Technical Report
CMU/SEI-87-TR-1

Evaluation of Ada Environments, Executive Summary

Nelson Weiderman

1987

Technical Report

CMU/SEI-87-TR-1

1987

Evaluation of Ada Environments, Executive Summary

Nelson Weiderman

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESC/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

John S. Herman, Capt, USAF (Signature on File)
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Executive Summary

Objectives

An important goal of the Software Engineering Institute is to assess advanced software development technologies and disseminate those that appear promising. Ada Programming Support Environments (APSEs) have already been developed under government contract and for the commercial marketplace, and more are certain to follow. Such environments play key roles in increasing the productivity of software engineers, in improving the quality of embedded systems software, and in reducing the cost of producing and maintaining software.

The initial purpose of the *Evaluation of Ada Environments* project was to determine the suitability of the Army/Navy Ada Language System (ALS) and the Air Force Ada Integrated Environment (AIE) for application to software engineering activities. The ALS was delivered to SEI in late 1985 and the AIE has yet to be completed as of the middle of 1986. Early in the project a decision was made to develop a systematic methodology for evaluation of environments and to include a number of commercial environments in the study. These environments were extensions of existing operating systems, but included toolsets rich enough to be considered Minimal Ada Programming Support Environments (MAPSEs).

One major outcome of the study has been the definition of a methodology that adds a degree of rigor and standardization to the process of evaluating environments. Without a systematic approach, evaluations offer little more than *ad hoc* evidence of the value of an environment. Our methodology is comprehensive, repeatable, extensible, user-oriented, and environment independent in the initial steps. It has been applied to several Ada environments at the Software Engineering Institute so that they may be compared objectively according to the same criteria. This cross-environment comparison of three environments is the second major outcome of the study.

The full report of the study provides a detailed description of the methodology and examples of its usage. Chapter 1 gives an extended cross-environment analysis of the results of the project. For each of five experiment groups it compares three APSEs. The chapter provides an overview of the results of all the experiments and is written for the technical manager. Chapter 2 describes in detail the methodology used for evaluating the environments, along with some of the background information and references to previous work in environment evaluation. Chapters 3 through 8 provide detailed descriptions of the six experiment groups. Here one can find the information on

particular criteria, questions, and life cycle activities that were tested for each experiment, as well as test scripts, checklists, and resulting data that were collected.

The purpose of this executive summary is to capture the most salient characteristics of the study and the most important results. The remainder of the executive summary will be devoted to a brief description of the environments evaluated, a brief description of the experiments conducted, a summary of the most important results, and finally some reflections on where we currently stand in APSE technology and where we are likely to make the most significant progress.

Environments Evaluated

A total of four environments were installed and tested at the SEI. The SofTech Ada Language System was developed for the Army and was designed to be retargetable and rehostable. In spring of 1986, continued development work on the ALS was halted by the Army and is being continued by the Navy. The ALS is hosted on the Digital Equipment Corporation VMS operating system and will be referred to as VMS/ALS. The remaining three environments are extensions to existing operating systems. The extension to VMS by DEC includes the VAX Ada and five additional tools collectively referred to as VAXSet. This environment will be referred to as VMS/VAXSet. The third environment considered is an extension to Unix developed by Verdix. Their product is called the Verdix Ada Development System (VADS) and is referred to here as Unix/VADS. A fourth environment is also based on Unix and includes an Apollo environment called DOMAIN Software Engineering Environment (DSEE) with an Alsys Ada compiler. This will be referred to as DOMAIN/IX/Alsys environment.

It must be emphasized that all experimentation took place in the spring of 1986. The results presented in this report represent a snapshot of the environments at that point in time. Corrections of errors, improvements in performance, and new functionality or documentation may have been incorporated in subsequent releases of the products tested.

The first three environments described above ran on MicroVAX II hardware. They were installed on three distinct machines. In each case the hardware configurations met or exceeded the vendor's initial recommendations. In the case of the SofTech Ada Language System, version 3.0, the configuration included 9 megabytes of main memory and disk space consisted of three RD53 disk drives (213 megabytes). The Digital Equipment Corporation product consisted of the VAX Ada (version 1.2) and VAXSet. Both the VMS/ALS and VMS/VAXSet environments were run using version 4.2 of microVMS. The configuration for VMS/VAXSet included 6 megabytes of main memory and 102 megabytes of disk space (one 31 megabyte RD52 disk drive and one 71 megabyte RD53 drive). The Verdix product (VADS, version 5.1) ran on top of the DEC supplied

version of Unix called Ultrix (version 1.2). The tests were run on a system configured with 6 megabytes of memory and 202 megabytes of disk space (one 31 megabyte RD52 drive and one 171 megabyte Fujitsu drive).

The fourth environment ran on an Apollo computer with the DOMAIN/IX Unix equivalent operating system. The Domain Software Engineering Environment (DSEE) and the Alsys Ada Compiler completed the software configuration. The hardware consisted of an Apollo DN460 workstation with 4 megabytes of main memory and 171 megabytes of disk space. The versions tested were version 2.0 of DSEE and version 1.0 of the Alsys Ada compiler.

Experiments Conducted

The six experiment groups are called Configuration Management, System Management, Design and Development, Test and Debug, Project Management, and the ACEC Tests. The details of these experiments including the functions tested, the questions asked and answered, the scripts used to exercise the environments, as well as the conclusions reached and are presented in Chapters 3 through 8.

Five of the six experiment groups (the exception being the Project Management experiment) were run on the three APSEs which ran on the MicroVAX II. The other experiment was run on an Apollo workstation. Five of the six experiment groups were constructed using the methodology detailed in Chapter 2. The other experiment group consisted of running the prototype Ada Compiler Evaluation Capability (ACEC) test suite that was assembled and instrumented by the Institute for Defense Analyses (IDA).

The configuration management experiment group exercises the configuration management and version control capabilities of the APSE. The experiments simulate the system integration and testing phase of the life cycle by having three separate development lines of descent from a single baseline system. This process provides information about the version control capabilities (space requirements, transparency, performance) as well as the configuration management capabilities (translation rules, specification of modules from releases, history collection, performance). The system management experiment group exercises the environment from the perspective of the system manager. The activities of concern here are the installation of the APSE on a raw machine or operating system, the management of user accounts, maintenance of the environment, and system resource management.

The design and code development experiment group exercises the activities normally associated with small projects, namely the design, creation, modification, and testing of a single unit or

module. Using an editor, Ada programs are entered along with a test harness to allow initial testing of these units. The unit testing and debugging experiment group exercises the environment from the perspective of the unit tester. It is designed to be a sequel to the design and code development experiment. A small set of Ada units was seeded with errors and debugged using the facilities available in the environment.

The project management experiment group took into consideration only a small portion of the total functionality which should be attributed to project management. Namely, the purpose of the experiment was to explore the activities surrounding the building and maintaining of the project database which is but one aspect of the technical management activities of project management. The ACEC tests are different from the other five in that they were externally generated and they test only a single component of the environment, namely the compiler. Instead of creating a new set of Ada programs to test the performance of the Ada compiler, we used the IDA prototype test suite.

Results

VMS/ALS

The requirements for the ALS placed a strong emphasis on rehostablility and retargetability. The extent to which it satisfies the first requirement is problematical since it has never been rehosted. It has been retargeted to an Intel 8086, but this capability was not tested in our study. In the areas of software development and lifecycle maintenance, however, our analysis of the ALS (version 3) has shown that the product has not reached a level of maturity, stability, or performance to recommend its use. Furthermore, we believe that it is unlikely to reach the required level of maturity without significant and costly changes. It is our observation that the current ALS product is not competitive with currently available commercial products in the four major criteria areas identified -- functionality, performance, user interface, and system interface. We found many errors and a general lack of robustness, which is uncharacteristic of products of high quality already available on the commercial market.

In the areas of functionality tested by the four experiment groups, the ALS does indeed have many of the tools required for the activities we have identified. The configuration management functionality is good with the exception of the ability to merge a group of changes made to variant versions from a given baseline and the automation of recompilation. The tools for programming in the small are also present for the most part. The exceptions in this case are the lack of an Ada browsing capability and syntax sensitive editing. In the area of testing, the functionality of the debugger was limited as were the dynamic analysis capabilities. By modern standards many of

the tools are not well designed and implemented. The command language lacks a number of commonly provided features and the editor lacks a multi-window capability.

The most serious deficiency of the VMS/ALS environment is in the area of performance. The ALS is, in many cases, more than an order of magnitude less efficient in its time and space characteristics than other commercially available environments. Compile times are, on the average, more than fifteen times slower than the VAX Ada compiler. Routine database commands are five or more times slower on the ALS than on VMS (creating or deleting directories, for example). New versions of files are stored in their entirety rather than storing the incremental differences (deltas) between successive versions. This particular design decision can result in two orders of magnitude difference in the total amount of space required for a typical series of revisions. The most notable instance of poor performance is the time required to create a new program library, which takes more than 15 minutes elapsed time on a dedicated MicroVAX II. Such performance can have a direct impact on the project management philosophy by discouraging the creation of multiple program libraries.

The user interface consists of a command language very similar to Ada. While this provides a level of consistency with the language being supported by the environment, it is cumbersome to enter long Ada commands without the appropriate automation for command entry. The poor quality of the error messages hampered development. They are not informative and in many cases are misleading or superfluous. The documentation is hard to use and in cases contradictory and incomplete. The on-line help facility is primitive.

The interface with the underlying operating system could be one of the primary causes for performance degradation. The layered architecture provides an isolation from the underlying system which may be advantageous from a portability point of view, but causes additional overhead. The sequential (rather than parallel) use of files for many operations causes the environment basically to be a single user batch system rather than a truly multi-user interactive system. Furthermore, the ALS user is prevented from using many of the tools provided by the underlying operating system. Some of these tools are not available in the ALS environment, and thus useful functionality is lost.

The software engineers found that learning the ALS was considerably more frustrating than learning and using other operating systems. This observation can be attributed to a number of bugs, poor documentation, poor performance, and the general lack of commonly provided features. Examples of bugs which are more fully documented in the body of the report include tools which do not work at all, tools which do not work as documented, tools which generate incorrect Ada programs, and tools whose names changed from one version of the ALS to another.

Our conclusion is that there are major deficiencies in the ALS (version 3) environment and that many of these deficiencies are not easily correctable. Discussions with a number of ALS users indicate that the ALS has been chosen for its ability to generate code for the Intel 8086 where it is the only available choice as of mid-1986. Only one of the four users we contacted is using the broad functionality of the environment. In fact, the SofTech representatives told us that we had exercised the configuration management system more than any users that they knew of.

VMS/VAXSet

The VMS/VAXSet environment is an extension of a mature and stable operating system (VMS) which was developed for the commercial market and which has been in widespread use for many years. In addition to the Ada compiler the VAXSet toolset includes a Language Sensitive Editor (LSE), a Performance and Coverage Analyzer (PCA), a Test Manager (TM), a Configuration Management System (CMS), and a Module Management System (MMS). Because the entire toolset has been provided by a single vendor the environment provides a conceptual integrity and consistency greater than any of the other three environments studied.

The VMS/VAXSet environment provided the strongest set of tools and features for those tools. In the area of configuration management, CMS and MMS provide all the identified activities with the exception of maintaining product release information. A complete set of functionality was provided in the system management area as well. In the area of design and development, there were no design tools and the only browsing capability was the ability to find a specified Ada object. There was also no pretty printing capability. However, the LSE provides the ability to enter Ada programs while catching syntax errors and formatting the code. In the area of testing and debugging, the PCA and TM tools form the core of a rich set of features which far surpass what is available in either VMS/ALS or Unix/VADS.

Performance of VMS/VAXSet was uniformly quite good. Most of the CM operations took small numbers of seconds and averaged one quarter of the time required by the ALS. Most user account management functions were also in the three second range. Program library creation time is on the order of 13 seconds. Space requirements were modest for most functions and successive CM versions are stored as deltas. Overall, the operations which one would expect to be accomplished quickly were accomplished quickly and those that one would expect to take longer were not excessively long. The Unix/VADS environment was faster in many areas, but the differences were not significant for the majority of the operations.

The user interface of the VMS/VAXSet environment was considered the best for various reasons. Because of its maturity, the command language supports many popular features including string

symbols, logical names, filename wildcards, command abbreviations, command line editing, command recall, and parameter prompting. Commands are syntactically consistent, each being comprised of one or more command keywords, command qualifiers, and parameters. Diagnostic messages presented to the user from all VMS utilities and tools are consistent and of high quality. They are timely, informative, accurate, uniform in format, and consistent with the documentation. The documentation was very good and supplemented by an excellent on-line help system.

The VMS/VAXSet environment is an extension to an existing operating system rather than a layer on top of another operating system. There is an interface with a VMS system services package called STARLET. For the most part this interface is invisible to the Ada developer and provides the operating system interface for the tool developers. The tools of the VMS/VAXSet environment are well-integrated. One example is the editor which is capable of invoking the compiler without leaving the editing session so that the user may uncover semantic errors without great distraction while making changes to Ada units. The debugger can invoke the editor or the Ada compiler.

Across the criteria dimensions of functionality, performance, user interface, and system interface, it is clear that VMS/VAXSet provides a very good, production quality environment that is capable of supporting development, integration, and maintenance of small to medium size Ada software systems.

Unix/VADS

The Unix/VADS environment is an extension of an operating system which was developed in the early 1970's for laboratory use (Bell Labs) and today has received widespread acceptance in its many versions. The two predominant versions are Berkeley 4.2 and ATT's System 5. The version of Unix used for this study is the one distributed by DEC called Ultrix. Unix has the reputation of being an environment in which it is easy to add tools and to have tools work well together, but because of its roots and history, it lacks the conceptual integrity and some of the functionality of many single vendor commercial operating systems.

In the area of functionality, the Unix/VADS environment provides a reasonably well-integrated set of utilities capable of supporting the development, integration, and maintenance of Ada software systems. Support is provided for all typical configuration management activities, but the operation of including variant versions in baselines is difficult. SCCS supports common version control activities while Unix/Make and VADS support the configuration control activities. Unix/VADS is clearly deficient in support of user account management functions. In the area of unit development, there was no support for syntax directed editing, but there was reasonable support for other

functions including automatic recompilation. Browsing through Ada programs was not particularly well supported. Unix/VADS had no data management capability and testing and regression testing are manual procedures. The debugger has a reasonably rich set of features and is good at displaying the current state of the program.

Performance of Unix/VADS was quite good, with many commands taking one quarter to one half the time required by VMS/VAXSet. The creation of an Ada program library took approximately 3 seconds. Space utilization was reasonable for most operations and the granularity of the allocation was smaller than it was for VMS/VAXSet. Successive CM versions are stored using deltas.

The user interface for Unix/VADS is in the style of Unix and is characterized by the brevity of both commands and diagnostic messages. It seems that there is an overriding emphasis on conserving keystrokes on input and character space on output. This may be helpful for the experienced frequent user, but is a liability for novice or occasional users. The Unix shell does support string aliases, wildcards, full pathname wildcards, command line editing, and command recall, but it does not support command abbreviations, in-line keyword expansion, or parameter prompting. Most of the VADS error messages are in the style of Unix, which tends to be terse, but the compiler error messages are very informative and clearly presented. The on-line help system is good and has keyword search capability. The VADS documentation contains a partial users guide but needs more examples and detailed explanations for command options.

The interface to the underlying environment in the case of Unix/VADS is to a Unix kernel which gives Unix its portability characteristics through a user-provided set of callable interfaces. All the tools use standard ASCII text files which support efficient information sharing, but which do not provide a rich information structures for persistent objects with many attributes. Tools in Unix/VADS are for the most part standalone and utilize the piping mechanism to allow tools to interact. The debugger can involve the editor, but not the compiler. It presented data well, identified errors precisely, and was easy to use.

In summary, the Unix/VADS environment is a good extension of the Unix environment and will be readily accepted by veteran Unix users. The performance is very good for both the compiler and tools, but its user interface, documentation, account management capabilities are somewhat lacking.

DOMAIN/IX/AIsys

The Apollo DSEE represents an interesting approach to some of the problems encountered in building and maintaining large software systems of programs and documentation. Unfortunately

the DOMAIN/IX/Alsys environment represents only a great unfulfilled potential. Problems with the environment for Ada development and maintenance include poor integration of the Ada compiler and DSEE, lack of individual file copy capabilities, and a multiplicity of user interfaces which presents an inconsistent system. Since the DSEE product is currently available only on Apollo hardware it has thus far had limited distribution. The Apollo/Alsys combination does deserve further scrutiny if the companies get together to provide a more integrated Ada environment with a consistent user interface.

Compiler Evaluation

Numerous problems were encountered in applying the three compilers to the ACEC test suite and the ACEC support software. To compile successfully using the ALS compiler, the ACEC support software had to be modified slightly.

The ALS compiler failed to compile 6 tests and, of the remainder, 21 executed incorrectly. VAX Ada compiled and executed all tests. The VADS compiler compiled all tests, but 4 tests executed incorrectly.

Of the many numerical results obtained by executing the ACEC suite of test programs and analyzing the measurement files, the most reliable and meaningful were the average CPU times. For compilation and linking, ALS took 34 times as long as the VAX Ada, while VADS took 3 times as long. For execution, the ALS took 1.7 times as long as VAX Ada, while the VADS took 1.3 times as long.

