PROGRAMMING WITH ADA[1] -- THE ADA ENVIRONMENT

by

Erhard Ploedereder, Ph.D.

Tartan Laboratories

477 Melwood Ave.

Pittsburgh, PA 15221

U.S.A.

Summary

This paper examines two aspects of using Ada for the implementation of large program systems. First, those elements of the Ada language that are particularly targeted at programming in the large are discussed. Conclusions about appropriate design methodologies that match these language features are presented along with an explanation of some potential problems. Second, an overview of efforts to develop programming support environments for Ada beyond the Ada compilation system is given. The rationale and the scope of on-going standardization work in the area of Ada Programming Support Environments (APSE) is presented.

Preface

A necessary prerequisite for using Ada as the implementation language in a project is the availability of Ada compilers. At the end of 1985 more than 15 Ada compilers had been validated on a variety of host and target systems (1). Hence, at present, there is a significant number of Ada compilers available to prospective users of Ada. The focus for satisfying prerequisites for the successful application of Ada is gradually shifting to the software environments around the Ada compiler.

Given a high-quality compiler, the productivity of software designers and programmers depends to a very large degree on the properties of the software development environment. While this is true for any language, it has been specifically acknowledged in the Ada program: considerable efforts are spent to address the issues and improve the quality of software environments.

In this paper, we concentrate on two aspects of the software environment for Ada: first, we discuss some properties of Ada that impact environment issues and programming in the large. Then we present U.S. Department of Defense (DoD) and industry efforts towards evolving software environments in support of projects using Ada.

1. *Program Structure in Ada*

Historically, programming languages and their compilers supported the programmers by translating individual modules, but failed to enforce interface conventions across module boundaries. Violations of these conventions were discovered in part at link-time and in part during testing by exploring the causes of incorrect program execution. In some languages, e.g., Pascal, C, and Pearl, the missing facility for consistency checking across module boundaries was added in subsequent language revisions or by language extensions of individual compilers or by tools of the programming environment that analyze the interdependent modules for possible inconsistencies.

The Ada language design has been guided by established principles that facilitate programming in the large (2,3). It supports the structuring of program systems into units, each of which can be compiled separately from other units with which it interfaces. The mechanisms for enforcing the consistency among separately compiled units are an integral part of the language. The rules of the Ada language for consistency checks across compilation unit boundaries are as if all units involved were translated in one monolithic compilation; unit boundaries have no influence on the strength and nature of the checks. Thus, dividing large program

---

[1] Ada is a registered trademark of the United States Government, Ada Joint Program Office

systems into many units according to some structure induced by the applied design methodology carries no penalty in terms of the compiler support for the early detection of errors in the program.

In the Ada language, separately compilable units come in two flavors: A unit can be a specification unit, which describes only the programatic interface presented to users of the unit, or it can be a "body", which implements the details of an associated specification. Separately compilable specification units are called "library units" and their bodies are referred to as "secondary units", as they require the existence of primary specifications that describe their interface presented to other units.

Library units are individual packages, subprograms and tasks; such packages and subprograms may be generic, i.e., they describe an entire family of closely related packages and subprograms. Secondary units are the corresponding package bodies, subprogram bodies and task bodies. Packages are used to group logically related specifications, definitions and declarations. Specifications of packages, subprograms and tasks can also be nested within other units, and their bodies can be segregated into so-called "subunits" that can be separately compiled as secondary units.

Any specification unit can be viewed as a contract between the provider and the user of the services offered by the unit. The rules of the language guarantee that neither the user (i.e., another unit) nor the implementor (i.e., the body for the specification) of such services can violate the programatic interface described by their specification. Moreover, the details of the implementation are hidden from the user of the service, thus guaranteeing that the implementation can be changed without affecting the validity of the programatic interface presented to the user. In more technical terms, interfacing is allowed only to specification units, which contain all the information required for checking the consistency of references that cross unit boundaries. Additional features of the Ada language, such as private types, permit the hiding of data representation defined in the specification of a unit, so that users of the service cannot inappropriately take cognizance of representational details that are to be considered implementation-dependent (but generally are needed by the compiler in translating dependent units).

