The importance of Ada programming support environments

by THOMAS A. STANDISH
University of California
Irvine, California

ABSTRACT

In this paper it is argued that even if we assume the most optimistic scenario we can think up for the introduction of the Ada* language, the language alone, in the absence of an Ada Programming Support Environment (APSE), is insufficient to achieve the gains in programming productivity and software reliability with which use of Ada tantalizes us.

Moreover, it is argued that the level of support envisaged in the Minimal Ada Programming Support Environment (MAPSE), specified in the STONEMAN, which provides a rudimentary level of capability incorporating a text editor, compiler, linker/loader, and symbolic debugger, is also insufficient; and that it is time to seize the opportunity to conceptualize what sort of advanced programming support tools should populate a mature APSE of high utility and effectiveness. In this context, consideration of support tools for software project management, interactive programming, modern programming practices, software reuse, and improved program understanding techniques arises.

*Ada is a trademark of the United States Department of Defense (OUSDREAJPO).

WHY THE ADA LANGUAGE NEEDS AN APSE

Suppose that the most optimistic scenario we can dream up for the introduction of the Ada language actually comes to pass. Will this be enough for us to reap the benefits we are hoping Ada can provide?

In this paper, it shall be argued that the Ada language, considered as an isolated tool, cannot solve all of the problems of reliability, performance, and productivity that must be addressed if Ada is to succeed in realizing the high hopes some have for it. Rather, it is argued that Ada must be buttressed by powerful programming support environments that provide the means for performing a variety of essential tasks lying beyond the reach of programming languages.

Defining an Ada Programming Support Environment (APSE) as the collection of tools, resources, procedures, and policies that support the development, repair, and upgrade of Ada software, it is argued that even more than the rudimentary level of capability envisaged in the STONEMAN's Minimal Ada Programming Support Environment (MAPSE) is necessary if we are to achieve the significantly improved levels of programming productivity and software quality with which use of Ada beckons us.

Thus, the central question that the paper addresses is: could the Ada environment be even more important than the Ada language in helping to achieve the benefits we seek from the use of Ada?

First, for the purposes of setting an appropriate context for the subsequent discussion, let's conjure an optimistic scenario for successful introduction of the Ada language.

An Optimistic Scenario

Our optimistic scenario for the introduction of Ada consists of the assumption that we succeed in accomplishing the following steps:

1. There would have to be success in establishing precise, comprehensible standards for the definition of Ada. If we don't know what Ada means, we can't write certified compilers which implement the common meaning and provide a framework for exchanging Ada programs.
2. There would have to be success in writing certifiable Ada compilers which produce efficient, reliable running programs. (Note: the author is not so naive as to be unaware that the risk that we might fail in getting this far isn't entirely trivial; but the rest of the paper would be rather uninteresting if we don't assume we can reach at least a state of affairs where we have certified Ada compilers running on many computers of reasonable size. That is, if we lose the game by failing to define a standard and by failing to implement certified compilers that achieve performance and reliability, there is no point in discussing what must follow for the whole game to be won.)
3. If certified Ada compilers run standard Ada on most machines of reasonable size, the best we can hope for is that a framework will be established permitting a flourishing commerce of Ada programs. (Few people these days are so ill-informed as to think that having a precisely defined and implemented standard programming language will solve the program portability problem 100%. After all, programs are written containing dependencies on operating system calls, device characteristics, and other interface requirements that lie outside the scope of definition of a programming language. Nonetheless certification, standardization, and widespread implementation of Ada compilers could help reduce the cost of program transfer, since machine dependencies could be isolated in Ada packages with invariant external interfaces, and since some machine dependencies could be expressed using Ada representation specifications. This tells us that we might be able to reduce the cost of program transfer with Ada to a point at which it costs less to do it with Ada than with other approaches.)

To continue by optimistic speculation, if Ada becomes widespread in use, begins to function as a medium of exchange, and opens up a substantial market for the sale and exchange of programs, what response might one assume from the free enterprise system?

Perceiving that a wide market will exist for sales, here is a list of possibilities: (a) computer manufacturers would develop certified Ada compilers for new computers; (b) software firms would develop and sell Ada programming support tools; (c) publishers would publish books and educational materials on Ada; (d) educators would introduce and teach Ada in programming courses (perceiving that Ada, in addition to being popular and useful, could be a good carrier of modern programming principles); and (e) enterprises could get Ada programmers, Ada compilers, and Ada programming support tools in the marketplace and could import and export Ada programs.

