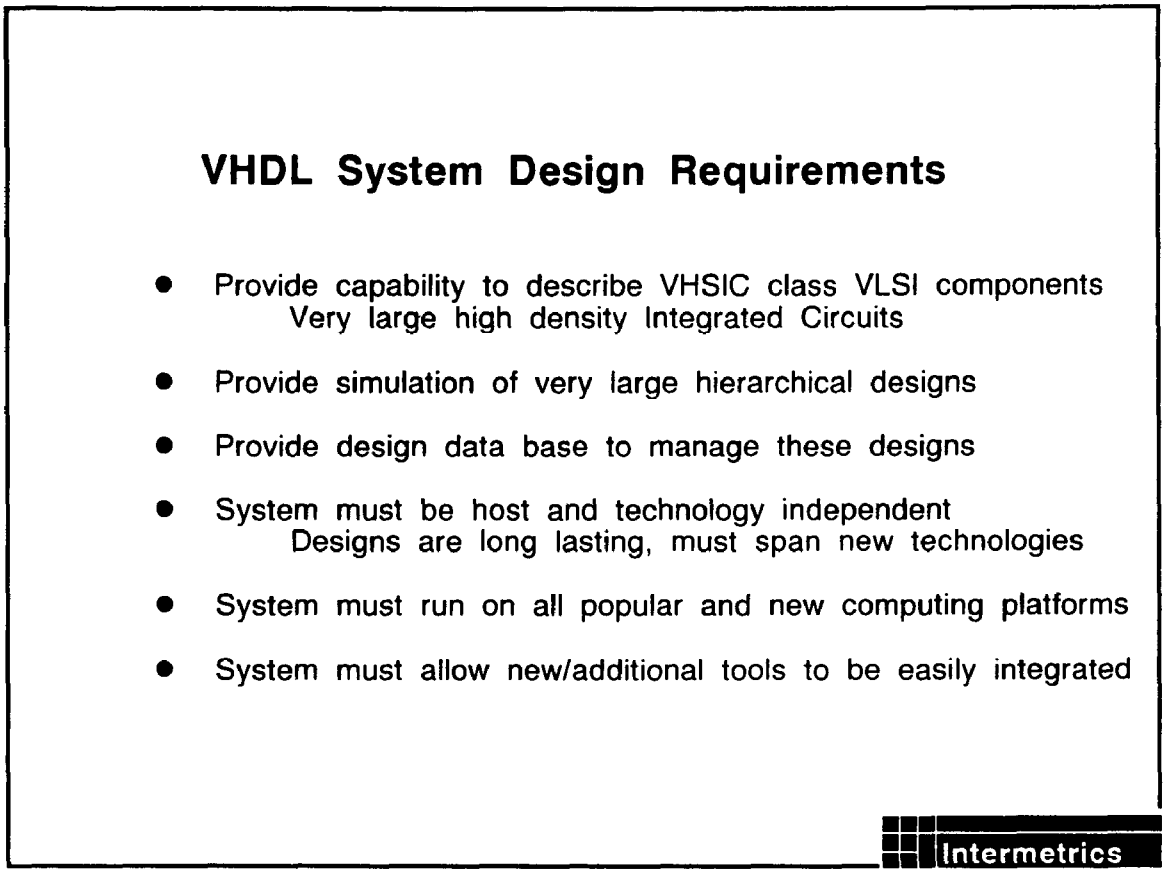
The VHDL Design System
An Ada-Based ECAD System for VLSI

Victor Berman
Carl Schaefer
Intermetrics, Inc.
4733 Bethesda Avenue
Bethesda, MD 20814

Carl Schaefer is a Senior Software Engineer at Intermetrics, Inc., in Bethesda, MD. He has been active in Intermetrics' VHDL program since 1983 and directed the implementation of the 1076 VHDL Analyzer and Simulator. He also directed a Diana configuration management effort which resulted in Revision 4 of the Diana Reference Manual (1986). He has a MS in Computer Science and a PhD in Linguistics.

Victor Berman is the Program Manager for the VHDL development program at Intermetrics, Inc. He has been involved in the development of computer languages and support software for eighteen years in both management and technical roles. He has a MS in Electrical Engineering from Carnegie-Mellon University where he also did graduate work in Computer Science toward a PhD.

