per-minute speed of Ada compilers there are several other compilation speed issues that are unique to Ada. These are start-up overhead, speed of separate compilation, and up-to-dateness checking.

Ada compilers have a start-up overhead greater than previous compilers. This arises from the size of the compiler executable and from the requirement to interact with a large database, namely, the program library. Consequently, the cost of compiling very small modules is greater than with previous compilers. This cost should be taken into account when estimating computing resource requirements and perhaps when partitioning your system into compilation units.

One Ada's most valueable characteristics is its requirement that the compiler verify module interfaces. Once again, this further computerization requires processing time. Each package that a given package "with's" must be accessed and its interface information made available to the current compilation. Extended chains of "with" dependencies across packages add further accessing cost. Thus, the hierarchical structure of large

systems must be designed carefully to avoid including extraneous dependencies among packages. Further the dependency structure should be periodically re-assessed during a long implementation effort to determine if adjustments to this structure would improve compilation time.

Ada compilers and linkers are required to check the "up-to-dateness" of Ada packages. In a large system with a complex library structure, the look-up required to verify up-to-dateness will be significant. Again an understanding of this issue is important when evaluating Ada compilers and when estimating required computer resources.

4.2.3.2 Disk Storage Requirements

Systems written in Ada will require substantially more disk storage than previous systems. This arises from two factors. First Ada requires a program library that maintains interface information from preceding compilations. Secondly, and more importantly, some Ada compilers, including the Intermetrics Ada compilers, provide an open interface into the internal data structures that describe the packages of the compiled system. The Intermetrics Ada compilers provide this open interface through DIANA. A program library containing a DIANA description of each package in the system enables the construction of a set of tools that can analyze these packages. These tools include static analyzers, dynamic analyzers, debuggers, package status reporting tools, and package documentation tools. An advantage of an open interface is that a given project, like the Space Station, can readily implement whatever analysis tools the project requires.

This open interface facility does have a computer resource cost, namely more disk storage than required by previous languages. In evaluating this cost, managers must recall that with the advent of Ada compilers which provide a DIANA-based program library, we are taking a significant step toward a database-oriented view of software systems. Such a methodology does imply the disk storage resources required for a large database.

Recognizing that a given project may not want to provide the resources necessary for a complete DIANA database, the Intermetrics Ada compilers will provide the option to retain only enough DIANA to support Ada interface checking. Even though Intermetrics will provide this option, we do anticipate that most projects will find the benefit provided by the DIANA-based toolset will substantially outweigh the cost of the disk storage.

It is interesting to note that the issue of program library size and program library functionality is only slowing beginning to appear in various Ada compiler evaluation criteria. This is

because a sophisticated program librarian and its disk storage requirements were never an issue with Ada's predecessors. With Ada, the characteristics of the program library may well become one of the key distinguishing characteristics of Ada compilers. The functionality of the library will determine how effectively a large number of programmers will be supported and how effectively parallel development efforts will be supported. The size of the program library will be an important parameter when a manager budgets for computer resources.

4.3 Language-Usage Lessons

Building one of the first large systems in Ada is like attending a grand buffet banquet in a foreign country. There's a table full of goodies that look incredibly delicious. The problem is: some of the goodies may not agree with you and there are so many goodies it would be very easy to overeat. Listed below are some of the Ada features that in some case turned out to be a little too rich.

4.3.1 Beware Abstraction Overdose!

From its inception, the Intermetrics compiler was designed and coded fully utilizing Ada's excellent support for data abstraction. Each of the compiler's major data structures is designed as a data abstraction with an appropriate set of access procedures. The compiler's heavy reliance on abstractions has worked out well from both the robustness and flexibility standpoints.

For example, the compiler was designed with a software paging system that would manage the storage for the various intermediate languages. During the first year, while the paging system was being the implemented, a simple, memory-resident system was used as a substitute. When the time came to switch over to the paging system, we anticipated a lengthy integration and debugging phase. However, because the underlying implementation of the storage primitives had been hidden, the switch-over phase proceded with almost no bugs.

Data abstraction does, however, have a negative side: data abstraction, particularly if overused, can substantially degrade a system's performance. Going through multiple levels of abstraction, each one of which is a procedure call, is expensive. As we complete our compiler, we find ourselves having to "collapse" some of these levels, specifically, the parser's access to the parse tables and the code generator's access to the code tables.

Having experienced both the benefits and costs of heavy use of data abstraction, we believe the best approach is to start out with those abstractions that best support initial

development and integration. However, a project manager must definitely budget time and effort to measure the cost of abstraction usage once the system has been integrated. And unfortunately, a project staff probably will need to tune some of the abstraction usage in order to meet the project's performance requirements.

4.3.2 Don't Touch that Spec (and leave my body alone too!)

A key Ada design prinicipal is the physical separation of package specification code from package implementation code. An intended benefit of this separation is the avoidance of re-compilation that could result from changes to the implementation code. Intermetrics experience shows, however, that the simple division into spec's & bodies does not guarantee minimal compilation.

To assure minimal re-compilation, management diligence is required. Ada's strong interface checking has its downside. In C, Pascal, or FORTRAN, modules are not strongly connected and hence modules may be recompiled readily. In Ada, packages are very strongly connected and if changes to packages are not managed, one can spend enormous amounts of computer dollars re-compiling.