To complete the optimistic scenario, we assume that some (but not necessarily all) of these economic consequences of the introduction of Ada take place.

Ada is Still Not Enough!

Even if an optimistic scenario such as this comes to pass, it is argued that the Ada language alone is not enough. More is needed. Here's why.

Wonderful as they are, programming languages play only a

small role in the software life cycle. It is estimated, for instance, that in software projects of substantial size, coding the design in a programming language accounts for only 15% of the total pre-release cost, and that the total pre-release cost may be only 10–30% of the total life cycle cost.

Furthermore, many activities in the software life cycle are supported by tools, procedures, or policies that are not directly connected with the programming language(s) employed by the project.

For instance, software project managers devise project schedules; manpower loading plans; milestone charts; and budgets for machine cycles, memory, and monetary resources. They monitor tasks on the critical path, report on progress and resource consumption, incrementally shift resources in response to perceived needs, and oversee the hiring and training of new project personnel. Recent evidence[1] suggests that upward of 80% of software project failures are software management failures as opposed to technical failures. Few if any of these management activities depend in any essential way on the choice of the programming language.

The system requirements and the system design may be expressed in natural language or in design representation notations separate from the programming language; and activities such as requirements tracing and design reviews may take place using notations, language, and procedures entirely separate from those given by the programming language.

Maintenance of current system documentation, module test sets, test completion status, and system configurations may depend more on database, word processing, and file system tools than on the programming language or on programming language support tools.

Let's take a glimpse at some software economics, for a moment, to try to establish a framework in which we can discuss the relative importance of trying to introduce various kinds of support tools and policies into a programming environment.

In the first place, the demand for computer instructions appears to be increasing rapaciously, and a serious shortfall of programmers to produce them exists in relation to demand.

For instance, the number of instructions NASA used to support the Mercury, Gemini, Apollo, and Space Shuttle programs has been growing at 24–25% per year for a couple of decades. While Gemini support took 1 million support instructions, and Apollo took 10 million, the Space Shuttle now takes 40 million. Most of the Space Shuttle instructions support ground launch and pre-launch check-out procedures and were designed to avoid the necessity of employing a ground launch support crew of 20,000 people. What is true for NASA appears to be true for the economy in general; namely, automation is being employed to avoid inefficient, labor-intensive production, and computers are being used to enhance product versatility and market appeal. Thus, the overall demand for computer instructions appears to be increasing in the neighborhood of 10% per year in many industries, and the national demand for computer instructions may, in general, be growing somewhere near 20% per year.

However, the supply of programmers is increasing perhaps only in the neighborhood of 5% per year, and programmer productivity has been falling! During the 1960s when high-level languages were replacing assembly languages, programmer output (in delivered instructions per person year) was increasing at perhaps 8–11% per year; but recently, the annual increase has been estimated to be in the range of 4–5% per year.

In short, given the poor prospects for increasing the output of new programmers from the educational system, there may be no alternative but to increase software productivity if the demand for production of computer instructions is to be met and if the shortfall in programmers will be a condition we will have to live with.

How then do we address the problem of increasing productivity? One approach is to analyze the cost drivers that correlate with the cost of software projects. In his new book, *Software Engineering Economics*[2] Barry Boehm introduces the COnstructive COst MOdel (COCOMO), which is a good fit to a database of measurements on 63 software projects spanning a range of different application areas. Briefly, one starts with a baseline estimation formula such as

$$MM = 2.4(KDSI)^{**}1.05$$

giving an initial unadjusted estimate of the number of man-months (MM) to complete a project as a function of the number of thousands of delivered source instructions (KDSI), and one multiplies by coefficients that determine whether the estimated man-months will increase or decrease as a function of measurable software project characteristics (which can be thought of as cost-drivers). The ratio between the best increase and worst decrease in productivity for each cost-driver forms a *productivity range*. Examples of such productivity ranges are as follows:

1.20—programming language experience
1.32—turn-around time
1.49—software tools
1.51—modern programming practices
1.57—applications experience
2.36—product complexity
4.18—personnel/team capability

Software Productivity Range
(from cover of Boehm[2])

Some of these cost-drivers are controllables. That is, by investing to provide software project resources or by following certain project disciplines, we can control factors that enhance productivity.