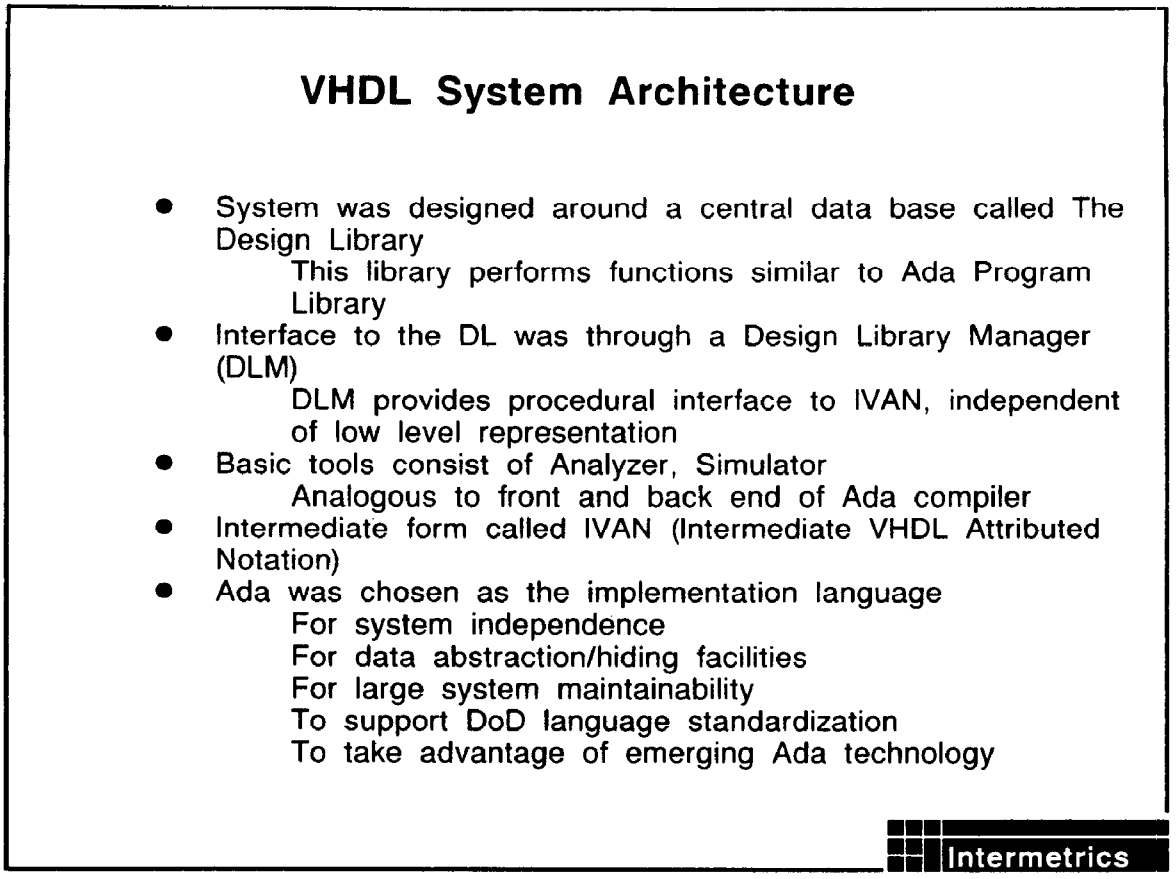


VHDL System Design Requirements

The VHSIC (Very High Speed Integrated Circuits) Program is a Tri-Service program sponsored by the Department of Defense to develop an integrated circuit capability that has the potential of greatly enhancing the performance of electronic defense systems.

The motivation for the development of a hardware description language came from the VHSIC program goals of reduced integrated circuit design time and effective VHSIC technology insertion into military systems. These goals indicated the need for a common interface, a standard means of communication, that would streamline the design and documentation of advanced digital systems. The means of communication was identified as a hardware description language.

The requirements for VHDL and for it Support Environment stem from these needs.