The described elements of the Ada language, which provide global structure in Ada programs, allow a variety of development strategies. Bottom-up development is supported in the sense that already implemented library units can be used to provide the building stones to construct additional library units and their bodies. Top-down development and stepwise refinement is supported in a dual fashion: first, the separation of specification and body for each library unit makes it possible to postpone the implementation of a unit and base all compilations of dependent units on the specification alone. Second, within secondary units, the implementation of locally declared packages, subprograms and tasks can be separated into subunits without impacting the capability to compile any units dependent on the units whose implementation is thus delayed. It becomes possible to code and compile programs whose underpinnings have been specified but not yet implemented.

Generally, the compilation of a unit cannot depend on a secondary unit. Exceptions to this rule are subunits, which depend on the secondary units within which their specification is provided, and compiler-introduced dependencies on secondary units. The latter may arise for the instantiation of generic units, if their bodies are expanded in place, for calls on inline subprograms, and for optimization-related reasons among units that are submitted in a single compilation.

Within the limitations of these exceptions, the clean separation of the implementation from the specification permits arbitrary replacements of secondary units to be made without affecting any dependent units, since such replacements must be in conformance with their respective specifications. This freedom has significant advantages for a top-down development, since defaulted bodies can be provided for units not yet implemented. With such defaults, the program can be brought to execution as long as the defaulted entities are not referenced in a way relevant for the results of the execution. Later, the defaulted bodies can be gradually replaced by their true implementation with a minimum of recompilation effort.

In practice, software development is often a mixture of top-down design and subsequent bottom-up implementation combined with corrections to the original design. The decribed features of the Ada language are ideally suited to support bottom-up and top-down strategies as well as a mixture of the two approaches.

2. *The Ada Program Library*

In order to achieve consistency checking across compilation boundaries, the compiler must retain information about the separately compiled units. This capability is provided by the use of the "program library" which contains the required information about compiled units.

The language rules require that each unit begins with a indication of all units on which it depends. This specification is called a "context clause". Based on the context clause, the compiler retrieves the stored information about the referenced specification units, uses this information for the compilation at hand, and, for each reference to an entity in such a unit, performs the required consistency checks the ensure the legality of the reference.

The concept of separate compilation requires that all units on which a given unit depends have been compiled prior to the compilation of that unit. Consequently, mutual dependencies of compilation units cannot be accommodated. Since dependencies generally exist only with respect to specification units, it is nevertheless possible that the implementation bodies of two units depend on the specification of the respective other unit. For example, it is not permissible that two specification units, A and B, reference each other; it is however possible that the body for A references the specification of B and vice versa. In terms of a design methodology, this restriction implies that mutually dependent definitions must be provided within a single package. It applies in particular to type definitions in package specifications, since establishing the representational details of types cannot be postponed to the compilation of the package body for reasons of compiler implementation constraints.

During software development, compilation units will be subject to changes, which may affect the validity of other compilation units that depend on the altered unit. Most programming languages leave this aspect of the software development process entirely up to the user or to tools provided by the language environment. The Ada language, in contrast, requires that the compilation system recognizes this potential danger and prevents the occurrence of inconsistencies introduced by a change. The mechanism for doing so is also embedded in the program library. It records the dependencies between compilation units and, upon recompilation of a unit, recognizes the fact that dependent units are now potentially inconsistent and may have to be recompiled.

A further application of the recording of dependency information is that the implementations of the program library are capable of retrieving the exact set of units needed to bring a given main program to execution. By transitively accumulating the units and their bodies on which the main program depends, any superfluous units contained in the library will be omitted from the linked image of the program.