The main purpose of the ACEC tests is to derive differential statistics for individual language features; that is to measure the *increment* in time or space caused by using a particular feature. Unfortunately, no conclusive differential statistics were obtained (many differentials were in fact *negative*!). After filtering out the clearly erroneous data, the relative ratios between compilers for time and space utilization were roughly consistent with the aggregated ratios listed above. The ratios showed considerable variation between language features, but without any clear pattern.

The exact cause of the erroneous differential measurements is still being investigated. The fact that they occurred with all three environments suggests a flaw, not necessarily in the ACEC concept, but in the ACEC implementation of that concept or in our adaptation and implementation. The erroneous measurements are not confined to the instrumented timings generated from within the test programs, but also appear in the external measurements performed by the respective operating systems, implying that more sophisticated measurement techniques are required (e.g., automatic calibration). Another problem with the ACEC is that the Report Writer

provides only a rudimentary analysis facility; an entirely separate analysis program had to be developed for this project.

Summary and Future Outlook

The work accomplished in this project represents a step forward in the ability to evaluate Ada environments in a more standard and systematic fashion. However there are limitations to the study which were brought about by time and resource constraints. Further work must be devoted to issues of programming-in-the-large, project management activities, and observing performance under load. Additional environments also need to be evaluated and compared to the initial four.

Some of the weaknesses of all the environments were the inability to navigate or browse through Ada programs, and the lack of design tools, test generation tools, and static analysis tools. Furthermore, all the environments had capabilities and features which represent the trailing edge rather than the leading edge of technology. They all adhere to the command interfaces and the edit-compile-link-execute cycle typical of the interactive batch operating systems of the 1960s and 1970s. All of the environments tested provide a monolithic compilation system which prevents access to intermediate forms of Ada programs which are useful to tool builders for highly integrated tools.

It is speculated that the new generation of Ada environments will be more in the flavor of Interlisp, Smalltalk, and the Cornell Program Synthesizer. These environments are built around a single programming language and support a highly interactive paradigm. Because of the dependencies built into the Ada programming language for inter-module consistency checking, even insignificant changes may cause propagation of recompilations throughout a large system. The only way to support interactive changes without excessive delays is to incorporate the changes in a rich data object (a DIANA tree, for example) which preserves syntactic and semantic information rather than in a simple ASCII text file. None of the environments tested supports incremental compilation at this time, but such environments are commercially available.

In order to support programming-in-the-large it will be necessary to support persistent objects with attributes. Requirements, code, test data, change logs, and documentation must all be retained and related for the life of the software. These and other capabilities will be necessary in order to provide the necessary breakthroughs in productivity for the Ada programming environments of the 1980s and 1990s.

Work on environment evaluation is continuing at the Software Engineering Institute. The current

focus is on refining the experiments in the areas of project management and tool integration. ESPRIT's Portable Common Tool Environment (PCTE) and Imperial Software Technology's IS-TAR are candidate environments for future study. The work described in this report is being used by Computer Sciences Corporation to evaluate Rational's R1000 environment and by Mitre to evaluate GTE's System Development and Maintenance Environment (SDME).

1. Summary Analysis

1.1. Introduction

The purpose of this chapter is to provide a cross-environment summary analysis of the results of the Evaluation of Ada Environments project. The results presented here are more detailed than those in the executive summary, but not as detailed as those found in later chapters. This chapter is aimed at providing a management level overview of the experiments conducted and the major lessons learned from those experiments.

For each of the six areas of experimentation, this chapter provides a section which summarizes the results of the experimentation broken down into the four criteria categories of functionality, performance, user interface, and system interface. The six experiment groups will be referred to in this chapter as Configuration Management, System Mangement, Design and Development, Test and Debug, Project Management, and the ACEC Tests. The details of these experiments including the functions tested, the questions asked and answered, the scripts used to exercise the environments, as well as the conclusions reached are presented in Chapters 3 through 8. Chapter 2 gives the details of the philosophy of the approach taken in this study and the broad criteria and methodology used in developing the experiments.

Five of the six experiment groups (the exception being the Project Management experiment) were run on nearly equivalent hardware configurations based on Digital Equipment Corporation's MicroVAX II. Each of these five experiment groups were duplicated for the three APSEs which ran on the MicroVAX II. The other experiment was run on an Apollo workstation. Five of the six experiment groups were constructed using the methodology detailed in Chapter 2. The other experiment group consisted of running the Ada Compiler Evaluation Capability (ACEC) test suite that has been assembled by the Institute for Defense Analysis.

The five experiment groups that were run on MicroVAX II hardware were actually run on three distinct machines with the three different environments installed. In each case, the hardware configurations met or exceeded the vendor's recommendations. In the case of the SofTech Ada Language System, version 3.0 [AlsText], (herein referred to as the VMS/ALS environment) the configuration included nine megabytes of main memory and disk space consisting of three RD53 disk drives (213 megabytes). The Digital Equipment Corporation supplied product consisted of the VAX Ada (version 1.2) [ACS] and a set of five tools called VAXSet. Since this runs on the VMS operating system, it will be referred to as VMS/VAXSet. The configuration for VMS/VAXSet included six megabytes of main memory and 102 megabytes of disk space (one 71 megabyte RD53 drive and one 31 megabyte RD52). The Verdix product is called the Verdix Ada Develop-

ment System (VADS, version 5.1) [VADSOper] and runs on top of the DEC supplied version of Unix called Ultrix (version 1.2). This environment will be referred to as Unix/VADS. The tests were run on a system configured with six megabytes of memory and 202 megabytes of disk space (one 171 megabyte Fujitsu drive and one 31 megabyte RD52).

The final experiment group ran on an Apollo computer with the DOMAIN/IX Unix equivalent operating system. The Domain Software Engineering Environment (DSEE) and the Alsys Ada Compiler completed the software configuration. The hardware consisted of an Apollo DN460 workstation with four megabytes of main memory and 171 megabytes of disk space. The versions tested were Version 2.0 of DSEE and Version 1.0 of the Alsys Ada compiler.

1.2. Configuration Management Cross Environment Analysis

This section presents an overall comparative analysis of the ability of each Ada environment to support typical Configuration Management (CM) and Version Control functions. The overall intent of the CM experiments is to provide a comprehensive evaluation of an APSE's version control capabilities (i.e., support of successive versions, variant versions, file checkin/checkout) as well as its configuration control capabilities (i.e., support of system construction and re-construction, baselining, release management, history collection). For each Ada environment under investigation (in order, VMS/ALS, VMS/VAXSet, and Unix/VADS), the results from developing and performing each CM evaluation experiment shall be analyzed (compared) along the criteria dimensions of functionality, performance, user interface, and system interface. This cross-environment comparative analysis summarizes the material presented in Chapter 3.

1.2.1. Analysis Along Evaluation Criteria Dimensions

1.2.1.1. Functionality

For the most part, the ALS CM toolset takes advantage of the capabilities of the underlying (ALS) node model supporting such functions as creating and deleting database objects, fetching database objects, reserving and replacing database objects, and baselining. However, it is apparent from the cross-environment functionality checklist (page CMCROSSFUNLIST) that the ALS environment does not provide support for typical Configuration Management activities in two areas: automatic re-compilation of a system and merging changes made in variant versions of a CM file element.

Configuration Management/Version Control Functionality Checklist

Activity	Supported (Y|N)		
	VMS/ALS	VMS/VAXSet	Unix/VADS
Version Control			
Create element	Y	Y	Y
Delete element	Y	Y	N
Create new version			
Successive	Y	Y	Y
Parallel	Y	Y	Y
Derived	Y	Y	Y
Merge variants	N	Y	Y
Retrieve specific version			
Explicit	N	Y	Y
Dynamic	Y	Y	Y
Referential	N	Y	Y
Compare different file versions	Y	Y	Y
Display history attributes	Y	Y	Y
Configuration Control			
Define system model			
Specify source dependencies	Y	Y	Y
Specify translation rules	Y	Y	Y
Specify translation options	Y	Y	Y
Specify translation tools	Y	Y	Y
Build system			
Current default	N	Y	Y
Earlier release	Y	Y	Y
Hybrid	Y	Y	Y
Product Release			
Baseline system	Y	Y	Y
Create system release class	Y	Y	Y
Maintain product release information	N	N	N
Display members of a released system	Y	Y	Y
Display system release history	N	Y	Y

Also apparent from the cross-environment functionality checklist, (page CMCROSSFUNLIST) is that the DEC VMS/VAXSet environment provides support for all typical configuration management activities. Specifically, DEC/CMS [CMSUser] supports all typical version control functions: creating and deleting database objects, fetching database objects, reserving and replacing database objects, creating successive and variant versions, and baselining. Furthermore, DEC/MMS [MMSInstall] provides support for the most common configuration control activities, namely, system modeling ("Makefile") and automatic software system construction. These tools in combination with the VAX Ada functionally provide the basis for an excellent Ada software development environment which is capable of supporting various project scenarios.

The Unix/VADS environment provides support for all typical configuration management activities, although the operation of including variant versions in baselines is difficult. Specifically, SCCS

within the context of the Unix file system supports common version control activities such as creating database objects, fetching database objects, reserving and replacing database objects, creating successive and variant versions, and baselining. Furthermore, Unix/Make and the Verdix Ada Development System support typical configuration control activities such as system modeling ("Makefile"), automatic software system construction, and maintenance of transactional history information.

1.2.1.2. Performance

As is evident in the cross-environment performance analysis table (page CMCROSSPERFTABLE), the ALS requires on average 4 times more elapsed time than the VAX-Set tools to perform typical Configuration Management activities. The overall performance the ALS could be poor for a number of reasons, but the most likely reason is its layered implementation on top of VMS. Specifically, it is conjectured that the slow response times are due to the method of database access and the ALS process model. For each reference to an ALS database object, one must pay an overhead penalty associated with the underlying locking mechanism that is employed for insuring the database's integrity and correctness. A characteristic of the ALS's process model that greatly degrades performance is its process intensiveness, meaning its use of many VMS processes to perform its functions.

Configuration Management/Version Control Performance Comparison

Activity	Elapsed Time (seconds)		
	VMS/ALS	VMS/VAXSet	Unix/VADS
System build operation	1937(9.44)	205(1.0)	169(0.82)
Create CM file element	13.64(2.43)	5.61(1.0)	1.42(0.25)
System baseline operation	220(2.31)	95(1.0)	45(0.47)
Merge operation	N/A	No change	126
Fetch CM element	14.46(3.24)	4.46(1.0)	0.82(0.18)
Create variant of CM element	48(4.0)	12(1.0)	4(0.33)
Fetch variant of CM element	28(5.6)	5(1.0)	0.90(0.18
Reserve variant of CM element	15(2.73)	5.5(1.0)	2(0.36)
Replace variant of CM element	19(2.88)	6.6(1.0)	3.2(0.48)
Merge variant versions of CM element	N/A	13(1.0)	3.7(0.28)
Reserve CM element	14(2.42)	5.78(1.0)	0.82(0.14)
Replace CM element	19(2.74)	6.94(1.0)	2.7(0.39)
Display history information for CM element	N/A	0.33(1.0)	3.55(10.76)
Re-build earlier baselined system	3656(11.23)	325(1.0)	188(0.58)
Delete CM element	N/A	5.44(1.0)	N/A
Compare different versions of CM element	11(6.32)	1.74(1.0)	0.23(0.13)
File size measurements		**Size change in bytes**	
Baseline inclusion	2560	No change	90
Successive version	100% increase	No change	107
Variant version	100% increase	No change	110
Merge operation	N/A	No change	126

The overall performance of the Unix/VADS environment for the CM activities is good relative to that of VMS/VAXSet. Specifically, to perform these CM functions, Unix/VADS requires on average one third the amount of VMS/VAXSet elapsed time except for the anomalous case of displaying history information pertaining to all the CM file elements in a CM library. This anomaly can be attributed to a difference (between SCCS and CMS) in the method employed for maintaining the history information. CMS maintains the transactional history information for all CM elements in one indexed file, whereas SCCS maintains this information as a prologue to each individual CM file element. Therefore, displaying history information for all CM file elements in a SCCS directory requires accessing each individual file and thus more elapsed time than CMS which need only access one special "history" file.

1.2.1.3. User Interface

Of the three environments, the user interface of VMS/VAXSet is the best for various reasons. First, it supports many popular features including: string symbols, logical names, filename wildcards, command abbreviations, command line editing, command recall, and parameter prompting. Second, the DEC Command Language (DCL) is very powerful and easy to learn. The commands are syntactically consistent each being comprised of one or more command keywords, command qualifiers, and parameters. A helpful feature of DCL is that the command qualifiers can be placed anywhere on the command line following the command keyword(s). Third, the diagnostic messages presented to the user (from all VMS utilities) are of a high quality. They possess many characteristics common to messages generated within any production quality software development environment: timely, informative, accurate, uniform in format, and consistent with the documentation. Lastly, consistency exists in the user interface across individual tools in the form of a similar command structure and functionality.

The ALS user interface is missing some essential features including: command abbreviations, command completion, parameter prompting, in-line keyword help, and wildcards. Additionally, the quality of the error diagnostics is poor since limited information is displayed and the usefulness of what is presented is diminished by the inconsistent and lacking documentation.

Overall, the Unix/VADS user interface is good with the only exception being the brevity of its commands and diagnostic messages. The diagnostic messages generated by SCCS and VADS are adequate, but as is the case with most error messages provided by Unix-based tools they tend to be brief and cryptic. Also, the Unix command language being targeted to the expert user is more difficult to learn (more odd command names to learn) than other command languages.

1.2.1.4. System Interface

A side-effect of the ALS's layered implementation approach is the subsetting or in some cases total hiding of some of the underlying operating system's features. Most notably the KAPSE interfaces compare unfavorably to the VMS system services provided in the STARLET (VMS system services package) [Run-Time] specification. Another shortcoming is the inability to submit batch jobs from within the ALS. Yet another missing capability is that of determining the size in blocks (or bytes) of a file within the ALS. Other VMS features are also limited or hidden by the ALS layer: monitoring the activity of the system, multiple command recall, invocation of third party software (e.g., the EMACS editor can be used within VMS but the ALS cannot currently support any editor other than the DEC standard EDT).

Both the DEC VMS/VAXSet and Unix/VADS environments provide a well integrated, comprehensive set of utilities capable of supporting the development and integration of Ada software sys-

tems. All of their respective CM tools utilize ASCII text files wherever possible to promote and support information sharing. Furthermore, both the VAX Ada and Verdix Ada Development System support a full complement of VMS and Unix system service calls respectively. One drawback of the VMS/VAXSet system interface is the fact that the smallest granularity of VMS files is a 512 byte block which affects the accuracy of the file size measurements.

1.2.2. Summary

Judging from the analysis across the criteria dimensions of functionality, performance, user interface, and system interface, it is clear that both VMS/VAXSet and Unix/VADS provide an excellent, production quality environment which is capable of supporting the development, integration, and configuration management of large Ada software systems. The analysis shows that although the ALS environment provides support for a majority of typical configuration management activities, its execution performance, user interface, and system interface prohibit it from being competitive with the other Ada programming support environments under evaluation.

1.3. System Management Cross Environment Analysis

This section presents an overall comparative analysis of the ability of each Ada environment to support typical system management (SM) functions. The overall intent of the SM experiments is to provide a comprehensive evaluation of an APSE's support for typical system management activities (i.e., environment installation, user account management, environment maintenance, and system resource management). For each Ada environment under investigation (in order, VMS/ALS, VMS/VAXSet, and Unix/VADS), the results from developing and performing each SM evaluation experiment shall be analyzed (compared) along the criteria dimensions of functionality, performance, user interface, and system interface. This cross-environment comparative analysis summarizes the material presented in Chapter 4 of this document.

1.3.1. Analysis Along Evaluation Criteria Dimensions

1.3.1.1. Functionality

Both the VMS/ALS and VMS/VAXSet environments provide a well-integrated operating environment capable of supporting a wide range of System Management functions (See the cross-environment functionality checklist on page SMCROSSFUNLIST for further details). Specifically, the underlying VMS operating system provides the context for performing all aspects of the ALS and VAXSet installation process, including such functions as reading the release media, reconfiguring the operating environment, establishing logical names, establishing command aliases, and performing verification and acceptance tests. Furthermore, the VMS operating system provides the Authorize utility which supports commonly performed user account management functions such as creating, deleting, renaming, copying, and listing user accounts, as well as creating and deleting user account groups. In general, VMS user accounts own a vast number of system resource quotas and attributes, all of which are maintained by the Authorize utility.