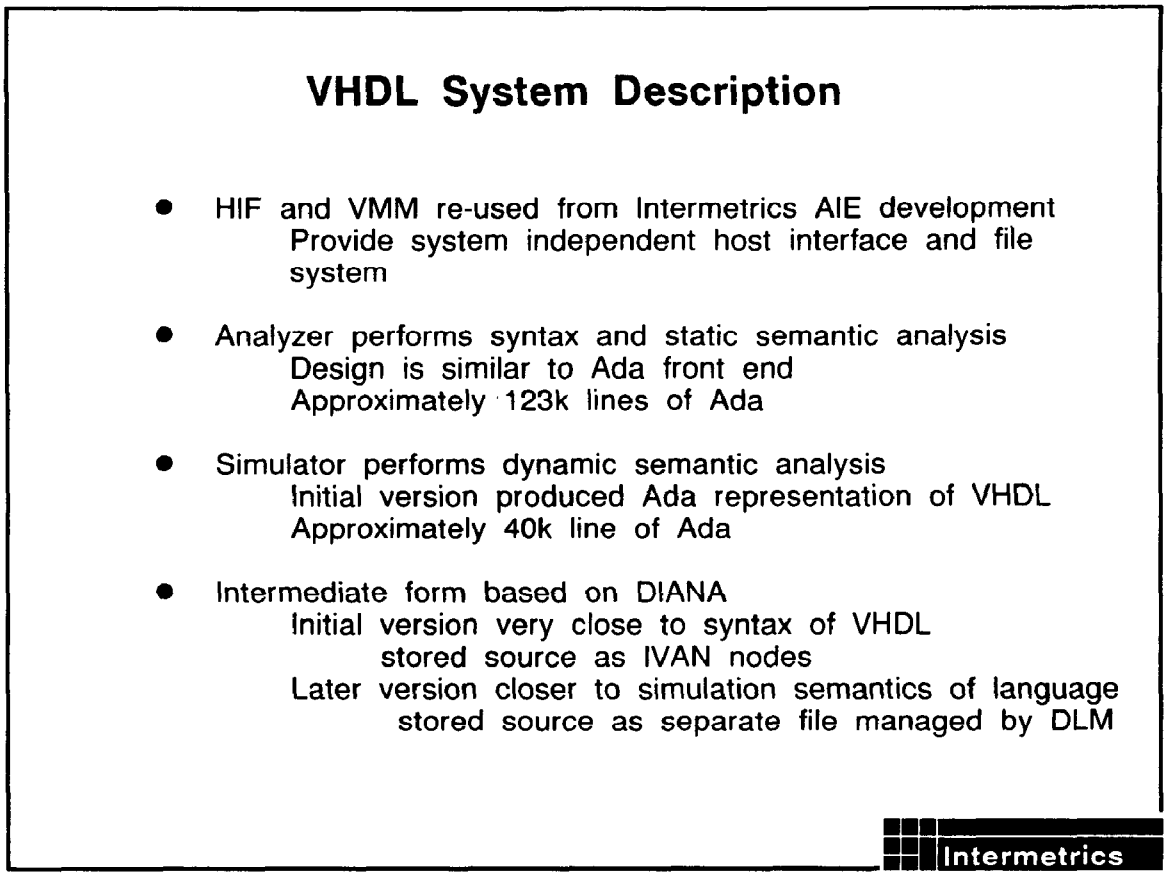
VHDL System Architecture

The central object in the VHDL support environment is the VHDL Design Library. The Design Library is a data structure that provides a single, installation-wide design repository shared by all users of the support environment. All of the tools communicate via the data in Design Library, as shown in the diagram of the VHDL Support Environment dataflow.

The Design Library holds internal representations of VHDL descriptions in a form called Intermediate VHDL Attributed Notation (IVAN 1076). IVAN is a well-defined, documented intermediate form that may be used by any design tool that manipulates VHDL designs. The library may be conceptually structured in a manner similar to a hierarchical directory structure on many operating systems.

Each tool communicates with the data in the Design Library through a piece of software called the Design Library Manager. This software component provides three sets of functions: a set of functions to allow the manipulation of the details of an individual IVAN representation, a set of functions organized around a network file system model incorporating relationships among entities, and a set of functions providing some commonly-used library access procedures at a somewhat higher level of abstraction.

The Design Library and Design Library Manager provide an open interface to the intermediate form representation of VHDL for design and synthesis tools which require access to that information.