With the requirement for the existence of a program library that tracks dependencies and the effects of changes, the Ada language goes beyond the nature of a mere implementation language. It attempts to address some of the problems that historically have been in the realm of software environment tools, i.e., version and configuration management tools. There can be no doubt that these rules of the Ada language will provide a maximum of safeguards against inadvertent errors in maintaining system consistency in the presence of changes.

The requirements posed by the language standard on the capabilities of the program library in this area are, however, rather rudimentary. They merely require that, after compilation of a unit, any previously compiled unit that is affected by the change to this unit must be treated as if it were as yet uncompiled.

An implementation of the Ada program library that satisfies only the minimal requirements imposed by the language standard is likely to create some difficulties for the users, unless a very rigorous programming discipline is enforced. The reason for such difficulties lies in the above rule which, in a unsophisticated implementation of the Ada library, will cause all compilations to obliterate the results of previous compilations of dependent units. Hence, a minimal change to and recompilation of a specification unit can cause many hours of additional recompilations for units that depend on the changed unit, even though the change may have had no real effect on them. The most pathological example of such a change is the addition of a comment to a specification unit. A compilation system that is not capable of recognizing the irrelevancy of this or other changes on the consistency of the compilation results for dependent units will have to require a recompilation of all dependent units. The undeniable advantage of strict enforcement of consistency by the recompilation rules can be quickly negated by the loss of productivity due to delays caused by such unnecessary recompilations.

As many Ada compilation systems will not possess the sophistication of analyzing changes with regard to their effects on dependent units, the methodology employed in the utilization of these systems for the development of Ada software must compensate for this lack. It must ensure that changes to specification units on which many other units transitively depend are minimized. The same holds, to a lesser extent, for changes to units that have subunits. Fortunately, this constraint is consistent with good software design and development practice that stabilizes central interfaces as soon as possible and performs a rigorous change control on these interfaces. Generally, bottom-up implementation strategies, following a detailed top-down design, will provide the best match with the properties of unsophisticated implementations of the Ada program library mechanisms.

The requirements on the Ada program library do not address the problem of parallel versions of units in multiple configurations of an encompassing program system. The solution to this problem, as it may be provided by environment tools, is somewhat constrained by the necessity to co-exist with the rules of the Ada program library. A typical scenario during software development is that new versions of some compilation units are installed for experimentation purposes. If these new versions have led to recompilations of dependent units, then reverting to the initial state after completion of the experiment is not possible without another recompilation of these dependent units. Short of an Ada program library that is fully integrated into a version and configuration control system, the only alternative for preserving the initial state consists in the creation of a new program library for each such experiment. It is therefore crucial that such creation of program libraries be possible with a minimum of effort and resources and a maximum of sharing with existing libraries. Again, the minimal requirements imposed by the Ada language standard do not address this issue.

Finally, it should be noted that the Ada language rules regarding the program library refer only to the results of compilations, but not to the input to these compilations. That is, the program library need not administrate the Ada source files. It is merely concerned with the internal representation of the compilation units as required by the separate compilation capability, and with the generated object code. It is not necessarily a source control system, nor is it required to support dependencies other than compilation dependencies, such as for example the interrelation between source code and documentation files.

It is entirely possible that the productivity of Ada software implementors will be influenced by the quality and functionality of the Ada program library at least as much as by the quality and speed of the Ada compiler itself. It is therefore of utmost importance that the software design and development methodology and the strategies for version and configuration control be in unison with the capabilities of the employed Ada program library, and that the program library mechanisms integrate well with software environment tools beyond the Ada compiler.

Despite all the caveats expressed in the preceding paragraphs, a reasonably sophisticated implementation of the Ada program library that supports functionality beyond the minimal requirements imposed by the Ada language standard can be an extremely powerful tool. In large application systems, errors that are caused by minor interface inconsistencies are very difficult to locate. The required capabilities of the program library prevent the occurrence of this class of errors. With only a moderate addition of functionality to the Ada program library, tedious and traditionally error-prone tasks, such as the recompilation of changed sources and ensuing recompilation of dependent but unchanged sources, can be totally automated and performed without errors, since the required information is directly derived by the compiler from dependency information in the program library.