System Management Functionality Checklist

Activity	Supported (Y|N)		
	VMS/ALS	VMS/VAXSet	Unix/VADS
Environment Installation			
Load environment software from media	Y	Y	Y
Integrate with existing operating environment			
Setup necessary alias/logical names	Y	Y	Y
Operating environment configuration	Y	Y	Y
Run installation procedure	Y	Y	Y
Install help files	Y	Y	Y
Establish access control	Y	Y	Y
Modify system wide start-up procedures	Y	Y	Y
Perform acceptance tests			
Query the on-line help facility	Y	Y	Y
Create a program library	Y	Y	Y
Edit an Ada source code file	Y	Y	Y
Compile a small (main) Ada program	Y	Y	Y
Link a small (main) Ada program	Y	Y	Y
Execute a small (main) Ada program	Y	Y	Y
Delete a program library	Y	Y	Y
User Account Management			
Create user account	Y	Y	Y
Delete user account	Y	Y	Y
Copy user accounts	Y	Y	N
Create user account groups	Y	Y	Y
Add user account to group	Y	Y	Y
Disable user accounts	Y	Y	Y
Delete user account from group	Y	Y	Y
Establish user account characteristics	Y	Y	N
Modify user account characteristics	Y	Y	N
Establish default account chars	Y	Y	N
Modify default account characteristics	Y	Y	N
Display user account characteristics	Y	Y	N
Display default account characteristics	Y	Y	N
Create initial working directories	Y	Y	Y
Establish default login/logout macros	Y	Y	Y
Verify creation of user accounts	Y	Y	Y

While the Unix/VADS environment provides support for all typical System Management activities in the category of environment installation, it is deficient in its support of user account management functions. Specifically, the underlying Unix operating system provides the context for performing all aspects of the VADS installation process, including such functions as reading the release media, establishing symbolic links, linking the source code debugger against existing object libraries, integrating the electronic error reporting program with the standard Unix mailer, and performing verification and acceptance tests. Unix does not, however, provide an account management utility similar to Authorize; rather it supports a minimal set of account management

functions (create a user account group, add/delete a user account) using command scripts provided within the Unix environment. In general, Unix provides a simplistic user account model (i.e., minimal account information, minimal functionality) relative to what is provided within VMS and supports more or less ad hoc maintenance of the user account files.

1.3.1.2. Performance

As is evident in the cross-environment performance analysis table (page SMCROSSPERFTABLE), the ALS installation (command) procedure takes a long time (3 - 3.5 hours) to complete. In part, this lengthy duration can be attributed to the fact that the installation command procedure performs a verification pass over the release media in order to guarantee that the ALS software and database are properly read. In terms of disk space usage, the ALS software and database consume 35 MB of disk space. In contrast, the VAXSet and VADS installations take considerable less elapsed time to complete (1 hour, 15 minutes respectively) and both have lower disk space requirements (15 MB, 9MB respectively).

System Management Performance Comparison

Activity	Elapsed Time (seconds)		
	VMS/ALS	VMS/VAXSet	Unix/VADS
Environment Installation			
Load the environment software	12080(3.10)	3900(1.0)	272(0.07)
Reconfigure underlying operating system	258(1.11)	232(1.0)	N/A
Install on-line help files	N/A	120(1.0)	2.40(0.02)
Establish symbolic links and/or logical names for execution access	0.80	N/A	N/A
Modify system-wide startup procedures	1.76(1.0)	1.76(1.0)	N/A
Perform acceptance tests on environment software	1918(20.62)	93(1.0)	77(0.83)
User Account Management			
Create a user account group	3.52(1.02)	3.44(1.0)	5.75(1.67)
Create a new user account	3.20(1.05)	3.05(1.0)	20.20(6.62)
Add a user account to a user group	3.22(1.01)	3.18(1.0)	N/A
Copy old account characteristics to a new account	3.79(1.01)	3.74(1.0)	Not supported
Display user account characteristics	2.64(0.85)	3.11(1.0)	0.33(0.11)
Modify a user account's characteristics	2.84(1.03)	2.75(1.0)	Not supported
Remove a user account to a user group	3.69(1.22)	3.03(1.0)	Not supported
Delete a user account	3.19(1.05)	3.03(1.0)	15.80(5.21)
File size measurements		**Size change in bytes**	
Create user account group	No change	No change	10
Create a new user account	No change	No change	56
Add a user account to a user group	No change	No change	5
Disable logins for a user account	N/A	N/A	N/A
Modify a user account's characteristics	No change	No change	N/A
Remove a user account to a user group	No change	No change	Relative
Delete a new user account	No change	No change	55

In the system management activity category of user account management, the performance of the VMS/ALS and VMS/VAXSet environments are identical since they both use the Authorize utility provided within VMS. On the other hand, of the few user account management functions supported by the Unix/VADS environment, displaying only user account information takes less elapsed time in comparison to the other environments; all other user account management functions involve more elapsed time. This can be attributed to the fact that these functions are supported using Unix command scripts rather than as individual tasks of a user account manager utility.

1.3.1.3. User Interface

Throughout the system management experiments two different user interfaces were presented in both the VMS/ALS and VMS/VAXSet environments. The first, used for interaction during the ALS and VAXSet installation, was simply the view of the environment presented to the user through the DCL Command Interpreter. This interface is good with the possible exception of the limited

capabilities of the Micro-VMS window manager. Features supported include string symbols, logical names, filename wildcards, command abbreviations, command line editing, command recall, and parameter prompting. The diagnostic messages presented to the user through this interface are excellent—timely, informative, accurate, uniform in format, and consistent with the documentation. The second user interface seen within the context of the system management experiments and the VMS/ALS and VMS/VAXSet environments was that of the Authorize utility. This interface provides a uniform set of syntactically consistent commands, all with mnemonic names and qualifiers, making it quite easy to learn and use. Furthermore, Authorize utilizes the hierarchical VMS help facility for on-line command assistance and general information regarding its use.

Unlike the other environments, interaction between the user and the Unix/VADS environment was completely with the Unix Command Language Interpreter (**shell**) during both the VADS installation and account management activities. Overall, the Unix user interface is satisfactory with a notable exception being the brevity of its command names and diagnostic messages. The Unix shell is a comprehensive user interface that supports string aliases, wildcards, full pathname wildcards, command line editing, and command recall; however, it does not support command abbreviations, in-line keyword expansion, and parameter prompting. Also, the Unix command language, being targeted towards the expert user, is more difficult to learn (more odd command names to learn) than other command languages.

1.3.1.4. System Interface

Both the installation of the ALS and VAXSet environments and the maintenance of ALS and VAX Ada user accounts depend on the underlying VMS operating system. The installation of the ALS and VAXSet requires that certain VMS system parameters be modified and a new, *Ada-tailored* operating system, be built. Additionally, certain ALS and VAXSet images are installed as either known or sharable images to reduce the invocation overhead and to minimize the memory requirements of multiple environment users. The activities of account management interact with VMS in numerous ways; specifying Ada users' account resource requirements (quotas), specifying Ada users' account attributes, creating home directories for Ada users, maintaining the ALS Access Authorization File, maintaining the ALS Team Configuration File, and maintaining the VMS user account files. The Authorize utility supports establishing and/or modifying VMS user account characteristics for ALS and VAXSet users. A privileged user account can be used not only to create an ALS user's root directory (within the ALS), but also to maintain environment users' directories within the VMS file system.

As with the other environments, both the installation of VADS and the maintenance of Ada user accounts depend on the underlying Unix operating system. The installation of VADS requires

installing VADS help files (**man** pages), linking the source code debugger to the existing Unix object code libraries, and tailoring the error reporting program to utilize the standard Unix mailer. The activities of account management interact with Unix in two ways, creating home directories for Ada users, and maintaining the Unix user account files *"/etc/passwd"* and *"/etc/group"*. The Unix **root** account can be used to create an Ada user's home directory with the appropriate ownership and protection attributes. The *"/etc/adduser"* and *"/etc/removeuser"* command scripts respectively support establishing and removing Unix user accounts for Ada users.

1.3.2. Summary

The analysis across the criteria dimensions of functionality, performance, user interface, and system interface indicates that both VMS/ALS and VMS/VAXSet provide an excellent, production quality environment which is capable of supporting typical system management functions such as software installation and user account management. The analysis of the Unix/VADS environment shows that it provides adequate support for software installation activities, but that it does not provide a sophisticated user accounting model nor does it offer direct support for a majority of the typical user account management functions. Since these account management functions are relatively infrequent activities in the overall scheme of software development and maintenance, the Unix/VADS deficiencies in this area should not be considered major. However, it is clear that large projects will be better served by the VMS/ALS or VMS/VAXSet environments in the area of account management. In the area of environment installation, the major difference among the environments under investigation is the time required to load the systems. Here Unix/VADS is significantly faster than VMS/VAXSet which is significantly faster then VMS/ALS. If these installations are infrequent (as they should be), the significantly different load times should not be a matter of major concern.

1.4. Design and Development Cross Environment Analysis

The purpose of the Detailed Design and Development experiment is to exercise environment tools that support the classical programming-in-the-small activities associated with the detailed design process and code development. This section encapsulates the results of implementing this generic experiment using VMS/ALS, VMS/VAXSet and Unix/VADS by comparing the three environments. The environments will be compared using four main criteria categories: functionality, performance, user interface and system interface. This cross-environment analysis summarizes the material presented in Chapter 5 of this document.

1.4.1. Analysis Along Evaluation Criteria Guidelines

1.4.1.1. Functionality

Several areas of comparison concerning code development and program library manipulation are outlined below. This is followed by a table that compares the functionality of each environment by using the functionality checklists completed in Phase 4 of the analysis of each environment.

None of the environments support a graphical design interface and do not support detailed design, nor do they explicitly provide an Ada browser. All of the environments support the creation of inter-program library relationships and the creation, modification and deletion of package specs and bodies.

There are differing mechanisms used to establish inter-program library relationships. The ALS employs a static pointer. Changes to a child library are not reflected in the parent library without explicitly reestablishing the relationship. The VMS/VAXSet environment also uses statics pointers, but unlike ALS, unit obsolescence is propagated back to the parent library thereby notifying the user that the relationship must be reestablished. The Unix/VADS environment circumvents this problem by utilizing a library searchlist to link program libraries.

The productivity of code development can be greatly affected by the functionality of the environment editor. The ALS uses DEC's screen-oriented keypad editor EDT. EDT is not language sensitive and provides only a single window for text entry. Source code entry in the VMS/VAXSet environment is facilitated by the DEC language sensitive editor (LSE), [LSEInstall] which is based upon EDT. The LSE supports source code templates, source compilation, review of diagnostics, interaction with the VAX/VMS symbolic debugger and an on-line help facility. As a user becomes increasingly familiar with LSE usage, Ada source code entry becomes easier and proceedes with increasing efficiency. The Unix/VADS environment supports any editor supported by the Ultrix operating system. EMACS was used in this study. EMACS is a highly flexible editor that supports multi-windowing but has no language sensitive features.

Each environment varies slightly in the functionality concerning translation of source code and the creation of an executable module. The most important difference lies in the area of recompilation where VMS/ALS offers no automatic recompilation facility. Both VMS/VAXSet and Unix/VADS support automatic recompilation of modified units in the closure of a specified unit.

All of the environments support most of the program library query and manipulation functions, including listing unit names and types, determining recompilation status, removing units and clearing the program library. VMS/ALS does not provide a vehicle for querying existing inter-

program library relationships; both VMS/VAXSet and Unix/VADS support this capability. VMS/VAXSet is the only environment that provides a command to query the completeness of the program library. None of the environments provide a mechanism to query for subprogram inter-dependencies.

VMS/VAXSet and Unix/VADS provide easy mechanisms for controlling program execution: starting, terminating, suspending and resuming execution. To terminate execution of a process under VMS/ALS, one must interact with a special break-in processor, which tends to be confusing. When a program abnormally aborts, both VMS/ALS and VMS/VAXSet produce traceback listings; Unix/VADS does not.

Detailed Design and Development Functionality Checklist

Activity	Supported (Y|N)		
	VMS/ALS	VMS/VAXSet	Unix/VADS
Detailed Design			
Def./redef. objects and operations	N	N	N
Def./redef. data structures	N	N	N
Def./redef. prog. units	N	N	N
Def./redef. prog. unit interfaces	N	N	N
Design/redesign control flows	N	N	N
Create system skeleton	N	N	N
Code Development and Translation			
Create program library	Y	Y	Y
Create prog. lib. interdep.	Y	Y	Y
Develop package specs			
create package spec.	Y	Y	Y
modify package spec.	Y	Y	Y
delete package spec.	Y	Y	Y
Develop package bodies			
create package bodies	Y	Y	Y
modify package bodies	Y	Y	Y
delete package bodies	Y	Y	Y
Browse code			
find a specified object	N	Y	Y
browse a body from the spec.	N	N	N
browse a dependent (WITHed) package	N	N	N
browse a called subprog.	N	N	N
browse the parent subprog.	N	N	N
browse a specified compilation unit	N	N	N
Query and manip. prog. lib.			
list unit names.	Y	Y	Y
list unit type .	Y	Y	Y
list prog. lib. interdep.	N	Y	Y
list package interdep.	Y	Y	N
list subprog. interdep.	N	N	N
determine completeness	N	Y	N
determine recomp. status	Y	Y	Y
remove unit	Y	Y	Y
clear prog. lib.	Y	Y	Y
Translate code			
trans. into a prog. lib.	Y	Y	Y
create cross-reference map	Y	Y	N
display error messages	Y	Y	Y
list subprog. interdep.	Y	Y	N
pretty print source code	Y	N	Y
Create executable image	Y	Y	Y
Execute code			
halt/resume/terminate execution	Y	Y	Y
trace execution path	Y	Y	N
clock CPU time by subprog.	N	Y	Y

1.4.1.2. Performance

The three environments are compared by summarizing the time and space required to perform several development tasks. This data is not to be viewed as tool benchmark data, but to give a feel for the time involved with performing routine tasks in each environment.

As seen in Table TC_TABLE, the ALS takes more than an order of magnitude longer than the other two environments for program library creation and significantly longer for unit compilation and executable module creation. These differences force the VMS/ALS user into a batch mode of code development. In addition, the VMS/ALS user may have a tendency to shy away from those activities which take an inordinate amount of time, such as program library creation, as it inhibits the use of experimentation when faced with implementation decisions.

Table 1-1: Timing Comparison

- Program library creation:

	VMS/ALS	VMS/VAXSet	Unix/VADS
Elapsed time	17 min. 17 sec.	13.0 sec.	3.2 sec.
CPU time.....	12 min. 26 sec.	2.9 sec.	.1 sec.

- Small Compilation Unit (average of two package specifications):

	VMS/ALS	VMS/VAXSet	Unix/VADS
Elapsed time	1 min. 23 sec.	11.6 sec.	8.0 sec.
CPU time.....	45 sec.	5.0 sec.	.9 sec.

- Executable Module Creation:

	VMS/ALS	VMS/VAXSet	Unix/VADS
Elapsed time	7 min. 45 sec.	23.9 sec.	25.1 sec.
CPU time.....	4 min. 50 sec.	1.5 sec.	10.5 sec.

Table SC_TABLE illustrates that there are significant differences in the space required to perform the itemized tasks. It appears as if VMS/VAXSet requires much more space than the other environments. However, ALS allocates roughly 4 megabytes of space for the program library prior to the creation of the first program library. When the command is issued to create a program

library, only pointers are created, and little additional space is needed. VMS/VAXSet builds an index file for each program library that requires the indicated amount of space. Unix/VADS requires very little additional space when creating a program library.

For translation of a small unit VMS/VAXset and Unix/VADS vary only slightly, but ALS utilizes approximately 15 times the space of the other environments.

The load module created by the VMS/VAXSet environment is the smallest. The Unix/VADS module is approximately 5 times as large. The VMS/ALS module is over ten times the size of that produced under VMS/VAXSet.

Space Comparison (in bytes)**:**

Note that the measurements for ALS and DEC are accurate only to 512 bytes (1 block).

Table 1-2: Space Comparison

- Program library creation:

	VMS/ALS	VMS/VAXSet	Unix/VADS
	512	58,880	690

- Translating a small unit:

	VMS/ALS	VMS/VAXSet	Unix/VADS
Object Code	81,920	1536	378
Intermediate Representation (included in obj.).....		3584	3639

- Executable Module Creation:

	VMS/ALS	VMS/VAXSet	Unix/VADS
Executable Module	151,040	11,776	64,512

1.4.1.3. User Interface

The ALS possesses its own command oriented user interface that is syntactically modeled after Ada. The other two environments utilize the command interface of their respective underlying operating systems. The ALS lacks many of the features that have become standard in modern day operating systems. It lacks a wildcarding mechanism, has very limited command history retrieval (only the last command) and has very little tolerance for minor command input errors.

On the other hand, the VMS and Ultrix operating systems exhibit the maturity gained from over fifteen years of evolution. VMS includes a rich set of wildcard capabilities providing an easy mechanism for complex file manipulations. Command history and command line editing are supported. Command entry is flexible; command abbreviations and arbitrary parameter and option ordering are acceptable. A high degree user interface customization is possible through command aliasing and command procedures which support parameter passing, lexical functions and string manipulation.

The Ultrix C-shell was utilized as the VADS user interface. It is highly tailorable and also supports wildcarding, command history, command line editing. Ultrix command names and command syntax are less readable and less flexible than the same for VMS.

The clarity with which information is conveyed to the user, both in written and machine readable forms, is an important aspect of the user interface. This includes error messages, on-line help facilities and documentation. The ALS suffers from extremely cryptic, at times incomprehensible and inaccurate error messages. It posesses a minimal on-line help facility which is not context sensitive and does not support keyword search. The written documentation is also poor; it consists of a single volume ALS Textbook and a two volume ALS VAX/VMS Target User's Reference Manual. Navigating through the documentation is difficult, especially in the user's guide, which lacks an index.

The environment feedback provided by the VMS/VAXSet environment is good. Error messages are consistent in both quality and format. The help system serves as comprehensive on-line documentation. It is not context sensitive nor does it support keyword search. The written documentation contains information that is similar to the on-line help with the addition of illustrative examples and sample sessions.