VHDL System Description

VHDL Analyzer

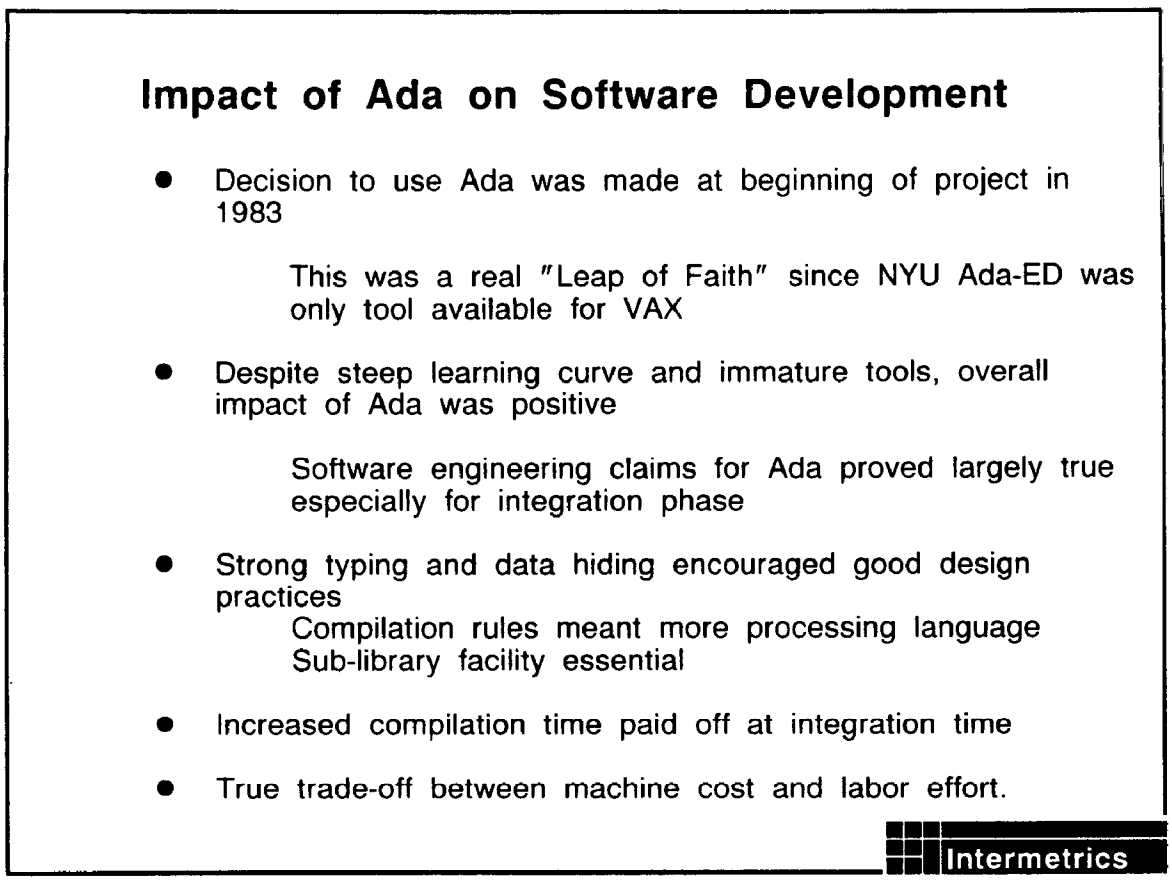
The Analyzer checks hardware descriptions for static errors - those that can be determined without simulating the passage of time. The Analyzer also translates the VHDL source text into IVAN intermediate representation. VHDL descriptions are prepared in Design Files which are simply host system text files. If the Analyzer determines that the syntax and static semantics of the VHDL input is correct, its IVAN representation is placed in the Design Library. Listings, cross references, error messages etc. are of course also produced at the user option.

VHDL Reverse Analyzer

The Reverse Analyzer reads the IVAN representation of a design unit in the Design Library and produces Design Files containing VHDL source text.

VHDL Simulator

The VHDL Simulator assists in the verification of hardware design by demonstrating how the hardware would behave if it were committed to silicon. The simulator supports the full modeling capability of the language, and permits mixed level simulation at various levels of abstraction. The simulator allows the specification of generics as parameters at execution of time thus allowing multiple experiments to be run without changing VHDL source or performing re-analysis of Design Units.