3. Programming Support Environments

For software engineers and programmers to be effective, the provision of an Ada compilation system alone is clearly not sufficient. They require additional tools, such as editors, debuggers, version and configuration management tools, network file-transfer tools, project management tools, and so on. The collection of these tools is generally referred to as a "Programming Support Environment (PSE)".

Some of the tools in a PSE are heavily language-dependent, for example the compilers, syntax-directed editors, program analyzers, symbolic debuggers, performance monitoring tools, etc. These tools need to be developed for any implementation language. Other tools are language-independent, such as file-transfer mechanisms, project administration tools, documentation systems, test harnesses, etc. Where available, these tools can be applied in a project regardless of the chosen implementation language.

Since the early stages of the Ada effort, considerable attention has been focused on the PSE for Ada and, in the process, on issues of language-dependent and -independent environment support in general, because the state of the art in this area is in its infancy despite its recognized importance. Unfortunately, this attention has led some observers to the misconception that Ada requires substantially more environment support than other languages or, worse, cannot be used at all without a complete environment specifically developed for Ada.

In reality, Ada has been used as a focal point of plans for improvements in software engineering and PSE technology, which are direly needed regardless of the choice of implementation language. It could even be argued that Ada may require less environment support than other languages, due to its high degree of compile-time error checking, its enforcement of implementation discipline, and the environmental aspects of the program library. There is certainly little reason to believe that Ada could not be supported in more traditional environment settings. Currently, the majority of commercially available Ada compilation systems are not embedded in an Ada-specific PSE, but are integrated into the standard PSE available on the respective host systems.

3.1. The DoD Requirement Catalogues

In early 1978, first efforts were made within U.S. DoD to arrive at a concept for the development of the environment support for Ada. A set of initial ideas were first collected in what became known as the SANDMAN catalogue, which was never published. By late 1978, it was consolidated into a requirement catalogue, named PEBBLEMAN, in which the desired functionality and cooperation of various tools were described. A revised version of PEBBLEMAN was published in 1979 (4). In February 1980, a further document, STONEMAN, was produced; it became one of the most cited references regarding Ada environments (5).

In addition to posing requirements for the desired tools in a PSE, STONEMAN introduced a model for "Ada Programming Support Environments (APSE)", which addressed both the issues of tool integration and of portability of the entire APSE as well as of individual tools or tool-sets. The term "APSE" has since become a synonym for the concept of integrated toolsets for Ada, as opposed to the so-called "tool-box" approach in which a set of independent tools is provided to the user.

Part of the motivation behind the STONEMAN model was the recognition that the general immaturity of current PSE implementations is not caused so much by the non-existence of powerful tools as by the missing capability to bring existing tools together on a single host system and integrate them with each other to form a coherent and efficient PSE. Consequently, the development of a PSE becomes unnecessarily expensive, as many tools are reimplemented although similar and, quite possibly, better tools are already available in other environments.

One might surmise that the enhanced portability of tools written in Ada would solve the portability problem. However, while the Ada language standard provides portability advantages in many areas, it must be recognized that most tools require a significant amount of interfacing with the host operating system. The Ada standard, which is primarily intended for the generation of application code for arbitrary target systems, could not justifiably prescribe the details of those language features that are intimately linked to operating system interfaces. Moreover, the requirements of application code on operating system services have been recognized to be quite different from those of PSE. This is due partially to the different problem domains and partially to the dynamic nature of evolving PSE as opposed to the relatively static nature of operational application systems.

Consequently, some standard interfaces suitable for application programming might be quite inappropriate for PSE development. Given that the Ada language leaves the details of operating system interfaces, such as calls on the file management or on terminal IO services, largely implementation-dependent, the porting of tools written in Ada will nevertheless have to contend with modifications of these host-dependent portions of the software.