Most of the VADS commands issue error messages in a manner consistent with the style of Ultrix, which tends to be terse. However the VADS compiler error messages are informative and clearly presented. The VADS on-line help facility presents information very much the same as Ultrix **man** pages and serves as an on-line reference manual. A simple keyword search mechanism is also supported. The documentation for VADS is well suited for a reference guide but a user's guide is lacking. The VADS Operations Manual represents a starting point for a user's guide but needs more examples and detailed explanations for command options.

1.4.1.4. System Interface

Factors considered when examining the system interface include the interaction with host operating system and level of integration of the environment tools. The ALS is an operating system built upon a host operating system. This results in the loss of accessibility to certain features and functions of the underlying operating system. Both the VMS/VAXSet environment and the Unix/VADS environment represent good extensions to existing, mature and stable operating systems.

The ALS database is implemented by a customized index file mechanism. The master index file resides in what is known as a frame file. This file grows in increments of 8000 blocks when additional space is needed. Once additional space is allocated, it is never returned. The VMS/VAXSet environment program library structure consists of a standard VMS directory plus a specially created index file. The Unix/VADS program library consists of a standard Ultrix directory plus four Ultrix subdirectories and several files.

Tools in the ALS operate in a standalone fashion. The VMS/VAXSet environment exhibits good tool integration as can be seen in the language sensitive editor. Unix/VADS tools, with a few exceptions, are standalone. The Ultrix piping mechanism allow the tools to interact easily.

1.4.2. Summary

With the exception of support for detailed design and browsing, all environments support most of the activities associated with code development. VMS/VAXSet is the only environment to provide a language sensitive editor. The ALS is the only environment that does not support an automatic recompilation facility. VMS/VAXSet and Unix/VADS perform program library creation, unit compilation and module creation well within acceptable bounds for interactive code development. On the other hand, ALS performance is conducive only to a batch mode of operation. The ALS lacks many of the user interface features that have become standard in modern operating systems such as VMS and Unix. The clarity with which information is conveyed in both written and machine readable forms is exemplary for VMS/VAXSet. Unix/VADS presents information in a manner consistent with Unix, which is terse at times. ALS information presentation is consistently poor, characterized by uninformative error messages and low quality documentation.

1.5. Testing and Debugging Cross Environment Analysis

This section presents an overall comparative analysis of the capability of each Ada environment to support typical testing and debugging activities. The intent of the Testing and Debugging experiments is to exercise the Ada debugger and to explore other tools to be used in testing. This section summarizes the results of implementing the generic experiment using VMS/ALS,

VMS/VAXSet and Unix/VADS by comparing the three environments using the criteria categories of functionality, performance, user interface, and system interface. This cross-environment analysis summarizes the material presented in Chapter 6.

1.5.1. Analysis Along Evaluation Criteria Dimensions

1.5.1.1. Functionality

VMS/ALS provides no special support for unit testing. The construction and management of test data files, as well as initial testing and regression testing, is totally manual. It is impossible to use existing test data files from within the debugger because there is no way to redirect standard input for the target program to a file.

There are no separate tools in VMS/ALS that perform static analysis. By specifying options to the compiler, certain static analysis information can be provided, including the location, scope, type, and size of each symbol, as well as the number of lines of source code, number of comments and number of times each operator and reserved word is used. VMS/ALS provides two types of dynamic analysis: statistical analysis, which yields information concerning the distribution of processing time among the subprograms in a module; and frequency analysis, which yields information about the number of times each subprogram is executed.

The VMS/ALS debugger has a basic set of functions. Breakpoints can be set, and tracepoints can be simulated. Breakpoints and tracepoints cannot be set upon the raising of an exception nor upon rendezvous with a specified task. The mechanism for specifying breakpoint locations in a subprogram is cumbersome. There are many occasions where breakpoints are not set at the requested location. In general, the debugger is not robust in its ability to manage source statement numbers and source-to-object code relationships. The debugger has commands for displaying source code and variable values, breakpoints, actions associated with breakpoints, and the call stack. With virtually no support provided for querying the status of executing tasks, debugging parallel programs is difficult in this environment.

The VMS/VAXSet environment does not provide tools to generate a test harness or test data. The Test Manager tool, however, does assist the user in organizing tests in a library, running the tests as a collection, and comparing test results. Regression testing (comparing new results against a benchmark) is therefore supported.

There is no support for the static analysis of programs in VMS/VAXSet. On the other hand, the Performance & Coverage Analyzer can thoroughly analyze the dynamic behavior of a program and produce plots or tables from the data collected. This tool collects and analyzes program

counter sampling data, I/O data, system services calls, exact execution counts, test coverage data, and page fault data.

The VMS/VAXSet debugger has a full complement of commands for setting, resetting, and displaying breakpoints and tracepoints. This includes the ability to set conditional breakpoints, set breakpoints upon a rendezvous, set breakpoints upon an exception, specify that a break action begins to take affect upon the *n*th time the breakpoint is hit, and associate a list of debugger commands with the breakpoints. All subprogram names can be made visible to the breakpoint command, thus allowing a breakpoint to be set at the entry point of any subprogram. For debugging Ada tasks, the techniques applicable to non-concurrent programs can be used, and additional commands can monitor the behavior of tasks. There are commands for stepping through the execution of a program one or more instructions at a time, for modifying the program state, and a comprehensive set of commands for querying the program state. An optional screen oriented display clearly indicates the current position of the program counter.

Unix/VADS has no test data management facility. Initial testing and regression testing are manual procedures, as is the management of test data. Unix/VADS provides no static analysis capabilities, although it does have a simple cross-referencing facility. It supports code profiling through the use of Unix tools and instrumentation code. Execution of the instrumented module produces an output file containing analysis data which is then interpreted by Ultrix tools to produce analysis reports. The granularity of analysis is at the subprogram level. Statistics provided include resident time in each subprogram and the number of times each subprogram was invoked.

The Unix/VADS debugger provides a full set of breakpoint capabilities, including conditional breakpoints and the ability to associate a list of debugger commands with breakpoints. Setting a breakpoint upon the raising of an exception is straightforward. There is no direct way to set a breakpoint upon task rendezvous. Tracepoints are not directly supported but they can often be simulated via a contrived form of a conditional breakpoint. The breakpoint command is not required to abide by Ada visibility rules. Instead, all subprograms are visible to the breakpoint command, making it easy to set a breakpoint at the entry point of any subprogram.

Unix/VADS is good at displaying the state of the program. The source code around the home position is automatically displayed in the source window when using the debugger in screen mode, or can be displayed manually in line mode. Variable values are also easily displayed. Other commands are available to display the call stack, list all active tasks, and display detailed information about the status of a particular task.

1.5.1.2. Performance

In VMS/ALS, creating modules with analysis options turned on does not affect the size of the load module. CPU time and elapsed time are increased, however, as shown in Table THREE.

Table 1-3:

Elapsed time	
without analysis	13.7 sec
with analysis	20.5 sec
CPU time	
without analysis	7.0 sec
with analysis	10.7 sec

In order to debug a program with VMS/ALS, it must be re-exported. This does not change the size of the load module. Qualitatively, the VMS/ALS debugger exhibits good response times (1-3 seconds) for all activities except loading a module, which requires more than one minute of elapsed time.

In the VMS/VAXSet environment, the creation of test descriptions and test collections takes only a few seconds. Running a collection of tests is also rapid, and regression testing on a small test suite takes less than 20 seconds. The Test Manager stores benchmark files, test system files, and (in a subdirectory) files relating to each collection in a special user-defined directory.

The collection of performance and coverage data in VMS/VAXSet is rapid, except for execution counts which, by their nature, incur high overhead. Once the data is collected, it can be analyzed and displayed with only a few seconds delay. Instrumenting code for analysis incurs negligible space overhead.

Using the VMS/VAXSet debugger does not appear to slow down the execution of a program, although this is hard to judge because the act of debugging a program in execution (setting breakpoints and tracepoints, displaying variable values and call stacks, etc.) naturally takes longer. Compiling and linking a program for debugging does increase the space utilization of the object code (by about sixty percent) and the executable (by about eighty percent).

Test management is not supported in Unix/VADS, nor is static analysis of programs. The Unix dynamic analysis tools were not available for this experiment, and consequently there are no performance results to report in this area. No recompilation or instrumentation of the Unix/VADS load module is necessary for debugging. Qualitatively, the debugger exhibits excellent response times for most functions.

1.5.1.3. User Interface

The VMS/ALS debugger is easy to learn but cumbersome to use due to its command-line orientation. Not only is there no automatic feedback concerning execution location, there is no source code window. Thus, the user must repeatedly display source code. In some cases, the source display is difficult to read. The VMS/ALS command level help facility is available from within the debugger. This facility provides a short description of each debugger command and can serve as a useful reminder of the available debugger commands. Error messages from VMS/ALS are confusing, and the display of information tends to be poor. Certain commands (for example, the setting of breakpoints) operate in a manner inconsistent with the documentation.

The VMS/VAXSet Test Manager [TMInstall] tool has a clear user interface and is easy to learn. The three steps in unit testing which are automated -- organizing tests into collections, running these tests with previously-written test data, and comparing test results to expected results -- flow in a natural order under the control of the Test Manager. At each stage, the tool interacts with the user by reporting on the results of running tests and comparing results.

The Performance & Coverage Analyzer [PCAInstall] in VMS/VAXSet is also easy to use. The Collector accepts defaults for the name of the file to contain collected data and for the type of data to collect. The Analyzer operates in screen mode, and the plots and tables it generates are readable and well labeled.

The VMS/VAXSet debugger can operate in line mode or screen mode. In screen mode, the debugger displays a window of source code with a cursor at the current statement location. Setting breakpoints and tracepoints, querying the state of the program, and modifying the state of the program are all straightforward. Overloaded symbols are detected and can be qualified. On-line help is accessible from within the VMS/VAXSet debugger, the test manager, and the performance analyzer.

The Unix/VADS debugger is easy to use and easy to learn. It requires knowledge of only a few basic commands to start using it effectively and allows for a natural progression to more advanced usage. The debugger offers both full screen and command line oriented user interfaces. The majority of commands can be used in either the screen mode or the command line mode of operation. In addition, there are special commands applicable only to the screen-oriented interface. These commands facilitate cursor movement and source window movement and also duplicate a subset of the command line functions. The debugger handles overload resolution elegantly. In the important area of task debugging, the Unix/VADS debugger proved quite satisfactory.

When using the screen-oriented interface to the Unix/VADS debugger, a window of source code around the home position is automatically displayed. Other aspects of the program state are very easily solicited. The entire Unix/VADS on-line help facility is accessible from within the debugger. The one area of weakness in this environment is a lack of consistently clear error messages.

1.5.1.4. System Interface

The VMS/ALS debugger is a standalone tool, not integrated with the editor or compiler. No VMS commands are accessible from the debugger or from the ALS, but the ALS command level is accessible from within the debugger.

With respect to VMS/VAXSet, the debugger, the dynamic analyzer, and the test manager are tools that can be used across a wide range of applications and programming languages. Where these tools need to keep track of auxiliary data, they store this data in files in VMS directories. From within the debugger, the user can invoke the editor or the Ada compiler but must re-link and re-execute the modified program to initiate a subsequent debugging session. That is, the debugger is not fully integrated with other tools.

Like the VMS/VAXSet debugger, the VADS debugger is not tightly integrated with the other tools in its environment. Specifically, source code cannot be modified and then incrementally recompiled from within the debugger. Editors are accessible from within the debugger to alter source code. The analysis tools are those provided by Unix.

1.5.2. Summary

The VMS/ALS unit testing and debugging tools provide a rudimentary set of functions and there is no test manager. Limited support is provided for static and dynamic analysis. The debugger is weak in its support of breakpoints and in its ability convey the program state. It also provides little functionality to facilitate task debugging. The debugger does not consistently operate as stated in the documentation and frequently responds with cryptic and uninformative messages.

The VMS/VAXSet environment demonstrates that separate tools can work together smoothly in a powerful way. The tools for test management and for dynamic analysis are functionally rich, easy to use, and more than adequate with respect to performance. The debugger is excellent. Clear error messages, a hierarchical on-line help facility, and comprehensive documentation contribute to a pleasant user interface.

Tools for the management of tests and test data are absent from Unix/VADS, there are no static analysis tools, and dynamic analysis are limited and are borrowed from Unix. The Unix/VADS debugger can enhance the unit development process. It supports task debugging and the revealing of the program state. Comprehensive on-line help is available, and the screen-oriented inter-

face greatly facilitates debugger usage, however, error messages need to be improved. Overall, the Unix/VADS debugger is an excellent tool.

Unit Testing and Debugging Functionality Checklist

Activity	Supported (Y|N)		
	ALS	VMS/VAXSet	Unix/VADS
PRIMARY ACTIVITIES			
Unit testing			
Create and debug test harness	N	N	N
Create test input data for functional testing	N	N	N
boundary case testing	N	N	N
structural testing	N	N	N
stress testing	N	N	N
Perform initial test			
create expected output data	N	N	N
produce actual output data	Y	Y	Y
compare actual and expected data	Y	Y	Y
Perform dynamic analysis			
measure execution time by subprogram	Y	Y	Y
perform test data coverage analysis	N	Y	N
identify code not executed	N	Y	N
measure statement execution frequency	N	Y	N
Perform regression testing	N	Y	N
Debugging			
Set/reset breakpoints on			
program unit entry/exit	N	Y	Y
exception	N	Y	Y
statement	Y	Y	Y
n'th iteration of a loop	N	Y	N
variable changing value	N	Y	Y
variable taking on a specified value	N	Y	Y
rendezvous	N	Y	N
Control execution path			
jump n statements	N	Y	N
enter a specified subprogram	N	Y	N
exit the current subprogram	N	Y	N
Query program state			
display source code	Y	Y	Y
display variable values	Y	Y	Y
display breakpoints	Y	Y	Y
display tracepoints	Y	Y	Y
display stack	Y	Y	Y
display history	N	N	N
display task status	N	Y	Y
Modify program state			
modify variable values	Y	Y	Y
add, modify and delete code	N	N	N

SECONDARY ACTIVITIES

Unit testing

Perform static analysis			
check against prog. guidelines	N	N	N
measure subprogram's complexity	N	N	N
identify unreachable statements	N	N	N

Debugging

Set/reset tracepoints on			
program unit entry/exit	N	Y	Y
exception	N	Y	N
statement	Y	Y	Y
n'th iteration of a loop	N	Y	N
variable changing value	N	Y	N
variable taking on a specified value	N	Y	N
rendezvous	N	Y	N

1.6. Project Management Analysis

The project management experiment took into consideration only a small portion of the total functionality which should be attributed to project management. As more environments support project management activities, it will become increasingly important to expand the scope of this experiment area. The purpose of this particular experiment was to explore the activities surrounding the building and maintaining of the project database which is but one aspect of the technical management activities of project management. Chapter 7 describes the whole realm of project management activities, but defines an experiment for key technical management activities only. Since the experiment was run only on Apollo hardware using the Alsys compiler, this section will describe only the lessons learned from a single instantiation of the experiment.

1.6.1. Analysis Along Evaluation Criteria Guidelines

1.6.1.1. Functionality

The DSEE/Alsys environment represents a great unfulfilled potential. DSEE represents some of the best technology available for project management and would be the Alsys Ada compiler is one of a family of Ada compilers from Alsys. Unfortunately, these two systems are provided by separate vendors and are not integrated so that they can be used usefully together. A project database consists of three fragmented libraries and directories. The DSEE library is used most profitably to hold program source and documentation text. DSEE provides configuration management, version control, file locking, and a history mechanism to record reasons for changes and maintaining file history. The Alsys Ada library must be used to hold compiled Ada modules before their assembly into running programs. It must be used to maintain the Ada program dependency relationships between files. Finally the Unix file system must be used to hold ex-

ecutable program files and any miscellaneous files required which do not fit in the DSEE or Alsys Ada libraries.

The DOMAIN/IX operating system is the basis for all the database elements described above. Both the DSEE and Alsys Ada libraries consist of a series of directories and files which can be examined using operating system commands. Safe and controlled access to these files is limited, however, to the utilities provided by the program creating the libraries.

The Apollo DOMAIN system provides a distributed file system built around a proprietary local area network (LAN). All elements of the database can make use of the distributed file system. A project database can be divided among multiple workstations, and each component of the database is fully accessible from any other workstation. Unlike the standard versions of either Unix or VMS, the Apollo Domain operating system supports a network file system that is transparent to the user.

Among the features supported by DSEE are a file control system supporting version control and configuration management, system building, monitors for automatic notification when file changes are completed, and tasklists for project tracking. Some of these capabilities (such as system building) are provided by Alsys Ada and are not fully compatible with DSEE.

1.6.1.2. Performance

During this experiment, all measurements were taken under conditions of light load for both network and local tests. When typing commands directly to the user interfaces, responses were consistent with interactive use, with the exception of Ada compilation. When Ada compilation was invoked from DSEE, builds were prolonged in comparison to the DEC and Verdix compilers running on MicroVAX hardware. Extremely simple programs required several minutes to complete. Creating, copying, and deleting the modest sized program libraries used in this experiment required less than a minute to complete.

Operating system commands, when invoked from DSEE, took approximately four seconds while the same action required only 1 second when invoked from Alsys Ada. Operations performed over the network took no longer than operations performed on the local workstation, suggesting that at least for lightly loaded systems there is little network overhead. For the most part, the performance was considered consistent with interactive use for the functions tested.