Impact of Ada on Software Development

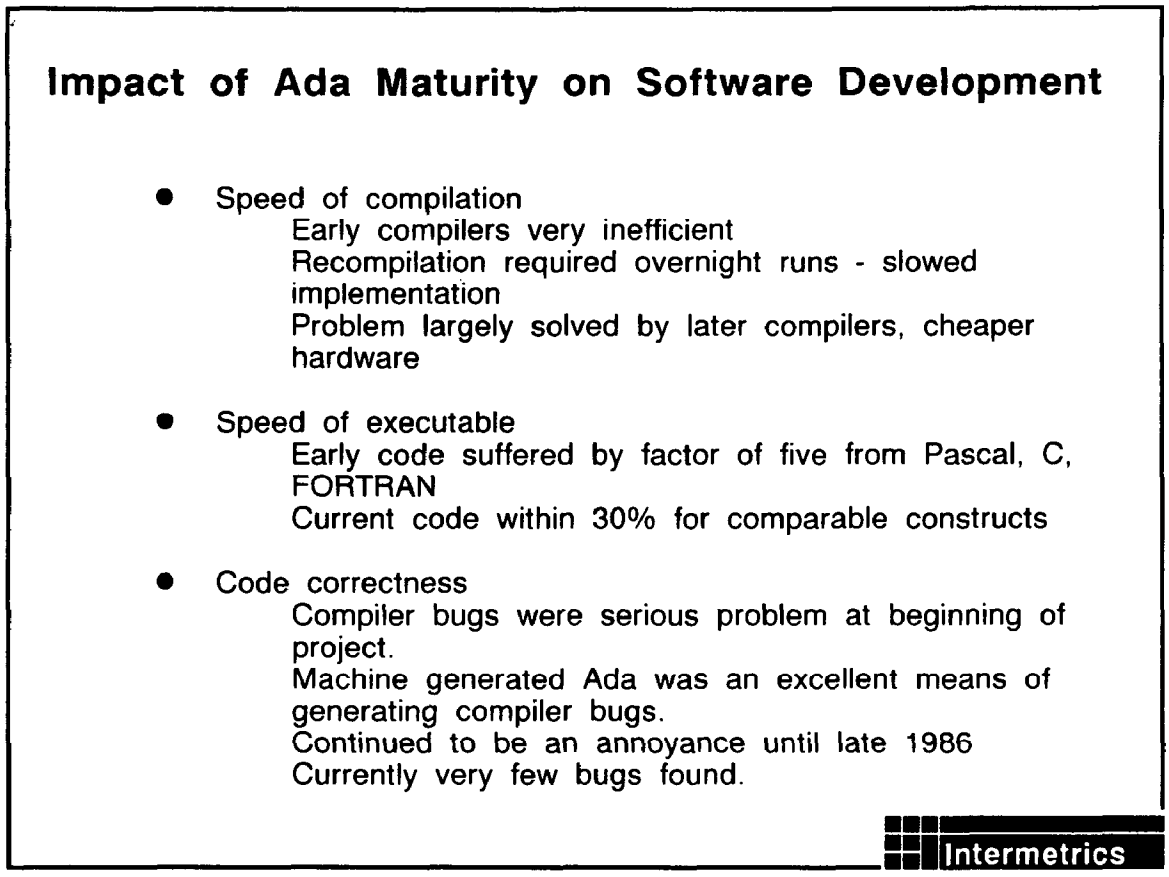
The decision to use Ada was made at beginning of project in 1983. This was a real "Leap of Faith" sine NYU Ada-ED was only tool available for VAX. During the first year of the project all avenues of obtaining Ada capabilities were explored and carefully followed. It was not until November 1984 (more than a year into the program) that a field test version of the DEC Ada compiler became available. This compiler was not released as a product until April 1985.

Intermetrics was fortunate to several have several very knowledgable Ada users and implementors on its staff to help in the training process for the engineers who would be building the VHDL software in Ada. Despite a steep learning curve and immature tools, the overall impact of Ada was positive.

Software engineering claims for Ada proved largely true especially for the integration phases. Strong typing and data hiding encouraged good design practises which made the complex software tractable to the many changes required by the fact the VHDL language was itself evolving.

Compilation rules meant more processing for changes compared to other languages. This fact made a sub-library facility essential to avoid excessive recompilation, especially in a multi-user environment. This important feature was also implemented as part of the VHDL Support Environment.

While compilation time and required machine resources were larger, system integration and test went more smoothly than with other languages. The net for this large, complex program was definitely a large gain. There was a true trade-off between cost of machine resources and labor effort. This trade-off favors the use of Ada as computer resources become cheaper and skilled computer engineers become relatively more expensive.