For instance, we could invest in good programming support equipment to give programmers excellent turn-around time. We could provide good software tools. We could train programmers to use the programming language well. We could adopt modern programming practices as a software project discipline, and we could attempt to select programmers with proven track records and applications experience, if possible. The cumulative effect of these measures on productivity could be very dramatic (e.g., factors of 4, 8, or 10 could be achieved using the short list of measures just given), and these could easily dwarf any effects of choosing to use Ada or not.

In summary, we see that software productivity may depend heavily on the characteristics of the environment employed

and not so heavily on the characteristics of the programming language employed.

What is true for software productivity may be true to a lesser but still significant extent for software reliability and software performance.

Software performance will obviously be influenced critically by whether or not it is possible to compile Ada source programs into compact, fast-running object programs. However, performance may also be influenced critically by whether or not the Ada Programming Support Environment provides effective tools to perform measurement and optimization. Frequently, upwards of 90% of the execution costs are attributable to 7 to 10% of the code. Identifying and optimizing the critical sections has been found to be an effective way to improve performance. If the name of the game is measurement and tuning, both the programming support environment and the compiler must work together to provide the solution, with the environment furnishing performance measurement tools and the compiler providing optimizations. Where compiler optimizations are insufficient, the environment may make the key difference by providing source-to-source program improvement transformations or by making manual program rewriting more manageable and systematic.

Where software reliability is concerned, the programming language can play a key role in promoting reliability. Proponents of Ada have argued that Ada will promote reliability because it supports clean module interfaces and information hiding (through packages) and because it permits clear expression of control and data (through exceptions, tasking, and an extensive data type system). Opponents have argued that Ada programs may not be reliable because the language is too complex or may have ill-defined interactions between its features. But we have all known reliable programs written in unreliable languages and unreliable programs written in reliable languages. Promoting reliability may have more to do with assuring clean designs and thorough testing than with the characteristics of the language in which the program is written. It is the environment, not the language, which must provide tools and disciplines to perform design, design review, and testing.

Thus, it could be that the reliability of Ada programs will be more dependent on the characteristics of the Ada Programming Support Environment and the programmers who write them than on the characteristics of Ada itself. In summary, the success of Ada in promoting software reliability, performance, and productivity depends critically on the characteristics of APSEs. While Ada is clearly *necessary* for success, even under the most optimistic scenario, Ada alone is *insufficient.*

A BRIEF VIEW OF THE STONEMAN PHILOSOPHY

The STONEMAN[3] requirements document for APSEs specifies three levels of structure: (a) a KAPSE or Kernel Ada Programming Support Environment, (b) a MAPSE or Minimal Ada Programming Support Environment, and (c) the APSE itself. In a nutshell, the KAPSE provides basic operating system services and database capabilities and is intended to provide a machine-independent set of kernel services on which APSEs may be built. The MAPSE provides minimal Ada programming support services such as: (a) a text editor, (b) an Ada compiler, (c) a linker/loader, (d) an Ada debugger, and (e) a command language interface (for logging on, calling tools, manipulating files, etc.).

The STONEMAN philosophy, expressed in its so-called "strategy for advancement," envisages that the KAPSE can be implemented as a standard operating system kernel on many machines to provide a standard foundation for APSEs. If the KAPSE could be standardized and expressed as an Ada package, then all the KAPSE services and capabilities could be made available to Ada programs, and a major deterrent to program portability could be overcome.

The MAPSE, if successful, could provide a means for using Ada as a systems programming language for implementing not only Ada applications programs, but also the tools that constitute the full APSE. Thus, one could get going by supplying a rudimentary Ada programming environment (the MAPSE), and one could bootstrap out of the rudimentary environment into an advanced APSE by using Ada as the systems programming language for populating the APSE with environment tools. Such tools could be compiled, loaded, and run on top of the MAPSE to form a highly portable APSE.

APSE tools would thus be supplied in an Ada library available as a companion to the MAPSE. Current design efforts (the Army's ALS or *Ada Language System* and the USAF's AIE or *Ada Integrated Environment*) focus on providing the MAPSE-level capability specified in the STONEMAN but do not call for design of full APSE toolsets.