1.6.1.3. User Interface

A variety of user interfaces were provided in the course of this experiment. First, the DOMAIN/IX operating system and Apollo hardware provide a bit-mapped windowing console screen. Second, DSEE provides a consistent user interface that utilizes the Apollo hardware. Third, the Alsys Ada

user interface is command based since it is meant to be portable and is available on a number of different systems. Each of the user interfaces was competent for the purpose designed. Each provided help on demand and error messages as required. The error messages were normally helpful.

Documentation was provided in a minimum of four separate manuals: a DSEE Reference, a DOMAIN/IX Reference [IX], an Aegis Reference [SCR], and an Alsys Ada [AlsyComp] manual. The Apollo manuals were of high quality, relatively complete, and well indexed. Illustrations were used as required, along with appropriate examples. An on line tutorial provided with DSEE was useful for initial exposure. The Alsys Ada manual was fair. It did not cover all topics needed during the experiment, and the command information was scattered in a fashion that made it difficult to locate the individual commands without consulting the table of contents. For example, there was no information on invoking the compiler directly from the operating system, the information on PRAGMA INTERFACE was incorrect, and the command summary was incomplete.

The user interface as suffers from the inconsistency one would expect when it is provided by two separate vendors. Separate commands and command styles are required by Alsys Ada, DSEE, and the two command shells. In addition, some important administrative functions must be performed by Aegis (the native operating system) rather than DOMAIN/IX (the Unix operating system) since there are no equivalents that will work with the DSEE database.

1.6.1.4. System Interface

Since there are three software components (DSEE, Alsys Ada, and the underlying operating system (DOMAIN/IX or Aegis)) there are several system interfaces to consider. The DSEE to Alsys Ada interface is the weakest interface link. The DSEE build operation assumes Unix-like compiler invocations, but Alsys Ada uses their own format for compiler invocations. While all possible solutions to the problem were not explored, the solution to this inconsistency is at best problematical and was not overcome.

The DSEE/operating system interface works reasonably well. The direct invocation of shell commands is surprisingly sluggish, but a user with a windowing terminal will make little use of this feature. Versions of DSEE files can be read into the operating system file space as required for modification, printing, or other uses. DSEE also can provide correctly versioned files for a number of operating system translators including compilers, text formatters, and other utilities.

The Alsys Ada/operating system interface also functions well. A user reads files to the Ada front end which processes them and, sometimes, writes the results back to the operating system's file space. Internal Alsys Ada files (notably the program binary files) are more or less inaccessible,

except via the intervention of Alsys Ada. However, combinations of binary files (e.g. the construction of an Ada program that calls C functions and libraries) works satisfactorily.

One major flaw in the system interface for database administration is the lack of copy facilities. Both DSEE and Alsys Ada lack any provision for copying individual, internal format files that the database administrator can use. This means that it is impossible to cleanly copy files between DSEE libraries. Instead one must copy files a version at a time. Likewise, Alsys Ada files can only be copied by recompilation. On the other hand, libraries as a whole *can* be easily copied.

1.6.2. Summary

The Apollo DSEE represents an interesting approach to some of the problems encountered in building and maintaining large software systems of programs and documentation. From the standpoint of technical management, it offers the features of:

- Distributed database with full functionality across a local area network
- Decentralized structure, allowing individuals to create personal database elements
- Automated allocation and return of disk space
- Maintenance of all versions of program source
- File locking for modification

Problems with the environment for Ada development and maintenance include poor integration of the Ada compiler and DSEE, lack of individual file copy capabilities, and a multiplicity of user interfaces which presents an inconsistent system. Since the DSEE product is currently available only on Apollo hardware it is of limited value to those with other hardware. The Apollo/Alsys combination does deserve further scrutiny if the companies get together to provide a more integrated Ada environment with a consistent user interface.

1.7. ACEC Analysis

This section presents an overall comparative analysis of the ability of each Ada compiler to compile, link and execute a set of test programs, consisting of:

- the 286 programs in the IDA/ACEC test suite
- the programs comprising the ACEC support software
- a large and a medium-size program (used as independent checks).

For each Ada compiler under investigation (in order, ALS, DEC, and VADS), the aggregate results of compiling, linking and executing the set of test programs will be compared along the criteria dimensions of functionality and performance. (The user interface and system interface to the compilers have been examined already as part of the Design and Development experiment.)

This cross-environment comparative analysis summarizes the material presented in Chapter 8 of this document.

1.7.1. Analysis Along Evaluation Crteria Dimensions

1.7.1.1. Functionality

ALS failed to compile 6 of the ACEC tests; DEC and VADS compiled all tests. A total of 21 ACEC tests did not execute correctly under ALS; all tests appeared to execute correctly under DEC; 4 tests did not execute correctly under VADS.

ALS did not compile the ACEC support software as given - the INSTRUMENT package had to be dissected, the IO_PACKAGE specification and body had to be compiled separately and **pragma** ELABORATE(TEXT_IO) had to be added to certain packages. DEC and VADS compiled the ACEC support software with or without these modifications. It is worth noting that the ACEC was developed using the VMS operating system that supports the DEC compiler.

ALS failed to compile a 'large' program of 2200 text lines (excluding comments), whereas DEC and VADS did compile it. All three systems compiled a 'medium-size' program of 1078 lines.

1.7.1.2. Performance

Running the complete ACEC test suite required nearly 5 days for the ALS and less than half a day for DEC or VADS.

For each of the test programs, three groups of measurements were taken:

Compilation	The facilities of the underlying operating system were used to measure the total time required to compile and link each program to produce an executable image. The O/S facilities were also used to measure the size of the object code module.
Instrumentation	Each test program was instrumented to record its execution start and stop times by calling the appropriate procedures in the ACEC support software.
Run-time	The facilities of the underlying operating system were used to measure the time required to execute each test program

In each group, two types of time were measured;

- "elapsed time" - which can be defined as "wall clock time", and
- "CPU time" - which is the portion of this time that the processor is busy executing the program.

Since the tests were run on dedicated processors, the difference between the two times is the time spent waiting for I/O (plus the time consumed by any residual system daemons).

The results are summarized in Table AMM.TABLE; times are rounded to tenths of seconds and code size is in bytes. The numbers in parentheses are the ratios of the ALS and VADS measurements relative to the DEC measurements. All problem programs have been eliminated and each group of measurements is for an identical set of programs on the three systems. (Table AMM.TABLE is restricted to the 'test' versions of the ACEC programs; the measurements for the 'control' versions have been omitted).

Table 1-4: Arithmetic Mean of Measurements for ACEC Tests

Times in seconds, code size in bytes (figures in parentheses are ratios with respect to DEC).

	ALS	DEC	VADS
Compilation_Quantity			
Elapsed Time	817.0 (15.6)	52.5 (1.0)	61.9 (1.2)
CPU Time	505.0 (33.6)	15.0 (1.0)	44.9 (3.0)
Object Code Size	224012 (93.6)	2392 (1.0)	2601 (1.1)
Instrumentation Quantity			
Elapsed Time	23.8 (1.5)	16.1 (1.0)	22.9 (1.4)
CPU Time	23.3 (1.5)	16.0 (1.0)	0.2 (0.0)
Run-Time Quantity			
Elapsed Time	47.6 (1.7)	28.6 (1.0)	36.0 (1.3)
CPU Time	29.0 (1.7)	16.8 (1.0)	22.7 (1.3)

The object code size for ALS is anomalous because it measures the size of an 'object container' which holds much more than just the straightforward object code. The instrumentation CPU time for VADS is also anomalous; in fact the reliability of the instrumentation results is questionable.

The compilation results for the 'medium sized' program are shown in Table AMCM_TABLE.

Table 1-5: Arithmetic Mean of Compilation Measurements for 'Medium-Size' Program

Times in seconds, code size in bytes (figures in parentheses are ratios with respect to DEC).

	ALS	DEC	VADS
Elapsed Time	2242. (14.4)	156.	123. (0.8)
CPU Time	1019. (9.6)	106.	108. (1.0)

The relative mean values in Table 1-4 show clearly that, for small programs such as the ACEC tests, ALS is by far the slowest compiler, and that VADS is marginally slower than DEC. However at run-time, the disparity between ALS and VADS narrows dramatically, with VADS again marginally slower than DEC. The instrumentation elapsed times mirror this trend, but the instrumentation CPU times do not seem useful (too many individual zero measurements).

The single, independent test on the 'medium size' program indicates that, as program size increases, the performance of the VADS compiler improves slightly with respect to DEC, and the performance of the ALS compiler improves dramatically.

The ACEC tests are divided into various architecture categories and each test is designed to test one or more specific features of the Ada language. The only significant variation of mean values across architecture categories is that, in the Optional-Features category, ALS and VADS perform better relative to DEC than in the other categories, in some cases even outperforming DEC; but this category consists of only three tests and is thus too small a sample to warrant any conclusions. The relative mean values between compilers for individual language features are roughly consistent with the overall mean values. There is considerable variation across language features, but no clear trend.

The main purpose of the ACEC tests is to derive differential statistics for individual language features; that is to measure the *increment* in time or space caused by using a particular feature. Unfortunately, no conclusive differential statistics were obtained (many differentials were in fact *negative*!). After filtering out the clearly erroneous data, the relative ratios for time and space utilization were roughly consistent with the aggregate results and showed considerable variation between language features, but without any clear pattern.

1.7.2. Summary

Judging from the analysis across the criteria dimensions of functionality and performance, it is clear that both the DEC and VADS provide reasonably robust Ada compilers with adequate performance both for the translation phase and for the execution phase, but with the DEC compiler being somewhat better. The analysis shows that the ALS provides a fragile compiler with poor performance for the translation phase, although the ALS performance for the execution phase is comparable to the other two systems.

References

[ACS 85] Developing Ada Programs on VAX/VMS.
Digital Equipment Corporation. 1985.

[ALS Text 84] Ada Language System Textbook.
SofTech Inc., 1984.

[Apollo Compiler Reference 86]
Apollo Domain Compiler User's Guide.
Alsys Inc. Waltham MA 02154, 1986.

[CMS User's 84] User's Introduction to VAX DEC/CMS.
Digital Equipment Corporation Maynard, Massachusetts, 1984.

[DOMAIN/IX 86] DOMAIN/IX User's Guide.
Apollo Computer Inc. 330 Billerica Road, Chelmsford, MA 01824, 1986.

[LSE Install 86] *VAX DEC/Language Sensitive Editor Installation Guide*
Digital Equipment Corporation Maynard, Massachusetts, 1986.

[MMS Installation 84]
Installing VAX DEC/MMS.
Digital Equipment Corporation Maynard, Massachusetts, 1984.

[PCA Installation 85]
VAX DEC Performance Coverage Analyzer Installation Guide
Digital Equipment Corporation Maynard, Massachusetts, 1985.

[Run-Time Reference 85]
VAX Ada Programmer's Run-Time Reference Manual.
Digital Equipment Corporation Maynard, Massachusetts, 1985.

[SCR 86] DOMAIN System Command Reference.
Apollo Computer Inc. Chelmsford, MA 01824, 1986.

[TM Installation 85]
VAX DEC/Test Manager Installation Guide.
Digital Equipment Corporation Maynard, Massachusetts, 1985.

[VADS Operations 85]
VADS Operations Manual Version 5.1.
VERDIX Corporation, 1985.

2. Methodology for the Evaluation

2.1. Introduction

2.1.1. Objective

An important goal of the Software Engineering Institute is to assess advanced software development technologies and disseminate those that appear promising. A number of Ada Programming Support Environments (APSEs) are being developed, and more are certain to follow. Such environments could play key roles in increasing the productivity of software engineers, in improving the quality of embedded systems software, and in reducing the cost of producing and maintaining software.

The initial purpose of the *Evaluation of Ada Environments* project was to determine the suitability of the Army/Navy Ada Language System (ALS) and the Air Force Ada Integrated Environment (AIE) for application to software engineering activities. The ALS was delivered to SEI in late 1985 and the AIE has yet to be completed as of the middle of 1986. Early in the project a decision was made to develop a systematic methodology for evaluation of environments and to include a number of commercial environments in the study. These environments must be evaluated with respect to their ability to support the complete software life cycle.

Most of the work on the evaluation of software development environments has fallen into one of three categories. First, there are evaluations of particular components, such as compilers, editors, or window managers (e.g. [ROBERTS83], [HOOK]). These evaluations are useful in their own right, but they fail to consider global aspects of the environment or how components interact. Second, there are evaluations of particular environments (e.g., [BRINKER], [BRINKER85A]) that usually consider the tools available in that environment, but they do not lend themselves to cross-environment comparisons. Third, there are lists of questions and criteria without the details of how to answer the questions or how to apply the criteria (e.g., [LYONS]). These lists are useful, but they are frequently difficult to apply.

There has been a long tradition of using benchmark tests and test suites for evaluation of hardware and certain kinds of software, such as compilers. The idea is to find a representative set of programs that provide a common yardstick against which competing products can be measured. These tests must be chosen carefully to reflect accurately the applications in the working environment of interest. For example, installations running commercial applications may emphasize input/output performance, whereas installations running scientific applications may emphasize floating point computations.

Machine or compiler benchmarking usually is aimed at a small set of performance measurements. The purchaser of a machine may be interested in throughput for batch-oriented systems and response time for interactive systems. The purchaser of a compiler may be interested in compile time in terms of statements translated per minute, in execution speed, or in the size of the resulting object code. In running a benchmark test, the tester attempts to maintain as many constants as possible, varying only those components to be compared.

The evaluation of environments is far more difficult than the evaluation of single components. There is much more diversity to consider and often no mapping from the set of tools in one environment to the set in another. For example, it is difficult to compare an environment that performs the traditional edit-compile-link-execute cycle for program unit development with an environment that performs incremental compilation during the edit step. The other complication in environment evaluation is the pervasiveness of the user in the process of software development: Performance variation among different software developers may exceed an order of magnitude.

The purpose of defining a methodology is to add a degree of rigor and standardization to the process of evaluating environments. Without a systematic approach, evaluations offer little more than *ad hoc* evidence of the value of an environment. The purpose of this chapter is to discuss a methodology that addresses the shortcomings of previous approaches to environment evaluation. The methodology is comprehensive, repeatable, extensible, user-oriented, and partly environment independent. This methodology has been applied to several Ada environments at the Software Engineering Institute allowing them to be compared objectively according to the same criteria. The following sections provide some background and previous work in environment evaluation and then discuss the principles for an effective environment evaluation methodology, its phases, and the evaluative criteria. Finally, we describe the environments which were evaluated and give a summary of the experiments conducted.

2.1.2. Previous Work

This study was not undertaken in a vacuum. A great deal of work has been done already in the evaluation of environments, and much of it can be applied to Ada environments. The purpose of this section is to give a brief overview of the previous work that has been done in this area. The impression that one gets after conducting a literature search and talking to the people interested in evaluation is that there has been a good deal of thinking about the problems of evaluation, but not much in the way of actual work. What has been done is mainly in the area of evaluation of editors and compilers. Evaluation has taken a back seat to studies of correctness and validation. Unless we have missed some important work, we believe that this study represents a venture into virgin territory.

2.1.2.1. Software Engineering Institute

The Software Engineering Institute has undertaken in its first year two related activities which provide input to this study. They are the Technology Identification and Assessment project and the Software Factory Workshop project [SFW]. The first project was undertaken to identify and assess available technologies that have potential to improve the software engineering process. The latter was undertaken to bring experts in the field together to explore existing technologies and ongoing efforts to develop new technologies to meet the objectives of automated software factories.

The Technology Identification and Assessment project is producing a series of assessment reports [Ellison86, Feiler85, Kellner85, Nestor86, Newcomer85] covering the areas of user interface technology, tool interface technology, database technology, distributed computing technology, and programming environment technology. Some of these reports are more applicable to the evaluation of APSEs than others, but taken as a whole, they represent a good approximation of the current state of the art and provide an indication of what is possible and desirable in APSEs.

The Software Factory Workshop project consisted of a series of workshops, some with restricted attendance, which culminated with an open workshop in Pittsburgh in February 1986. The purpose of these workshops was to explore means of transforming the labor intensive process of developing software for embedded systems into a more automated capital intensive process. The "software factory" is the embodiment of all those tools and techniques which can enhance the process and the product of software engineering. The initial meetings took place in Pittsburgh in March and April and in Morgantown, West Virginia, in October of 1985. The results of these workshops have provided a strong influence on the development of the criteria and desirable functionality for APSEs. The final report for the Software Factory Workshop project was submitted at the end of April, 1986.

2.1.2.2. Stoneman

One of the earliest efforts at specifying the requirements for a programming environment specifically for the Ada programming language was written by Buxton and Stenning [STONEMAN]. This document is known as "Stoneman" and introduced the concepts of the Kernel Ada Program Support Environment (KAPSE) and Minimal Ada Programming Support Environment (MAPSE) as an approach to providing portability. It specified the purpose of an APSE: "to support the development and maintenance of Ada applications software throughout its life cycle, with particular emphasis on software for embedded computer applications." It divided the APSE into three principal components: the data base, the (user and system) interfaces, and the toolset.

The shortcomings of Stoneman have been well-documented [STONEANL], and there has been a migration to a somewhat different concept of portability with the Common APSE Interface Set (CAIS) [CAIS]. There is a Stoneman Working Group within STARS which has already produced one update of Stoneman [STONEMANII] and is working on another. This series of work on the requirements of an APSE provides the context in which this study is undertaken.