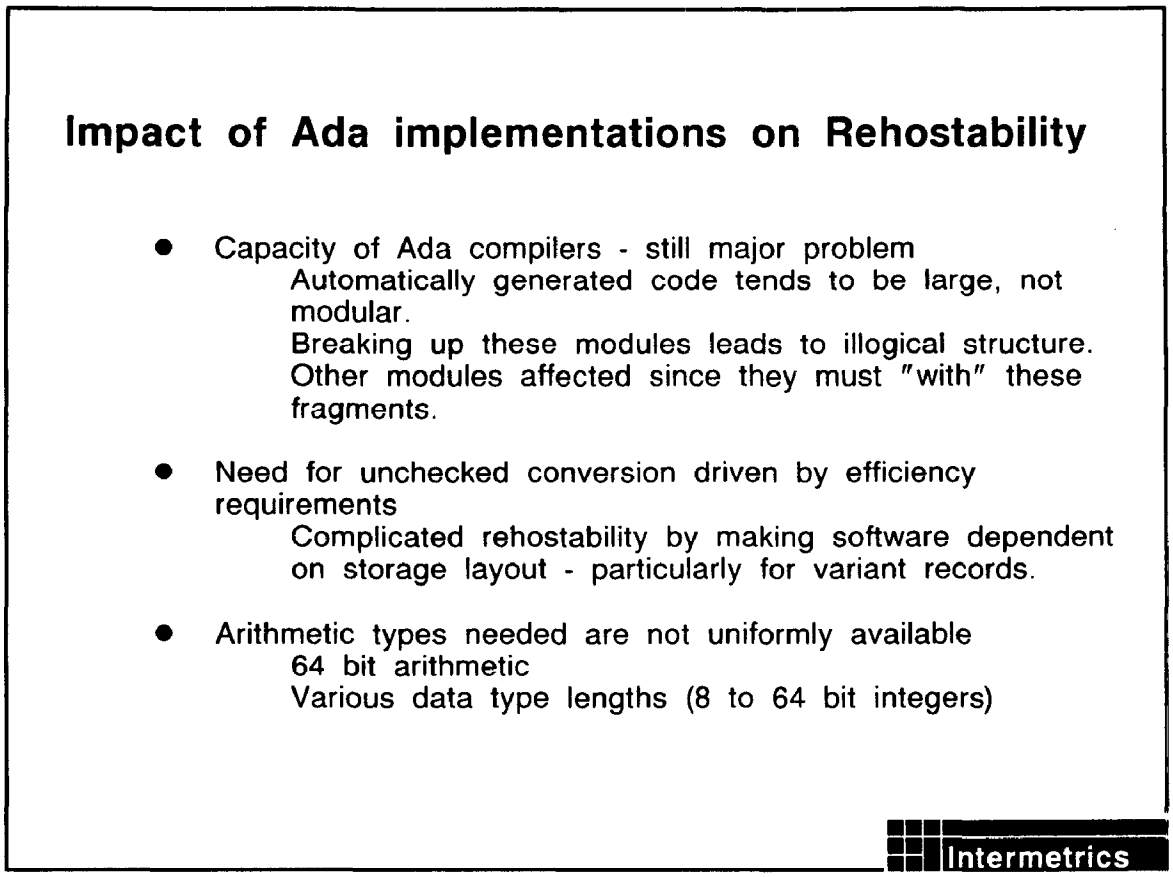


Impact of Ada Maturity on Software Development

The resource requirements of early Ada compilers were much larger than had been anticipated. This was particularly true of the virtual memory requirements needed to perform large compilations in reasonable time periods. This lack of resources made code changes very expensive in terms of schedule impact since small changes could result in and all-night recompilation. These problems have largely been solved by improvements in compilers and by the availability of relatively cheap large-capacity computers.

The quality of generated code has also improved greatly. Early code suffered by a factor of roughly five from average Pascal, C, and FORTRAN compilers. Currently our experience is that code is generally within 30% for comparable constructs and this has become a relatively unimportant issue.

As with most immature compilers, code correctness was a major problem. Adding to this problem was the lack of unanimity on Ada language interpretation. Cases where two compilers disagreed on the correctness of constructs was fairly frequent, especially when dealing with complex type definitions and uses. With experience and maturity of the Ada community, these problems are now quite rare.



Impact of Ada implementations on Rehostability

One of the original and important goals of Ada was to greatly increase the transportability of software. There are still several areas in which this goal has not been fully realized.

One of the most important practical problems in this area stems from the difference in capacity between compilers and library systems. The fact that a program will compile on one Ada system is not a guarantee that it will compile on another one. Capacity is also difficult to quantify because it is not necessarily measured by lines of code in a compilation unit. It is generally a function of several variables which may include expression complexity, name environment, type declaration complexity, and number of units in a host file.

This is particularly problematic for automatically generated code such as parse tables which tends to be large and not modular Breaking up these modules leads to illogical structures which are more difficult to maintain. The decomposition affects other modules since they must "with" these fragments. This tends to complicate configuration management for multiply hosted/targeted systems.