The STONEMAN model postulates a system architecture in which the host dependencies and the tool intercommunication are encapsulated in a Kernel APSE (KAPSE). The individual tools of the APSE are built on top of the KAPSE services. Since the interfaces offered by the KAPSE are conceived to be host-independent, and since the Ada language goes to great length in facilitating the portability of Ada software, tools could be written in Ada to be portable among different hosts offering the same KAPSE services. Port-

ing of an entire APSE to a new host consists of re-implementing the KAPSE on the new host. Furthermore, individual APSE tools could be transferred to another APSE as long as the tool intercommunication interfaces needed by these tools were provided by this APSE. Some tools are so heavily dependent on interfaces with other tools that it is unreasonable to expect that all such interfaces are provided on each KAPSE. These tools cannot be ported individually; they form an tightly integrated tool-set. By relying only on the common mechanisms for tool communication provided by the KAPSE, but not on the specific details of the communicated information, such combined tool-sets could equally be ported to a new APSE.

STONEMAN established a set of requirements for the services that must be provided by the KAPSE in order to reach the described goals. These requirements relate in particular to the data administration and communication interfaces needed by tools, and to the run-time system that enables the execution of Ada programs. STONEMAN adopts the paradigm of distinguishing host and target systems. It postulates that software development, in particular for embedded targets, needs to take place on a host system sufficiently powerful to accommodate the various tools required for supporting the development and maintenance of software throughout its life-cycle.

Among the many conceivable tools of an APSE, STONEMAN identifies a minimal set perceived to be necessary to make an APSE an effective tool for software developers and maintainers. This set was designated as the Minimal APSE (MAPSE). It comprises the Ada compilers, linkers and loaders, simple static program analyzers and debuggers, text editors and pretty-printers, a file and configuration management system, and the command interpreter.

The STONEMAN principles were readily accepted by the majority of efforts concerned with the development of a PSE for Ada. In particular, the emphasis of STONEMAN on the provision of better data administration capabilities than offered by the file management of traditional operating systems has been reflected in almost all major PSE developments. While some of the details of the STONEMAN requirement catalogue may require revisions in the light of experience gained since 1980, the fundamental principles are still valid contributions to the area of PSE design.

3.2. APSE Implementations

Within U.S. DoD, two major developments of Ada Programming Support Environments were procured: In 1980, the U.S. Army initiated the development of the Ada Language System (ALS) with SofTech, Inc., as the lead contractor (6); in 1981/82 the U.S. Air Force contracted with Intermetrics, Inc., for the development of the Ada Integrated Environment (AIE) (7,8). Initial plans called for the ALS to be an interim tool-set for the introduction of Ada, while the AIE was intended to be a STONEMAN-conforming integrated environment and to eventually become a standard DoD Ada environment.

The developers of the ALS adopted many of the STONEMAN principles in their implementation strategy, such as utilization of a KAPSE-like kernel to facilitate the porting of the ALS, which has been developed on a VAX/VMS host system. In December 1983, the first version of the ALS was made available to prospective users. The self-hosted Ada compiler of the ALS was validated in December 1984. The ALS comprises a set of about 75 tools to be used in software development and maintenance. The development of the AIE, whose initial design promised a superior integrated environment, encountered serious funding problems; at present, it is extremely doubtful that a comprehensive AIE will become available in the foreseeable future.

The U.S. Navy decided in early 1985 to base their Ada environment efforts on the ALS and to enhance this environment by additional tools and by code generation capabilities for the prevalent instruction set architectures in use by the Navy. This extended ALS has been named "ALS/N" and is expected to become available in early 1989.