2.1.2.3. STARS Software Engineering Environment

STARS (Software Technology for Adaptable Reliable Systems) is one of three efforts, along with the SEI and the Ada Joint Program Office (AJPO), which constitute the DoD's Software Initiative. The STARS Joint Program Office formed a STARS Joint Service Team for Software Engineering Environments. This group produced version 1.0 of an Operational Concept Document which describes the mission, functions, and characteristics of the STARS Software Engineering Environment (SEE) [SEE]. The purpose of the STARS-SEE is to define the benchmark for acceptable SEE capability, to provide a framework and technical interface standards and guidelines used to build SEEs, and to ensure compatibility among SEEs and associated software tools.

2.1.2.4. IDA Benchmarks

Under contract to AJPO, the Institute for Defense Analyses (IDA) has produced a Prototype Ada Compiler Evaluation Capability (ACEC) [HOOK]. This is a test suite collected from Ada test programs that have been in the public domain for some time. These include contributions from IBM, SRI, Harris, Ada Fair '84, and SigAda and were originally collected by the Evaluation and Validation (E&V) Team. The test suite was designed to be a representative sample of programs which measure Ada language feature performance. The tests have been instrumented to provide performance and capacity statistics as well as differential execution statistics for the various language features.

These IDA benchmark tests were used as the primary vehicle for evaluating the Ada compiler itself. This is in keeping with the desire to avoid replication of the work done by others and our emphasis on components of the environment other than the compiler.

2.1.2.5. Evaluation and Validation Team

In June of 1983, AJPO defined the E&V Task and established a tri-service Ada Programming Support Environment (APSE) E&V Team, with the Air Force designated as the lead service [EVPLAN]. The overall goal of the E&V Task is (1) to develop the techniques and tools that will provide a detailed and organized approach for assessing APSEs and (2) to determine conformance of APSEs to the CAIS. The Team consists of 30-40 members and has met quarterly since December, 1983. The Team has also held annual E&V Workshops at which distinguished reviewers have had the opportunity to comment on the documents produced by the Team. The Team has four working groups:

- The Requirements Working Group (REQWG) is producing the functional requirements for the evaluation and validation of APSEs.
- The APSE Working Group (APSEWG) is gathering information and obtaining expertise on the three government sponsored APSEs (the ALS, ALS/N and the AIE) as well as commercially available APSEs.
- The Coordination Working Group (COORDWG) is identifying technical issues, gathering technical information and interfacing with the public.
- The Standards Evaluation and Validation Working Group (SEVWG) is exploring issues related to the CAIS interface standard.

The REQWG has produced a document called the Requirements for Evaluation and Validation of Ada Programming Support Environments [EVREQTS]. This document, which is being continually updated, levies the requirements for a comprehensive evaluation of APSEs. Another related document, the E&V Classification Schema Report [EVSCHEMA], provides a framework and organization for an E&V Reference Manual [EVREF] which provides information on the classification of APSE components and identifies the criterion/standard or metrics capability used to assess a particular component. An E&V Guidebook ([EVGuide] is meant to provide detailed descriptions and instructions regarding application of E&V techniques or references to other documents where such details may be found. The Reference Manual and Guidebook have been delivered in draft version to the E&V Team.

The COORDWG has kept track of APSE evaluation efforts and produces a regular report which updates this information. Many of the efforts described in this section are described in greater detail in their Technical Coordination Strategy Document [EVCOORD]. The APSEWG maintains an APSE Analysis Document [EVANAL], which provides descriptions and taxonomies of the features provided in APSEs developed by the DoD.

2.1.2.6. Ada-Europe

The Environment Working Group of Ada-Europe is chaired by John Nissen. They have recently released a document entitled *Selecting an Ada Environment* [LYONS], which is being published by Cambridge University Press. The group's meetings are partially funded by the Commission of the European Communities. The guide is meant to be used in the selection of APSEs, but it could be used also for the specification of Ada environments. The book provides considerable help in understanding the issues of specifying, implementing, selecting, using, and producing standards for environments. The book divides the issues into six parts: host and target considerations, kernel, aids for tool building, man-machine interaction, tool functions, and other issues. Each of the 19 chapters ends with a series of questions which help characterize an environment. This excellent reference has helped to focus and clarify our understanding of environments.

2.1.2.7. Other Evaluations of APSEs

There are at least three known evaluations of the ALS environment. Two of the studies were funded by the DoD. They are the study done by System Development Corp. (SDC) for Ballistic Missile Defense in Huntsville, Alabama, and the study done by GTE for the Army WIS at Ft. Belvoir, Virginia. NASA has performed evaluations of the ALS [BRINKER85A] and the DEC VAX Ada [BRINKER]. These studies provide much information but they are less systematic and comprehensive than the study described in this report and they are not easily used for cross-environment analyses.

2.1.2.8. Academic Studies

A study done by the Wang Institute of Graduate Studies by Mark Ardis, *et al.* [ARDIS] is titled *An Assessment of Eight Programming Environments*. Their report deals primarily with "programming-in-the-small" environments, which concentrate on only the program development portion of the life cycle. The work is interesting, however, for the methodology that was chosen. Each of the coauthors has ranked the eight environments according to nineteen features. The environments are also ranked according to their support for various user tasks. These rankings are boiled down to a three level scale of importance and support. This work not only provides some interesting ideas for a methodology, but also lists some useful features of the programming portion of the life cycle.

An extensive study was undertaken at the University of Maryland to monitor an Ada Software Development project [BASILI84]. The project began in February 1982 and ended in July 1983 and was supported in part by the Office of Naval Research and the Ada Joint Program Office. Among the goals of the project were to develop a set of metrics which could be used to evaluate APSEs. Unfortunately, at the time of the experiment there were no integrated APSEs and only rudimentary Ada compilers. The New York University (NYU) Ada/Ed interpreter was used and logistical and performance problems caused early termination of the project. The primary results described in the final report were the number of errors and the changes required at each phase of the life cycle. These errors are broken down in various ways including the type of programmer making the error, the type of the error, and the language feature associated with the error. This study is useful from the point of view that it can be used to make our experiments more realistic with respect to error seeding.

The Georgia Institute of Technology is the site of the Software Test and Evaluation Project (STEP). This group is supported by the Army Institute for Research in Management Information and Computer Science (AIRMICS). They have produced a software test tools baseline which has been located as a result of an extensive literature search [STEP]. While this initial list is concerned with languages other than Ada, it does provide a taxonomy for test tools as an indication of what should be possible in an APSE.

2.1.2.9. User Interface Studies

There has been a significant body of experimental work done in the area of user interfaces, particularly in the area of text editors [ROBERTS83]. This work has been primarily a collaborative effort between Xerox Palo Alto Research Center and Carnegie-Mellon University. The researchers in these efforts include Card, Moran, Newell, and Roberts [CARD80]. These studies are important from the perspective of providing a user or task oriented approach to the evaluation rather than a function or feature oriented approach. This is the same approach that is adopted here for the evaluation of APSEs.

2.2. Principles

Any sound methodology should be based on a set of principles that represent the underlying philosophy from which the methodology is derived. Six principles form the basis of the SEI approach and govern the the methodology:

- environment independent,
- user oriented,
- experimentally based,
- emphasizing primary functionality,
- evolutionary,
- extensible.

These principles are described in detail below, with our rationale for selecting them.

2.2.1. User Oriented

Perhaps the most important criterion for developers of an environment evaluation methodology is to focus on the activities of the users rather than on the tools provided by the environment. This focus on user activities provides the common ground for comparing environments. This approach is adopted by Roberts and Moran in their evaluation of text editors [ROBERTS83]. Some of the basic operations of text editing include inserting text, deleting text, searching for strings, and replacing one string with another. The actions required to accomplish these tasks vary greatly depending on the type of editor one is using (line oriented versus screen oriented, for example).

In the context of environments, one should postpone as long as possible the issues relating to particular tools of the environment. For example, given that a generalized user interface could be command based, menu based, graphically based, or some combination, it is best initially to base an evaluation on the interactions the user needs to have with the system rather than the mechanisms needed to perform them. Eventually the evaluation must consider the details of implementation; but by postponing decisions as long as possible, the evaluator keeps the process environment independent longer, thereby avoiding the tendency to assume that locally optimized solutions are global solutions.

2.2.2. Environment Independent

The principle of environment independence is motivated by two factors. First, it ensures that the methodology is not biased for or against any existing environment. Second, it means that much of the difficult work can be performed once for all environments, while the more mechanical parts of the evaluation take place for each environment. The mechanism for ensuring environment independence depends on the first principle (basing the evaluation on user activities) as well as making sure that the initial formulation of any criteria and tests is generic.

The approach of focusing on the activities of the users rather than the tools requires a great deal of care, forethought, and expertise. The implementor of the environment-independent part of the process must be knowledgeable enough about environments to understand the complex interplay between tools and activities. Experience with a large number of environments is necessary, but so is independent judgement as to what the underlying user model is or should be.

2.2.3. Experimentally Based

The evaluations should be based on the analysis of the results of a number of well-defined experiments. This principle ensures that the evaluation is objective and repeatable. The experiments should be based on different activities of the life cycle, and the steps of the experiments should be defined rigorously. Accompanying the experiments should be questions and measurements that must be recorded as the experiment is carried out. These will not be experiments in the sense of controlled experiments using many subjects performing the same task, but logically connected sequences of tests performed by a single experimenter or team. These sequences of tests provide a concrete mechanism for answering questions about environments.

The objectivity of the experiments will be demonstrable if they yield the same or similar results and the same or similar conclusions when applied by different groups. Although it is desirable to be as objective as possible, certain criteria do not lend themselves to quantitative measurement. When subjective judgements are required, there may be a certain amount of variability in the evaluation of the results. Sound professional judgement is required in cases where there is subjectivity in the criteria, the questions, or the analysis of the results. Others using this methodology may impose different subjective judgements and reach somewhat different conclusions. This is particularly true in the area of the user interface, where the preferences of users vary significantly.

There are some strong advocates of a methodology that removes much of the judgement and subjectivity from the evaluation [BAILEY, LINDQUIST]. Criteria such as "ease of use," or "ease of learning" would be given operational definitions in terms of how long it takes a group of people, with specified training and experience, to do a certain task. The results of this controlled experi-

ment would then be used as the basis for evaluation. Using suitably large numbers of subjects and the appropriate statistical techniques, this approach is deemed by its proponents to be very effective, despite high variability (one or more orders of magnitude) of the abilities of the subjects.

We question the controlled experiment approach for evaluating environments, because of its high cost, and because of its underlying assumption that one can make objective operational definitions for concepts that are inherently subjective. We believe that informed subjective judgement based on systematic use is more valuable than batteries of controlled experiments that may give only the appearance of objectivity.

2.2.4. Emphasizing Primary Functionality

No environment is likely to perform equally well on all experiments, and it is not the purpose of the experiments to set absolute standards for environments. Different environments are built to satisfy different requirements. For example, portability may be emphasized at the expense of performance. Our intent is to test a broad set of necessary functionality and to report on how well an environment implements that functionality. Conclusions based on the experiments may vary according to the weight given to various criteria. The methodology must provide a standardized benchmark of core functionality against which environments may be measured. This core functionality must include activities associated with programming-in-the-small and, especially, activities associated with programming-in-the-large. Requirements for environments can be influenced then both by what is desirable and by what is achievable.

2.2.5. Evolutionary

Just as it is important in software engineering to iterate through several stages of the life cycle, it is important to iterate through several phases of an evaluation. The methodology should recognize that the evaluator will not be able to write down all the criteria until some experiments have been executed. Similarly, it is difficult to specify all desirable functionality until experience has been gained with several environments. For these reasons, the approach should allow iteration and refinement of the various artifacts of the methodology as the evaluation proceeds. By the time several experiment suites have been applied to a number of environments, the evaluation technology will be much stronger than it would have been if each phase were completed entirely before moving to the next phase. This criterion has the usual effect of uncovering problems as early as possible and minimizing the cost of recovery.

2.2.6. Extensible

The methodology should be extensible so that additional user activities and accompanying experiments can be added easily. This enables the evaluation to become more comprehensive or better tailored to the current needs of the evaluator. It also permits future expansion of the evaluation into new areas as the capabilities of environments are expanded. Because there is a

shortage of tools and aids for evaluating software development environments, it is important to concentrate first on the most important user functions. Optional or specialized functionality that may be important for certain communities can be added when evaluations are undertaken on behalf of those communities.

2.3. Methodology

There are six discrete phases of the methodology. The overall approach is to determine the key software life-cycle activities rather than use any pre-existing tool taxonomy. These activities form the basis for experiments that are designed to extract evaluative information. The first three phases are environment independent and are performed only once for all environments, while the remaining three are environment dependent and are performed once for each environment evaluated. Several of the six phases contain more than one step.

The six phases of the methodology are as follows:

Phase 1: Identify and Classify Software Development Activities
Phase 2: Establish Evaluative Criteria
Phase 3: Develop Generic Experiments
Phase 4: Develop Environment Specific Experiments
Phase 5: Execute Environment Specific Experiments
Phase 6: Analyze Results

Figure PRODUCTS shows the products of the methodology and their relationships. It is not a flowchart of the process, but rather it shows how the artifacts of the process depend on one another. The figure is divided into two parts: The products on the left are gathered or produced once and apply to the evaluation of environments in general; the products on the right are produced for each environment under evaluation and in some cases for each experiment. The products on the left of the figure can be used by anyone wishing to evaluate a new environment, while the products on the right are useful in evaluating particular environments.

It should be noted in the following discussion that the methodology is amenable to parallel activities on several experiments. For example, development of environment specific experiments for one set of activities can take place in parallel with development of generic experiments for another set of activities. This is in keeping with the rapid prototyping philosophy of the approach. It is important to recognize that the methodology applies to the evaluation of environments as a whole as well as to its individual parts or tools. The phases are described as a set of experiments, but they also apply to each experiment.

Figure 2-1: Products of the Evaluation Methodology

2.3.1. Identify and Classify Activities

The first phase of the methodology consists of three steps:

Step 1: Identify the activity classes that will be exercised by the set of generic experiments.

Step 2: Refine each activity class of the previous step into a list of specific activities.

Step 3: Classify as primary and secondary the refined list of activities. An environment's level of support for primary activities will be carefully scrutinized.

This phase of the methodology answers the question "What do the software developers, system administrators, project managers, and others involved in the software development process have to do to accomplish their jobs?" It attempts to avoid the question of "How are they going to accomplish their tasks?" This phase is therefore based on the underlying activities of software engineering rather than on the tools of the environment. The second step in this phase is simply a recognition that stepwise refinement makes sense in defining the activities. The third step is a recognition that often it will be unnecessary or too costly to carry out a comprehensive evaluation and that the evaluation should concentrate on those activities that are considered most important.

There are a number of sources, in addition to one's own experience, for defining and classifying activities that can be supported by an environment. Some are more oriented toward tools than activities, but all are helpful in defining what can be automated. Those that have been used in defining the activities in our study of Ada Programming Support Environments include the SEE taxonomy [SEE], the NBS taxonomy [NBS], the Software Development Standard [SDS], the Evaluation and Validation Team's Classification Schema [EVSchema], and the AdaEurope study [LYONS].

There is no requirement for this phase to result in an exhaustive set of activities. An evaluator may choose to evaluate only a subset of the entire set of software development activities. The product of this phase is labeled "activities" in Figure 2-1.

2.3.2. Establish Evaluative Criteria

This phase of the methodology consists of two steps:

Step 1: Establish the criteria by which each activity may be judged using the broad categories of functionality, performance, user interface, and system interface.

Step 2: Develop a set of questions spanning the criteria, highlighting specific areas where quantitative and qualitative assessments will be made when performing each experiment step.

The evaluative criteria must be defined at two levels. First, the criteria must be defined at a high level to apply to the environment as a whole. Second, the criteria for particular activities must be established. The criteria are grouped into four major areas: functionality (the broad capabilities available in the environment and the narrower capabilities of individual tools), performance, the user interface, and the system interface. These are not mutually exclusive categories, but they are reasonably complete and distinct.

A reasonably complete set of criteria categories has been constructed by the Evaluation and Validation Team of the Ada Joint Program Office [EVReqts]. The criteria categories must be quantified by measurements and questions related to these categories. It is generally better to avoid absolute standards for criteria, allowing the marketplace of products to provide a set of achievable standards. As more environments are evaluated using a particular set of generic experiments, what is desirable and what is achievable will become more clear.

2.3.3. Develop Generic Experiments

Step 1: Develop a logically continuous sequence of environment independent (generic) experimental steps instrumented with data collection operations. The generic experiments collectively should be designed to involve a large subset of the individual activities enumerated in Step 2 of Phase 1.

Step 2: Identify those evaluative questions from Step 2 of Phase 2 that apply to each experimental step.

Based on the user activities, the evaluative criteria and questions, and what is currently available in exemplary systems, a set of generic experiments is produced. First, a series of general areas of experimentation is defined. In each of these areas, one or more experiments is developed to test the broad criteria areas. The experiments are generic in the sense that they do not use specific tools in specific environments, but refer to generic tasks that must be performed in sequence. These experiments must be constructed with a great deal of care. They must be detailed enough to allow the implementor to translate them into specific experiments, but general enough not to imply a specific set of tools. The specific questions to be asked and the specific measurements to be made at each step of the experiment must be inserted at each step. The products of this phase are the "generic experiments" themselves as well as the "functionality checklists" as shown in Figure 2-1.