The strongly connected aspect of Ada necessitates a software development approach that emphasizes bottom-up coding and unit testing. The hierarchy of packages must be built in a manner that freezes the interfaces and thereby prevents undesired recompilations. This development approach is, of course, a standard aspect of good software engineering and most projects do attempt to adhere to this approach. Nevertheless, when using Ada, the cost of not following this approach become greater since Ada will force recompilations whenever the interfaces appear to have changed (even if the programmer knows they haven't).

Another aspect of interface management arises because Ada's spec and body separation is not as strong as normally believed. Changes to generic bodies and to in-lined procedures will cause recompilation. Consequently, managers must make sure that the staff is aware of these possible body dependencies and structure their packages to minimize re-compilation necessitated by changes to both spec's and bodies.

In addition to fostering a package partitioning that minimizes recompilation, a manager should also make sure the project's APSE includes a what-if analyzer. A what-if analyzer answers the question: "What recompilation would result if I make the following change to this spec or to this body." This tool is particularly valuable during maintenance when a substantial change, for example for performance reasons, is being contemplated. It is likely that a maintainer would not fully

understand the recompilation dependencies in a large system. A what-if analyzer could guide the design toward one which avoids substantial recompilation.

4.3.3 Lady Lovelace, she doth nag.

Ada's pervasive constraint checking is thought by many to be a meddlesome annoyance best handled by liberal use of pragma suppress. Intermetrics did not agree with this view at the inception of its Ada development and our experience has shown our perception to be correct.

Constraint checking has been perhaps the most valuable Ada feature we've enjoyed during the compiler's development. The positive attributes of contraint checking include:

> Bugs manifest themselves very close to their "time of occurrence." In developing a compiler this is critical, since the generation of incorrect code, when undetected, produces the most difficult bugs. Fortunately, ninety percent of the time, our compiler failed with a constraint check rather than blithely generating incorrect code.
>
> By providing appropriate exeception handlers, bug occurrences can be made somewhat self-documenting. That is, an exception handler can identify the context in which the constraint error occured. For example, when a contraint error occurs in our compiler, it prints out the line number of the source line being compiled and dumps the relevant internal data structures. (This contrasts with the more conventional, unadorned "memory exception" and "operation exception".)
>
> Given self-documenting failures, contraint checking allows an independent test group to play a much more active and productive role in the checkout and debug process.

Because of the value of constraint checking, Intermetrics took special care to design an optimizer that would remove all unnecessary constraint checks. Unfortunately, with constraint checking, the compiler can't do the whole job. Minimization of constraint checking also requires good Ada programming. Precise type definition is critical to avoid unnecessary constraint checks. A carefully written Ada program compiled by a good Ada compiler should result in no more checking-code than would an equivalent C program containing that amount of assertion checking mandated by good software engineering standards.

While we were using our subset compiler for development, we were concerned with the possibly unacceptable amount of constraint checking that would exist in the completed compiler. Fortunately, we were quite pleased with the contrast between no constraint check elimination in the subset compiler and excellent constraint check elimination in the production compiler. In fact, Intermetrics currently plans to achieve its performance requirements without resorting to pragma suppress. Retaining the necessary constraint checks in the compiler will markedly improve the maintainability of the compiler.

4.3.4 Look Ma - No Regressions!

The problem of regressions is indeed lessened in Ada. Prior to Ada it was often the case that in fixing a bug in a large, complex system other bugs were introduced into the system. The strong structuring support and strong typing that Ada provides make it more difficult to introduce incorrect fixes into a large system.

This characteristic of a system written in Ada was clearly indicated during both the validation and the bootstrap of our compiler. We had expected, based on prior compiler experience, that we would experience a two or three week "tail" at the end of our pre-validation testing. This tail would occur as we attempted to pass the final five percent of the ACVC suite. We expected that a fix introduced to pass one of the last ACVC's would cause one or two previously passing ACVC's to begin to fail. In fact, this regression did not occur. Our rate of getting new ACVC's to pass remained high right up through the week in which the last ACVC's were passed.

A similar phenomenon occurred when we bootstrapped our compiler. To manage the bootstrap process, we decided that we would first bootstrap the smallest compiler phase, using this mini-bootstrap to expose the majority of compiler bugs we would experience during the full bootstrap. This smallest phase is the 70,000 line, global optimizer phase. Its bootstrap required three months. During the three months 55 bugs were exposed and fixed. This bug rate corresponds to 8 bugs for each new 10,000 lines of new code exposed to the compiler.

In forecasting the bootstrap of the remaining 330,000 lines of the compiler, we estimated that these new lines would produce bugs at 4 bugs per 10,000 lines, for a total of 130 bugs. Given this number of bugs, we estimated it would require twelve weeks to bootstrap the entire compiler. To our pleasant surprise, we bootstrapped the compiler in five weeks and the additional 330,000 lines exposed only 10 new bugs!

We attribute these two instances of fewer bugs than expected to the "correctness" discipline which arises from programming in Ada. Ada does indeed appear to result in systems which have few "lingering" bugs and are readily maintainable.

5.0 Conclusion

Intermetrics realized five years ago that writing a production quality Ada compiler would be a tough job. Writing the compiler in Ada itself made the job really tough.

This heightened difficulty arose not because Ada isn't an excellent systems programming language. The difficulty arose from a situation which occurs too often in our industry: the dependence on a brand-new programming support environment for a large systems programming effort.

Fortunately, this situation is behind us. Intermetrics has a production quality, programming support environment that efficiently supports continued development of the Intermetrics Ada compilers. Intermetrics has also learned a great deal from its 150 person-years of Ada development; hopefully, the lessons described in this paper will benefit the planning and implementation of the Space Station software.