The German Ministry of Defense, Bundesamt fuer Wehrtechnik und Beschaffung, began the procurement of components of an Ada software environment, named SPERBER (Standardisiertes Programm-Erstellungssystem fuer den Ruestungsbereich) in 1979 (9). Major efforts have been directed at developing Ada compilers, debuggers, and a program development data base system. The first two compilers were validated in November 1984. The British Ministry of Defense in cooperation with British industry co-financed several design efforts towards the development of software environments for Ada (10, 11). The Commission of European Communities, under its multi-annual program to advance the European software technology in the commercial sector as well as under the ESPRIT program, has co-financed several multi-national projects developing Ada programming support environments, most notably the PAPS project (11).

Commercial suppliers have been somewhat reluctant to embark on a course of providing integrated Ada Programming Support Environments, recognizing that such integrated solutions, while desirable in principle, constitute a truly major capital investment. Moreover, customers who have built a considerable wealth of software to support their in-house software development are concerned with preserving the usefulness of this software for future development projects using Ada as the implementation language. The challenge to commercial suppliers is to provide integrated environments that nevertheless allow the inclusion or easy transitioning of existing tools.

Apart from ALS, which is also commercially marketed by SofTech, Inc., only the Ada Development Environment (ADE), marketed by ROLM Corporation (12), and the Rational Environment, marketed by Rational, can be regarded as largely integrated Ada Programming Support Environments. The Rational Environment takes a quite unique approach: its host system has been specifically designed for the development and execution of Ada programs. The entire environment is exclusively centered around Ada. Many of the language concepts are immediately reflected in the concepts of the environment whose command language is Ada. Other commercial suppliers of Ada compilation systems have taken the path of embedding their systems into the environments offered by existing operating systems, so that users could continue to utilize the tools which which they are most familiar.

4. *Environment Standardization Efforts*

The more the software development process is assisted by tools of a programming environment, the more difficult it becomes to transition software developed in one environment to a different environment. The current practice in military procurement of mission-critical software is that this software is developed by contractors but maintained in military maintenance centers. Just as a proliferation of implementation languages raises the cost of such maintenance considerably, so does a proliferation of software environments needed to maintain the software even in a single language. It is therefore in the interest of DoD to minimize the proliferation of environments in its maintenance centers.

Here, DoD is faced with a dilemma: while language research was advanced enough to embark on the standardization of a single language, Ada, today's state of the art in environments is much less mature. Therefore rigorous standardization on a single environment may be ill advised, as considerable advances in the environment technology can be expected to occur in the next decades.

A compromise can be found within the framework of the STONEMAN model. If commonality of KAPSE implementations were advanced by standardization at this much lower and less ambitious level, tools used in application development could be transitioned into existing maintenance environments, thus reducing the number of necessary environments while, at the same time, continuously enhancing the support provided of these environments.

4.1. The Kapse Interface Team (KIT)

When it became apparent that there would be two competing designs of KAPSE interfaces for the ALS and the AIE respectively, a Memorandum of Agreement was signed in January 1982 between the U.S. Army, Air Force, and Navy to work towards establishing commonality of the KAPSE interfaces in DoD environments (13). Under the lead of the U.S. Navy, the KAPSE Interface Team (KIT) was created and chartered with this task. The KIT is assisted by an advisory group of software environment experts from industry and academia with international representation, the KITIA (KIT - Industry and Academia). The objective of KIT/KITIA was set to establish requirements for the interoperability and transportability of tools among APSE and to subsequently develop guidelines and conventions for achieving these requirements with the ultimate goal of evolving standards in this area (14).

KIT/KITIA has been meeting quarterly since 1982. The results of its deliberations are contained in periodically published reports (14). Among its most relevant products are the "Requirements and Design Criteria for the Common APSE Interface Set (CAIS)" (15), and the proposed "Military Standard Common APSE Interface Set" (16), which is an initial set of KAPSE-like interfaces for the encapsulation of host dependencies. At this time, the later document is being reviewed by the DoD services for adoption as a military standard within DoD.