2.3.4. Develop Environment Specific Experiment

Step 1: Instantiate the generic experiment in the host environment. This involves transforming each generic step into a sequence of environment specific actions and assessing the level of support the environment offers for each activity. A functionality checklist will be completed as part of this step.

This phase must be performed by an expert in the specific environment so that the translation will take best advantage of the available tools. The translation process results in a "script" in the command language of the subject environment so that the execution of the script is a mechanical process. There may be some subjectivity in the translation process, but the idea is that the expert will find the most effective way to accomplish the objectives of the generic experiment.

The results of the translation process include the determination of the difficulty of the translation (how many actions must the user perform to accomplish a task?) and answers to questions about

functionality (what can and cannot be done easily?). It is not assumed that all of the generic experiment can be performed easily in the subject environment. Those areas in which the environment lacks functionality or where the functionality is less than what is desirable will be noted. The products shown in Figure 2-1 are the "environment specific experiments" and the "translation analysis."

2.3.5. Execute Environment Specific Experiment

This phase represents the execution of the environment specific experiment on the subject environment.

Step 1: Perform the experiment in the host environment, creating an experimental transcript that includes command responses and measurement data. Answer the appropriate questions at each step and make the measurements and observations indicated within each experimental step.

From this step comes a "transcript" of the execution of the experiment and the measurements and answers to questions that were formulated in phase 3. This represents the "raw data" from the experiment without any interpretation.

2.3.6. Analyze Results

The final phase requires that we analyze the raw data. Both the results of the translation and the results of the experiment are inputs to this phase.

Step 1: Examine the experimental transcript, the response to each question and the functionality checklist and draw conclusions about each activity class for each major criterion category.

In this phase, there will be qualitative statements made about the experiment as it relates to the specific environment. The analyses for the individual experiments will form the basis for the "environment analysis" as shown in Figure 2-1.

After the methodology has been applied to several environments, it will be possible to compare the results of the same experiments across different environments. Recommendations for selecting existing environments or for constructing new environments can therefore be derived from a common pool of experiments using the same activity and criteria base.

2.4. Software Development Activities

The following taxonomy of APSE functions was derived from a number of sources and represents the areas to be covered in this study. The four primary areas are (1) system management activities, (2) project management activities, (3) technical development activities, and (4) configuration management activities. While there is not a one to one relationship between these primary functional areas and the experiment categories, the experiments tend to test the test the functionality of one area more than the others.

1. System Management Activities

    a. Environment Installation
        - Load environment software from release media
        - Integrate with existing operating environment
        - Perform acceptance tests

    b. User Account Management
        - Create/Delete user accounts
        - Copy user accounts
        - Rename user accounts
        - Create/Delete user account groups
        - Disable User Accounts
        - Add user account to group
        - Remove user account from group
        - Establish user account characteristics
        - Modify user account characteristics
        - Establish default account characteristics
        - Modify default account characteristcs
        - Display user account characteristics
        - Display default account characteristics
        - Create initial working directories
        - Establish default login/logout macros
        - Verify creation of user accounts

    c. System Resource Management
        - Collection of accounting information
        - System workload monitoring
        - Modification of system configuration

    d. Environment Maintenance
        - Software bug reports
        - Software updates
        - Disk Back-ups

    e. System Statistics Collection
        - Accounting information
        - System performance statistics

2. Project Management Activities

    a. Administrative Management [Project planning, estimating, reporting and communication]
        - Create work breakdown structure
        - Allocate resources to task

- Identify when resources are available
- Create high level project schedule
    - Establish milestones
    - Perform critical path analysis (PERT)
    - Create reports (PERT/GANTT)
- Document preparation
    - Text formatting
    - Business graphics
    - Spreadsheet capability
- Intra-project communication
    - Mail
    - Bulletin boards

b. System Management [Project database configuration, access and control]

- Directory structure
- Program library sharing
- Default access control
- Security precautions
- Very large system activities
- Distributed development and access
    - File access (transparent or file transfer program)
    - Remote login
- Subsystems
    - Interfaces, specifications and hiding above the package level

c. Technical/Supervisory Management [Project monitoring, execution, and control]

- Automatic notification
    - Set user defined dependency monitors (semantic dependencies)
    - Alert changing user when dependency triggered
    - Alert dependent user when dependency triggered
- Project activity management
    - Enter activity list for individual or group
    - Display activity list for individual or group
    - Update activity list for individual or group
    - Name, date, and completion time for each task
    - Automatic iteration between library unit changes and activity list
    - Graphical task editor
    - Check off for completed tasks
- Analyze progress against schedule
- Analyse resource utilization against schedule
- Perform "what if" analysis
- Modify project schedule

d. Quality Management [Project quality assurance]

- Project review - structured walkthroughs
- Documentation review
- Code audits
- Dependency tracking
    - Requirements

- Design
- Code
- Documentation
- Help files

- History management
  - Requires reason for changes - plus time, date, node id, programmer name
  - Retrieval of changes line-by-line
  - Notification of changes

3. Technical Development Activities

   a. Requirements analysis

   b. Requirements definition

   c. System specification

   d. Preliminary design

   e. Implementation
      - Detailed design
      - Code development and translation
      - Unit testing
      - Debugging

   f. Product integration

   g. Product testing

   h. Product maintenance

   i. Documentation
      - Requirements specifications
      - Design document
      - Test plans and procedures
      - User's manual
      - Installation guide
      - Maintenance manual

4. Configuration Management Activities

   a. Version control

   b. Configuration control

   c. Product release

2.4.1. Excluded and Deferred Issues

2.4.1.1. Portability/CAIS

The initial requirement for portability of tools from one host machine to another was contained in the Stoneman document. Since that time a great deal of work has been done in defining a Common APSE Interface Set which represents the core of functionality upon which an environment could be built. In January 1985 the DoD issued a proposed military standard CAIS [CAIS]. The stated purpose of this document was to encourage prototype implementations and experimentation with the CAIS. Over the last year there have been numerous suggestions for

changes in the CAIS, and the CAIS Working Group has been addressing issues as they arise. As of the end of 1985 there were three prototype implementations being developed at Arizona State, Mitre, and Gould.

There has been some success in porting environments that are extensions of the Unix operating system. However, it is felt that the current state of the practice in portability is not sufficiently developed to consider it in this evaluation. The issues of rehosting tools are many and complex and involve the internal structure of the environment. Therefore, it is beyond the scope of this study to consider portability.

2.4.1.2. Run-time Systems

One of the most important results of a software engineering environment is the final product: the program running on a target system. The performance of the run-time system in a Mission Critical Computer Resource (MCCR) is our important evaluation criterion. Unfortunately, we are only beginning to see environments which produce object code for a target other than the development machine. There are a whole series of issues that are raised (such as the ability to dynamically configure the run-time system based on the needs of the object program). In order to evaluate these systems, it is necessary to have access to the target machines. Because this area is so poorly developed at present, this topic of evaluation will be left as a deferred topic. It is worth noting, however, that there is an Ada Run Time Environment Working Group (ARTEWG) under SigAda which was formed in early 1985 to address these issues. The group consists of 20-30 members and meets quarterly.

2.4.1.3. Commodity Software

Commodity software refers to those programs which are peripheral to the software engineering process, but important to communication and management functions. Examples of these programs include mail systems, document preparation systems, and spreadsheet programs. For any but the smallest of projects, commodity software is critical. This study considered commodity software to the extent to which it can be integrated with the APSE. It did not consider the functionality and performance of the individual programs. One of the criteria for the APSE will be the ability to use the software of the underlying operating system. This will be tested and evaluated.

2.4.1.4. Validation Issues

Validation deals with the correctness of a program and requires rigorous testing by the vendor as well as the accepting organization. This study has not consciously attempted to uncover bugs in the environment. When they were found inadvertently, the report has noted them. This study has tried to exercise the environment fully, but the emphasis has been on the usability of the environment rather than the correctness.

2.5. Criteria

The four broad categories of criteria with their relationships to each other, the user, and the underlying system are shown in Figure CRITERIA. The functionality criteria category dominates and controls the three other criteria categories of user interface, performance, and system interface. Without functionality, there can be no other criteria. Performance relates to the user in the form of responsiveness and to the system in the form of time and space efficiency.

The criteria described here are still general in that they provide only guidelines for tools and environments in the aggregate. The generic experiments will provide more detailed criteria that apply explicitly to each activity. It should be emphasized that the criteria described in this section apply in some cases to the components of an environment, in some cases to the environment as a whole, and in yet other cases to both.

Figure 2-2: Criteria Categories

2.5.1. Functionality

The functionality criteria will be described for the most part by checklists to be provided in the individual experiments. The functionality of a programming support environment (as opposed to more narrowly defined programming environments) should span the entire life cycle. That is, there should be tools available for system management activities, project management activities, technical development activities, and configuration management and version control.

The functionality of each individual tool should be complete in the major functional areas that are commonly available in other similar tools. The spanning set of functions for tools can be derived from a number of exemplary environments in common use today. In that way we avoid the criticism that we are asking for what is impossible or for what will be available only in the future after considerable work.

There is an overlap between functionality criteria (what is available in an environment) and the user interface criteria (what is available to help the user perform tasks effectively). In some cases, there may be duplication of this information in the taxonomy of criteria presented.

The following activities are considered essential in any environment used for Ada program development and should be provided with the environment from the beginning.

Translation: The language translator is central to any Ada environment. However, since this function is being evaluated extensively by others, the SEI study is applying existing tests. It will evaluate broad performance characteristics, diagnostics, documentation, and the user interface.

Command Interpretation: The Stoneman guidelines [STONEMAN] allow a wide latitude for implementors of the generalized user interface. The environment may use a command language or a graphically driven menu system. Evaluating this function involves specifying a number of generic tasks to be accomplished and then observing what is required by the command interpreter to carry out those tasks.

Program and Data Management: Each environment must have a database system that provides a mechanism for organizing and retrieving a variety of program and data objects. There must be operations for creating, deleting, moving, and copying these objects. The process of navigating within a database and the process of browsing through Ada programs are functions included in this category.

Access Control: Users of the environment have different access rights to the objects in the database. For example, once a program has been released as a product, all rights to modify a program object may cease to exist. Experimental versions of a program object may be modified by selected programmers but not others.

Version Control: In medium and large projects, program modules undergo many improvements and modifications. Versions coexist with minor variations in performance and memory utilization. Official releases coexist with experimental versions that have restricted access. A system development environment must provide the necessary bookkeeping facilities for version maintenance.

Configuration Management: In medium and large projects, the interdependencies between independently constructed modules must be recorded and tracked by the environment.

Project Management: In large projects, it becomes necessary to add tools for management of people, tasks and software modules.

Linking/Loading: Independently developed modules must be linked together and loaded in order to be executed.

Editing: Editing of source and documentation objects is an essential task, that has received extensive study. The SEI evaluation treats the editor similarly to the translator in that it is not studied in detail. Its general characteristics as well as its interface with the system and the user are important criteria.

Debugging: Debugging is a critical part of the program development process used to isolate and correct execution-time errors. The debugger allows the development process to be more interactive and provides significant productivity gains in the testing and integration phases of the life cycle.

2.5.2. Performance

Performance, like functionality, can be considered a criterion for each tool and for the aggregate. Each tool should be space and time efficient. The environment as a whole must allow the user to be efficient by executing trivial requests quickly. Raw compilation speeds are less important than the strategy of recompilation when changes occur. The following represent several of the global criteria that relate directly to performance.

Responsiveness: How quickly does the system respond to various user requests? Certain functions (such as a long compilation) should take longer than simple requests (such as changing the current working directory). There are no fixed standards for responsiveness, but users who are required to wait for more than a second or two for simple requests generally find systems unusable.

Efficiency: How efficiently do the tools of the environment use the resources of the underlying abstract machine? This will influence how much resource is required or conversely, how many users can be supported on a given resource. Both time and space requirements of the software are considered part of this criterion.

Avoidance of Recompilation: Some changes to modules (e.g., addition of comments) do not strictly require recompilation. Changes to the body of an Ada unit do not strictly require the recompilation of dependent units. Many environments for a variety of programming languages currently support incremental compilation and hence avoid recompilation in many cases. This is certainly a desirable performance improvement and can be more important than raw compilation speed. The benefits can be enormous in the large systems for which Ada was developed. The costs are more complex compilers and the retention of more information during program development.

2.5.3. User Interface

The evaluation of a user interface must take into account the characteristics of the tool as well as the characteristics of the user employing the tool. At least three types of users may be expected to become familiar with Ada environments. The novice or first time user needs a great deal of assistance in the form of documentation, tutorials, help facilities, and a command structure that is easy to understand. A novice will be content to use a small subset of the full power of the system. A casual or occasional user can be expected to know many of the functions but will need help in remembering how to use them. The proficient or frequent user can be expected to know most of the functionality of the system and will be impatient with features that require long sequences of repetitious or redundant input.

Among the points to be considered in an evaluation of the user interface are the following:

Learnable/usable: Ideally, a system should be easy to learn and easy to use; however, there may be a tradeoff between the two criteria. The emphasis in the SEI study is placed on the "power user," i.e., the reasonably experienced user who prefers as little system intervention and interference as possible. Thus, we consider ease of learning to be important, but secondary to ease of use.

Interactive: Users are more productive in an interactive mode than in a batch mode. Interactivity has to do with the responsiveness of a system in providing feedback to a user. In general, users make fewer errors if informed about errors early.

Consistent/uniform: The environment is easier to use if it is consistent within and across tools. For example, the mechanism for exiting a tool should be the same whenever this is reasonable. Command languages or other tool invocation methods should be consistent in naming or pointing conventions.

Not unnecessarily complex: Certain complex functions require complex mechanisms to implement them. Simplicity of structure is a desirable goal, but not always achievable. Thus, the rule of thumb is to make a mechanism no more complex than it has to be. If a given capability or feature is seldom used, it should not affect those who do not use it.

Predictable: The environment should not make surprising responses or cause unexpected results, except perhaps to warn the user of catastrophic types of errors. In an environment that appears to be predictable, the user will be comfortable and productive. If forced to interpret the meaning of unexpected responses, the user will be confused and unproductive.

High bandwidth communication: The user should be able to communicate a maximum amount of information with minimum effort. On the input side, the user should be able to tell the system what to do using as few keystrokes or as little physical motion as possible. On the output side, the system should be able to use the minimum amount of space on the screen. This criteria is highly machine dependent and relates to the system interface criteria of using all the capability of the underlying machine. If a multiple window capability can provide higher bandwidth communication by providing information about several topics simultaneously, and if the underlying system provides these capabilities, then the environment should take advantage of them.

Helpfulness: The environment should include the following characteristics to help the user when necessary.

- **On-line assistance**: The environment should provide on-line assistance rather than require the user to refer to paper documentation for all questions and problems. This feature should permit the user to perform keyword searches when the user knows what to look for and hierarchical searches when the user does not know precisely what to look for. Both the on-line and paper documentation should have appropriate indices and cross references.
- **Command completion**: The environment should help the user by providing command completion and prompting for command parameters. For keywords and contexts in which only one entry is possible, the user should have the option of allowing the environment to provide the remainder of the entry. This is similar to allowing the user to provide an abbreviation file, but it takes no intervention on the part of the user and is consistent across all users in the environment.
- **Avoid taxing user's memory**: Mnemonic names should be chosen in environments that use a command language. When there are a number of options to a command, there should be on-line help to explain those options. The syntax of the command language should be consistent throughout the environment to avoid the need to remember more than is necessary.
- **Effective documentation**: Both paper and on-line documentation should be complete, clear, and easy to use. It should be easy to find the answer to a specific question. For vague questions, the user ought to be able to navigate through either form of documentation to find an answer.
- **Tutorials**: For new users, there should be both paper and on-line tutorials that step through both the fundamental and the more advanced features of the environment. These tutorials should be comprehensive and based on sound educational principles. Advanced tutorial systems could track the user, tailoring the lessons for the user's current state of knowledge.

Customization: Users should be able to customize the environment according to their preferences.

- **Key bindings**: The user (or at least the system administrator) should be able to change the meanings of keys to customize the keyboard. This is particularly important for control keys and function keys, but must be used judiciously because it may cause inconsistency of the generalized user interface.
- **Command procedures**: It is generally accepted practice to permit commonly used

sequences of commands to be placed in a file so that they can be referenced with a single name. Various levels of capabilities exist for adding parameters to command procedures and allowing various kinds of program control within the command procedures. Special command procedures can be invoked automatically upon login to customize the environment and upon logout to provide cleanup operations.

Error handling: Errors should be handled in a way that improves the user interaction.

- **Tolerance**: Minor errors that are correctable by the system ought to be corrected by the system. One of the most annoying things a system can do in response to an erroneous command is tell you that you made an error, tell you what you meant to do, and then tell you to reenter the command. The system need not guess what you meant to do, but neither should it force you to take three steps backward if it is not necessary. If an error does not require stopping forward progress (e.g. a warning) then it should not stop forward progress.
- **Location and identification**: The system should have a fine granularity in locating and identifying errors. The environment should provide precise information on the source of the error and avoid cascading error messages if possible.
- **Early detection**: Failure to detect errors early can cause a cascade of subsequent errors and a significant amount of backtracking on the part of the user. For example, if the editor has information about the syntax of the Ada language, syntax errors can be detected and corrected as the program is entered rather than when it is translated. This may also, as a side effect, prevent semantic errors.
- **On-line messages**: Error messages produced in the output listings of language translators seldom provide enough information for diagnosis. An on-line error message capability can provide an extended explanation of the error and its probable causes.
- **"Undo" ability**: When a user makes a mistake, such as deleting a portion of a file by accident, it is very useful to have a capability to "undo" the deletion. This capability can apply to the last executed command, the last **n** commands, or back to the beginning of the session. Obviously, there is a tradeoff here between performance and desired functionality.