While STONEMAN provides some guidance on what APSEs must support effectively (such as maintenance and configuration management), STONEMAN does not attempt to present an extensive or very refined view of how to populate an APSE.

At the moment, therefore, an important opportunity exists for conceptualizing what an advanced APSE should contain.

Thus it is important to do our homework on what a full APSE should look like, and a number of important targets of opportunity come to mind.

POSSIBLE TARGETS OF TECHNOLOGICAL OPPORTUNITY FOR APSES

Interactive Programming

Although it is hard to cite credible experiments that demonstrate that interactive programming is more productive than batch programming, some experiments suggest an improvement of roughly 33% if interactive programming is used in place of batch.

Good interactive languages, such as APL and LISP, permit sophisticated and powerful actions to be taken by programmers while interacting with their programs. For example, at a point of suspension of a running program, a user at a terminal can do such things as: (a) print formatted values or texts of defined procedures; (b) define new procedures or assign new values to new variables; (c) perform queries ("Where am I?, Who calls this procedure? Who can read and set this variable?"); (e) resume program execution at the point of sus-

pension (or at other valid points of control); (f) call for explanations to be printed from online manuals; and (g) set breakpoints, traces, and performance measurement probes.

It is rare that such interactive services are available to the user of a high-performance systems programming language, such as JOVIAL, CMS-2, BLISS, C, Pascal, or MESA. In order to support queries and incremental changes characteristic of interactive programming, programs must usually be represented in a somewhat elastic (and thus incrementally updatable and explicitly queriable) representation. Usually this implies that program representations must be interpreted to be executed. On the other hand, to get high performance, programs must usually be compiled into rigidly efficient machine code. Such machine code does not conveniently support incremental editing in source program terms, and it usually does not contain symbolic information discarded by compilers yet needed at run-time during interactive sessions to reply to user queries in source program terms.

To the author's knowledge, nobody has succeeded satisfactorily in providing the combined advantages of compiled systems programming language performance with the power of interactive language query and incremental change. There may be a considerable technological challenge in providing this kind of support for Ada (or for any other high-performance systems programming language for that matter).

Management Support

If the evidence suggests that upward of 80% of software project failures are management failures and not technical failures, and that these failures result, in large measure, from ignoring software practices of proven effectiveness, what might we do to support project management so it can avoid well-known pitfalls?

Might we have online management interviews at the time a project is being organized to remind managers about software practices of proven effectiveness and to enable them to select thorough, effective project disciplines well-matched to a particular organization's characteristics?

What sort of management support tools might help managers devise project schedules, estimate resources required, make required reports, track project activities (monitoring especially the activities on the critical path), and adjust resources incrementally to fit changing needs?

Programming Methodologies?

Should an APSE support a programming methodology? If so, should it try to support a standard one and encourage its use? For example, should an APSE provide for use of an Ada-based program design language (PDL) together with some sort of discipline for design composition, design review, and requirements tracing?

Any suggestion that an APSE should support a standard programming methodology usually engenders heated opposition and dire warnings about the evils of premature standardization, and the points about such evils are usually well-taken. However, it may be possible to phrase the policy on the use of such methodologies in order to overcome most of the objections. One might say, for instance, "Here is a recommended methodology which is provided in the APSE library, and here are its abstract characteristics: (a) it provides a clear, comprehensible design representation, (b) it is accompanied by effective ways of getting design review by independent teams, and (c) one can determine which design modules are responsible for implementing which items of the system requirements, and so on." An RFP might then specify the following: "You can propose either to use the recommended methodology or you can propose to use your own, but if you choose to use your own, you should give some justification as to how using your own meets the essential abstract characteristics of the recommended one."

Software Reuse

Since the biggest cost-driver in software projects is the size of the software, any method that permits successful reuse of software to implement portions of a system dramatically increases productivity. Although the idea of software reuse has been around for a long time, and although it works in limited application areas (typified by well-defined interface and composition paradigms and by libraries of useful, well-indexed, well-explained components), we do not generally build software by assembling catalogued, prefabricated components. Getting software reuse methods to work as a general program composition technique may involve surmounting challenges such as finding ways to reuse designs and higher-level program abstractions and finding how to generate concrete refinements of the abstractions that meet the extraordinary variety of concrete usage constraints encountered in practice. Nonetheless, the payoff for finding an effective software-reuse technology would be dramatic, especially if performance measurement and certification were performed on all components entered into a component catalogue.