While the initial motivation of achieving commonality between ALS and AIE has decreased, due to the lessening importance of the AIE, the KIT/KITIA effort has been recognized as an important contribution to ad-

vance the knowledge in the area of KAPSE-like interfaces and of issues of tool portability and interoperability. It also creates a forum for information exchange between DoD and industrial efforts, thereby preserving the opportunity to prevent a complete divergence of these efforts.

4.2. The Common APSE Interface Set (CAIS)

In late 1982, a KIT working group began examining the ALS and AIE KAPSE interfaces for areas of commonality and of divergence in order to establish a common set of interfaces that could be supported by both KAPSE designs. In March 1983, this group was joined by several KITIA members to form a group that later became known as the CAISWG (CAIS working group). With the participation of the ALS and AIE designers, this group began developing the specification for a set of common interfaces deemed important for the portability of tools. A crucial design decision was made in mid-1983 to pursue an interface set that was not constrained by the current environment efforts of ALS and AIE, although much of the practical experience of these and other similar efforts influenced the choice of interfaces.

The main goals of the CAISWG were to develop interfaces based on a simple, yet powerful and extendible, model, to apply uniform concepts throughout the design of the interfaces, and to cover those interfaces that are most crucial for the portability of many tools. A first public review of the concepts of the CAIS took place in September 1983. Various revisions were produced and publicly reviewed until, in January 1985, the final document was delivered to AJPO as a proposed military standard.

The CAIS in its present form contains interfaces for the administration of files, their interrelations and attributes, for process administration, and for terminal IO targeted at three standard kinds of terminals. Some other utilities frequently used by tools are also present. The CAIS design for the file administration has departed from the traditional view of hierarchical file systems and instead administers files by their interrelations. In this regard, the CAIS cautiously joins a trend that has been apparent in almost all recent designs of PSE. Special attention has been paid to security aspects, so that the CAIS would not be in conflict with requirements posed by DoD (17). The concepts and the set of interfaces provided by the CAIS are open-ended. It is expected that additional interfaces will be added to the CAIS and that the existing global concepts are sufficiently flexible to accommodate a large variety of such extensions.

It was realized that the work of the CAISWG could be only a first step in defining a basic set of uniform interfaces. The U.S. Navy contracted with SofTech, Inc., in 1986 for the further enhancement of the CAIS beyond this initial set and for a pilot-implementation of the CAIS to validate its usefulness and efficiency. Other pilot-implementations of major portions of the CAIS have been undertaken by TRW, Inc., under government contract, and by Gould, Inc. and MITRE Corporation, as internally financed projects.

5. Future Benefits of APSE

As various studies have shown, the demand for application software is rising exponentially over time, while the workforce engaged in producing this software is growing at a slow linear rate. Already the demand is far beyond the production capability. In addition, the complexity of the software has increased considerably, making the production of quality software more and more difficult and time-consuming. Without substantial improvements in the software development process, industry will not be capable to meet the increasing demand nor will it master the growing complexity of this process in the future.

Four factors can have a major impact on ameliorating the current situation:

- Application of better methodologies: Ada is a first step to improve the methodological basis at the implementation level. Clearly, better methodologies or increased application of already available methodical approaches for requirement analysis and specification are necessary as well. Almost certainly, these approaches will be supported by software tool-sets. APSE can provide the vehicle for a wider penetration of these tool-sets and associated methodologies.

- Provision of better tools: Today the portability and integration problems of tools lead to a deadlock situation. For lack of portability, the development of truly sophisticated tools is expensive and commercially risky; it therefore remains largely in the realm of proto-types developed in academia. For lack of commercially available tools, software developers are forced to expend their resources in the duplicating development of tool support, which, because of the resulting financial constraints, continues to be of low quality. Improvements in portability and integration, as possibly provided among APSE, can release resources for the development of more advanced

and better tools as well as widen the commercial market for such tools.

; Use of standard components: similar to the hardware manufacturers, software producers will increasingly have to rely on reusing pre-fabricated components in order to meet the rapidly expanding volume of the software demand. Within the framework of an application-oriented implementation language, whose major goal is the enhancement of software portability, Ada is an almost ideal breeding ground for such standard components.

; Automated generation of customized components: a leading edge in hardware research is concentrating on the development of systems for the fast production of customized components. A similar approach can be expected to emerge in the software field as well. In fact, in some specialized areas, e.g., in compiler construction, the automated generation of components that are customized to particular languages or target architectures has already been successfully applied in commercial products. Again, APSE may provide the means for a wide distribution of such software-generating tools.

Of all these factors, the last two are likely to be the most important ones. As long as application software is developed from scratch, whatever improvements in tool support can be provided to the software developers will increase their productivity only by some constant multiplying factor. Significant as this factor may be, it will not be sufficient to catch up with the exponentially developing software demand in the long run. The only hope to match the ascending curve of demand consists in an ever increasing reduction of the amount of work to be performed to produce products. Only the utilization and increasing availability of standard components or of automated generation of customized components can lead to this reduction. Ada and APSE can play a major role in meeting these challenges, in particular, if the promise of increased portability of tools and software components among partially standardized APSE can be realized.

6. *References*

(1) Ada Information Clearinghouse, "Validated Ada Compilers", AdaIC, 3D139 (1211 Fern St., C-107), The Pentagon, Washington, November 1983.

(2) U.S. Government, "Reference Manual for the Ada Programming Language", ANSI/MIL-STD-1815A, U.S. Government Printing Office, Washington, February 1983

(3) Honeywell, Alsys, "Rationale for the Design of the Ada Programming Language", Draft for editorial review, Honeywell, Minneapolis, January 1984

(4) United States Department of Defense, "PEBBLEMAN revised - Requirements for the Programming Environment for the Common High Order Language ", January 1979

(5) United States Department of Defense, "Requirements for Ada Programming Support Environments - STONEMAN", February 1980

(6) SofTech, Inc., "ALS Specification", November 1983

(7) Intermetrics, Inc.,"System Specification for Ada Integrated Environment", November 1982

(8) Bray, G., "AIE Support for Management of Embedded Computer Projects", Ada Letters, Vol. II, Number 1, pp. 33-49, August 1982

(9) Ploedereder, Erhard, "Project SPERBER - Background, Status, Future Plans", Ada Letters, Vol. III, Number 4, pp. 92-98, February 1984

(10) Department of Industry, "U.K. Ada Study, Final Technical Report", June 1981.

(11) Bevan, S. et al. "Investigation into the Differences Between PAPS and M-Chapse", unpublished paper, December 1983.

(12) Elliott J.K., Klein D.M., Williams J.S., "APSE Tools - ROLM's experience", in: Teller J. (ed.) "Proceedings of the Third Joint Ada Europe/AdaTEC Conference", Brussels, 26-28 June 1984, Cambridge University Press, 1984.

(13) Memorandum of Agreement, published in (14), Volume III, pp 3B 18-19

(14) Patricia Oberndorf, "Kernel Ada Programming Support Environment (KAPSE) Interface Team Public Report", Naval Ocean Systems Center, San Diego.

Volume I, NOSC Report TD-209, NTIS AD A115 590, April 1982
Volume II, NOSC Report TD-552, NTIS AD A123 136, October 1982
Volume III, NOSC Report TD-552, NTIS AD A141 576, October 1983
Volume IV, NOSC Report TD-552, April 1984
Volume V, NOSC Report TD-552, August 1985

(15) KIT/KITIA, "DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS)", prepared by KIT/KITIA for the Ada Joint Program Office, September 1985

(16) U.S. Government, Ada Joint Program Office, "Military Standard Common APSE Interface Set (CAIS)", Proposed MIL-STD-CAIS, NTIS AD 157-587, January 1985

(17) United States Department of Defense,"Trusted Computer System Evaluation Criteria", Computer Security Center, Fort Meade, Maryland, CSC-STD-001-83, August 1983.