Program Understanding

Software maintenance accounts for 70 to 90% of the cost of the software life cycle for many large, long-lived systems. If, as recent evidence suggests, upward of half of the software maintenance time is devoted to trying to understand how a program works and what the effects of a proposed alteration would be, the activity of trying to understand programs and the effects of incremental program changes could be a dominant cost-driver in the system life cycle.

If this is the case, there might be an inviting technological target of opportunity in trying to devise ways of making it vastly less expensive and more effective to go about understanding programs. We might ask the following questions:

1. How can we write comprehensible program descriptions?
2. Is paper a good container for program documentation, or can we do better by using a computer to store explanations appropriate for different intended audiences and by computing various appropriate views for the different audiences?
3. Given a projected software lifetime (and other appropri-

ate unit costs), what level of capitalization is appropriate for developing program explanations?
4. Are there any techniques for "program archaeology," wherein, if we are confronted by an undocumented or poorly documented program, we could systematically go about trying to develop an understanding of it and whereby we could estimate the cost of doing so ahead of time?

Advanced APSE Tool Sets

What kinds of tools could an APSE provide the system builder? (Unfortunately, there is a great variety of answers to this question, and space does not permit the author to do more than provide a pointer or two to the literature. Two good sources that provide a variety of views and excellent bibliographies are Hünke[4] and SIGSOFT.[5])

RISK AREAS IN APSE DEVELOPMENTS

What are some of the risk areas that confront the development of APSEs? Since this is highly speculative, the author prefers to give just two areas where he perceives risk:

1. *No KAPSE Standardization:* What if the KAPSE never gets standardized? The KAPSE in STONEMAN is envisaged as a machine-independent Kernel operating system and database support system. If it can be standardized (with a machine-independent interface, given, for example, as an Ada package), one can write machine-independent Ada programs which call on KAPSE services in a standard notation (much as package Standard functions in Ada now), and such Ada programs will transfer to every machine on which an Ada compiler interfaces to a standard KAPSE. A special case of program transfer of great interest is a full APSE with tools written in Ada and depending on KAPSE services for support. Thus, KAPSE standardization holds the key to the machine independence of APSEs and to providing a powerful conduit for portability of Ada programs and environments. If the KAPSE cannot be standardized, can the market for APSE tools, which depends on having a viable method for the exchange and portability of Ada programs, ever become an effective reality?
2. *Too Little and Too Late:* What if some of the thirty or so current efforts to write Ada compilers succeed and serious Ada programming begins before MAPSEs and APSEs can be designed, built, and used? If serious Ada programming begins starting with a compiler, does one not then tend to use the available tools in the *de facto* environment surrounding that compiler (meaning the text editors, file system, linkers, and so forth), and do not critical dependencies then develop which inhibit program transfer to other different environments (with other different file system and operating system conventions)? To what degree is the timeliness of APSE development a critical factor in its possible success?

CONCLUSIONS

In conclusion, this paper argues that the benefits some seek for the introduction of Ada cannot be realized effectively without also introducing advanced APSEs that provide capabilities well beyond the STONEMAN MAPSE level. Furthermore, the time is ripe to do our homework on what a full APSE should look like, and a number of inviting targets of technological opportunity present themselves.

ACKNOWLEDGMENTS

This work was supported by the Defense Advanced Research Projects Agency of the United States Department of Defense under contract MDA-903-82-C-0039 to the Irvine Programming Environment Project. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

REFERENCES

1. Boehm, Barry. "Software Engineering as it is." 4th International Conference on Software Engineering, Munich, September 1979.
2. Boehm, Barry. *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
3. Buton, J.N. and Druffel, L. E. "Requirements for an Ada Programming Support Environment, Rationale for Stoneman." *Proceedings COMPSAC 80,* Chicago, Ill., October 1980.
4. Hünke Horst. *Software Engineering Environments.* North Holland, Amsterdam, 1980.
5. SIGSOFT. NBS Workshop Report on Programming Environments, *Software Engineering Notes,* Vol. 6, No. 4, August 1981.