Support for multiple views: The environment should enable the user to view objects at different levels of detail.

- **Formatting**: The user of an Ada development environment ought not to have to worry about how Ada program appears on a page. The mechanical process of indenting a program for readability and inserting a pleasing amount of white space in the appropriate spaces should be left to a tool. Either this function should be provided by the editor, or by a program called a "prettyprinter." In either case, some flexibility should be permitted to account for different tastes and different local standards.
- **Elision**: The user ought to be able to view Ada programs and libraries in a way that suppresses those parts of the program or library that are not currently of interest at the level of abstraction that the user is working. As one example, the user may wish to look at the specification part of a unit or group of units and not be concerned about the implementation details of the bodies.
- **Browsing capability**: When a user is trying to understand the complexities of Ada programs, it is important to be able to browse rapidly through a program library. For

example, when one unit is "withed" into another, it may be necessary for the user to determine the definition of an object that is made visible in the current unit. The user may wish to know all those units that depend on the unit being worked on. For these and other similar functions, a browsing capability is important to an Ada programming environment.

2.5.4. System Interface

Many operating systems provide much of the functionality required by Ada environments. When an Ada environment is built on top of an existing environment, it is important that the interface be a smooth one. If it is not, a duplication of effort and a consequent loss in performance results. For this criteria category, we propose that each duplicated or nearly duplicated feature in the underlying environment be tested to compare the functionality, performance, and user interface of the two features. If the characteristics of the tool in the operating environment are superior to the similar tool in the Ada environment, one should question the wisdom of using the new tool.

The consistency of the command set with the command set of the underlying environment and the compatibility of the tools with those of the underlying environment may be especially important during a start-up period in which software professionals are being retrained. All people resist change; but if it is perceived that the change does not bring a significant improvement, then it will be extremely difficult to impose the change.

If the Ada environment does not provide a significant improvement over the underlying system for the basic functionality, it will be necessary to evaluate the possibility of using an Ada compiler with the underlying environment at least as an interim measure while environments are improved to the extent that Ada compilers have been improved over the last several years.

Open (easy to add tools): The openness of an environment refers to the ability to add new tools to that environment. When an environment is highly integrated from the start, there may be specialized interfaces between tools that make it difficult to add new tools from outside the environment. When an environment is an extension to an existing environment, the question is whether all the tools of the host environment are available to the Ada environment.

Integrated (tools work well together): The extent to which an environment is integrated refers to how well the tools work together. Integrated environments tend to be more uniform and consistent. If the output from one tool must be altered in format before it can be used as input to another tool, then there is evidence that there is a lack of integration. Furthermore, we can distinguish between tools provided with the environment which ought to be well integrated because they should be designed to work well together, and tools added later. If tools are to be added later, then the environment should provide the definition of the interfaces to the base environment.

Able to use operating system tools/facilities: When an environment is built on top of an existing operating system, it should be possible to use those operating system tools and facilities when it is appropriate to do so. Conversely, the environment should be able to prevent the use of tools and facilities that are inappropriate in an Ada environment.

Able to use hardware effectively: The Ada environment should make appropriate use of the underlying hardware. Input devices such as mice should be used when appropriate to choose from a menu. Output devices such as bit mapped displays should use windowing. Memory allocation schemes of the underlying machine, such as caching or virtual memory, should be used to maximum advantage and not be subverted or duplicated.

Support distributed development (when appropriate): There is currently a trend away from the centralized computing systems of the 1960s and 1970s and toward distributed systems of personal workstations connected by local-area networks or wide-area networks. When environments for large system development are hosted on distributed systems, the distribution of function and data must be as transparent as possible. In particular, it should be possible for a project group to be working on a common software library without being concerned about where the library resides in the network.

2.6. Environments Evaluated

The contract called for the evaluation of the Ada Language System (ALS), the Ada Integrated Environment (AIE), and "other environments." The ALS was installed at the SEI since November, 1985. It was upgraded from version 2 to version 3 in May, 1986. The AIE was downgraded from an APSE to a single component Ada compiler and was renamed the VAX Ada. As of mid-1986, it still had not been delivered to the Air Force. Also installed were DEC's VAX Ada compiler with five tools called VAXSet running under the VMS operating system and the Verdix Ada Development System (VADS) running under the Ultrix operating system. The Alsys compiler running on the Apollo Domain operating system was evaluated on one group of experiments (Project Management). Five groups of experiments were conducted on the other three environments.

2.6.1. Hardware Considerations

The performance of an environment will depend critically on the hardware configuration on which it runs. In particular the processor, memory size, and the disk configuration are crucial to good performance. In this study we have run the environments on configurations that meet or exceed the vendor's initial recommendations. It would at first appear to be desirable for all hardware configurations to be exactly the same, but this is not necessarily the case. It is possible that one system uses more resources in order to make significant gains in performance or to improve the

functionality. Thus in all cases we have attempted to meet or significantly exceed vendor recommendations for resource requirements. Three of the environments tested operate on DEC VAX hardware and the MicroVAX II was selected to test all three. This machine offers the advantage of being a single user system to avoid problems of interactions with other users, while still being approximately 80 percent as powerful as the older multi-user VAX 11/780. The Alsys Ada testing was conducted on an Apollo DN460 node.

The five experiment groups that were run on MicroVAX II hardware were actually run on three distinct machines with the three different environments installed. In each case the hardware configurations met or exceeded the vendor's recommendations. In the case of the SofTech Ada Language System, version 3.0, the configuration included 9 megabytes of main memory and disk space consisting of three RD53 disk drives (213 megabytes). The environment supplied by Digital Equipment Corporation consisted of the VAX Ada, version 1.2 and a set of five tools called VAXSet, all of which run on the VMS operating system. The configuration for VMS/VAXSet included 6 megabytes of main memory and 102 megabytes of disk space. The Verdix environment is called the Verdix Ada Development System (VADS, version 5.1) and runs on top of the DEC supplied version of Unix called Ultrix (version 1.2). The tests were run on a system configured with six megabytes of memory and 202 megabytes of disk space.

The final experiment group ran on an Apollo computer with the DOMAIN/IX Unix equivalent operating system. The Domain Software Engineering Environment (DSEE) and the Alsys Ada Compiler completed the software configuration. The hardware consisted of an Apollo DN460 workstation with 4 megabytes of main memory and 171 megabytes of disk space. The versions tested were version 2.0 of DSEE and version 1.0 of the Alsys Ada compiler.

2.7. Experiment Groups

Six experiment groups have been defined. Each experiment group consists of a set of tests which exercise various aspects of an environment. The first five experiment groups are based on the methodology defined in this chapter. The last experiment group is a suite of compiler tests assembled by IDA. The succeeding chapters describing these experiments identify and classify the programming activities being tested, specify the evaluative criteria and associated questions, and define the specific steps required to carry out the experiment. While the IDA test suite does not conform to the methodology defined in the report, it is, nevertheless, an integral part of the total evaluation.

2.7.1. Configuration Management

The configuration management experiment group exercises the configuration management and version control capabilities of the APSE. In medium and large systems projects, modules undergo many improvements and modifications. Versions exist with minor variations. Official releases coexist with experimental versions that have restricted access. This experiment defines a set of modules which are derived from a real system, the Texas Instruments APSE Interactive Monitor (AIM) project [TIAIM, TIAIM2], and strips them of all their code other than the intermodule dependencies. The experiment then simulates the system integration and testing phase of the life cycle by having three separate development lines of descent from a single baseline system. This process provides information about the version control capabilities (space requirements, transparency, performance) as well as the configuration management capabilities (translation rules, specification of modules from releases, history collection, performance).

2.7.2. System Management

The system management experiment group exercises the environment from the perspective of the system manager. The activities of concern here are the installation of the APSE on a raw machine or operating system, the management of user accounts, maintenance of the environment, and system resource management. Installation is concerned with loading the environment from the release media, integrating it with the existing operating system, and performing acceptance testing. User account management is concerned with establishing, modifying, and deleting accounts or groups of accounts, creating access control, establishing default login/logout macros, and displaying account information. Maintaining the environment has to do with updating the environment software, performing backups, and archiving. System resource management involves the collection and recording of various accounting and system performance data, monitoring the system workload, and reconfiguring the operating environment for optimal performance.

2.7.3. Design and Code Development

The design and code development experiment group exercises the activities normally associated with small projects, namely the design, creation, modification, and testing of a single unit or module. In particular, this involves the entry of an Ada program unit and evaluating that unit using a test program. The hypothetical setting is one in which a small team is tasked to create vector and matrix handling modules. This experiment provides a primary vehicle for the testing of the user interface of the environment. The ease of use, consistency, helpfulness, and error handling capability are evaluated here. The system's space and time efficiency are recorded, but this experiment is not the primary source of information about the performance of the compiler.

2.7.4. Unit Testing and Debugging

The unit testing and debugging experiment group exercises the environment from the perspective of the unit tester. It is designed to be a sequel to the design and code development experiment. A small set of Ada units was seeded with errors and debugged using the facilities available in the environment. This experiment tested the capabilities and functionality of the debugger and examined the user interface of the debugger. This experiment also looked at the dynamic and static analysis tools as well as the tools available for test creation and test management.

2.7.5. Project Management

This experiment group is meant to investigate the facilities available for very large programming projects, particularly as they relate to management functions. Among the functions that should be automated is the traceability of a number of data objects back to their requirements. There should be databases which show relationships between programs, requirements, specifications, documentation, and other artifacts of the programming process. There ought to be mechanisms to automate the maintenance phase of the software so that all change requests and changes are documented and cross referenced. Work assignments to project members should be recorded, tracked, and closed when completed. Furthermore, there should be mechanisms which allow projects to be broken down into subsystems in order to minimize the impact of changes to one subsystem on another.

The project management experiment group took into consideration only a small portion of the total functionality which should be attributed to project management. Namely, the purpose of the experiment was to explore the activities surrounding the building and maintaining of the project database which is but one aspect of the technical management activities of project management. Chapter 7 describes the whole realm of project management activities, but defines an experiment for key technical management activities only. The experiment was run only on Apollo hardware using the Alsys compiler.

2.7.6. IDA Prototype Ada Compiler Evaluation Capability

This experiment is different from the previous three in that it was externally generated and tests only a single component of the environment. Instead of creating a new set of Ada programs to test the performance of the Ada compiler, we used the Institute for Defense Analysis (IDA) prototype test suite. This test suite is derived from a number of sources and is designed to give a fairly rigorous test of the performance of the various features of the language. The suite is designed to be automated and instrumented to provide a report for each test and the entire suite. Numerous problems were encountered in applying the test suite to the three environments. These problems as well as the many useful results are documented in Chapter 8.

2.8. Summary of the Remainder of the Report

This report contains six additional chapters, one for each of six experiment groups. Chapter 3 gives the results of the System Management experiment group. Chapter 4 gives the results of the Configuration Mangement experiment group. Chapter 5 presents the results of the Design and Code Development experiment group. Chapter 6 gives the results of the Test and Debug experiment group. Chapter 7 presents the Project Management experiment group. Finally, Chapter 8 provides the results of the IDA ACEC prototype test suite.

Glossary

ACEC	Ada Compiler Evaluation Capability
ACVC	Ada Compiler Validation Capability
AIE	Ada Integrated Environment
AIRMICS	Army Institute for Research in Management Informations and Computer Science
AJPO	Ada Joint Program Office
ALS	Army/Navy Ada Language System
APSE	Ada Programming Support Environment
APSEWG	APSE Working Group
ARTEWG	Ada Run Time Environment Working Group
CAIS	Common APSE Interface Set
COORDWG	Coordination Working Group
CVC	CAIS Validation Capability
DoD	Department of Defense
E&V	Evaluation & Validation
KAPSE	Kernal Ada Programming Support Environment
MAPSE	Minimal Ada Programming Support Environment
MCCR	Mission Critical Computer Resource
MIL	Military
REQWG	Requirements Working Group
SDC	System Development Corp.
SDS	Software Development Standard
SEE	Software Engineering Environment
SEI	Software Engineering Institute
SEVWG	Standards Evaluation and Validation Working Group
SigAda	ACM Special Interest Group on Ada
STARS	Software Technology for Adaptable Reliable Systems
STD	Standard
WIS	WWMCCS Information System
WWMCCS	World Wide Military Command and Control System

References

[Ardis 85] Ardis, M.A., Gehling, J., Gill, T.D., Glushko, R. and Vishniac, E.
An Assessment of Eight Programming Environments.
Technical Report TR-85-16, Wang Institute of Graduate Studies, August, 1985.

[Bailey 85] Bailey, E.K. and Kramer, J.F.
A Framework for Evaluating the Usability of Programming Support Environments.
Technical Report, Computer and Software Engineering Division, Institute for Defense Analysis, 1985.

[Basili 84] Basili, V.R., Panlilio-Yap, N.M., Ramsey, C.L., Shih, C. and Katz, E.E.
A Quantitative Analysis of a Software Development in Ada.
Technical Report TR-1403, ADA165337, University of Maryland, May, 1984.

[Brinker 85a] Brinker, A.
A Critique of the DEC Ada Compilation System (ACS).
NASA/GSFC Code 522.1, NASA, December, 1985.

[Brinker 85b] Brinker, E.
An Evaluation of Softech, Inc. Ada Language System, Version 1.5.
NASA/GSFC Code 522.1, NASA, May, 1985.

[Card 80] Card, S.K., Moran, T.P., and Newell, A.
The Keystroke-Level Model for User Performance Time with Interactive Systems.
Communications of the ACM 23(7):396-410, July, 1980.

[E&V Guide 86] E&V Team.
E&V Reference Manual.
Technical Report, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB, February, 1986.

[E&V Ref 86] E&V Team.
E&V Reference Manual.
Technical Report, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB, February, 1986.

[E&V Team 84a] E&V Team.
Evaluation and Validation Plan.
Technical Report, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB, December, 1984.

[E&V Team 84b] E&V Team.
Requirements for Evaluation and Validation of Ada Programming Support Environments, Version 1.0.
Technical Report, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB, October, 1984.

[E&V Team 84c] E&V Team.
APSE Analysis Document.
Technical Report, Air Force Wright Aeronautical Laboratories, September, 1984.

[E&V Team 85] E&V Team.
Technical Coordination Strategy Document, Version 2.0.
Technical Report, Air Force Wright Aeronautical Laboratories, August, 1985.

[Ellison 86] Ellison, R.J.
Technology Identification and Assessment: Distributed Computing Technology.
Technical Report draft, Software Engineering Institute, February, 1986.

[Feiler 85] Feiler, P.H.
Technology Identification and Assessment: User Interface Technology.
Technical Report SEI-85-MR-4, Software Engineering Institute, September, 1985.

[Flashpohler 85] Flashpohler, J.C., Harder, R.M., and Offutt, A.J.
The Software Test and Evaluation Project.
Technical Report GIT-ICS-85/28, School of Information and Computer Science, Georgia Institute of Technology, September, 1985.

[Hook 85] Hook, A.A., Riccardi, G.A., Vilot, M. and Welke, S.
User's Manual for the Prototype Ada Compiler Evaluation Capability (ACEC) Version 1.
IDA Paper P-1879, ADA163272, Institute for Defense Analyses, October, 1985.

[Houghton 82] Houghton, R.C.
A Taxonomy of Tool Features for the Ada Programming Support Environment (APSE).
Technical Report, U.S. Department of Commerce, National Bureau of Standards, December, 1982.

[Kellner 85] Kellner, M.I.
Technology Identification and Assessment: Database support for Software Engineering Environments.
Technical Report SEI-85-MR-6, Software Engineering Institute, November, 1985.

[Lindquist 85] Lindquist, T.E.
Assessing the Usability of Human-Computer Interfaces.
IEEE Software 2(1):74-82, January, 1985.

[Lyons 86] Lyons, T.G.L. and Nissen, J.C.D.
Selecting an Ada Environment.
Cambridge University Press, New York, 1986.

[Milton 83] Milton, D.
Requirements for Ada Programming Support Environments.
January, 1983.
Computer Sciences Corporation.

[Nestor 86] Nestor, J.R.
Technology Identification and Assessment: Programming Environment Technology.
Technical Report, Software Engineering Institute, February, 1986.

[Newcomer 85] Newcomer, J.M.
Technology Identification and Assessment: Tool Interface Technology.
Technical Report SEI-85-MR-5, Software Engineering Institute, October, 1985.

[Reedy 85] Reedy, Ann.
Stoneman Analysis.
September, 1985.
KIT/KITIA Working Paper, 6/5/84.

[Roberts 83] Roberts, T.L. and Moran, T.P.
The Evaluation of Text Editors: Methodology and Empirical Results.
Communications of the ACM 26(4):265-283, April, 1983.

[SFW 86] Barbacci, Mario R.
Software Factory Workshop.
1986.

[STARS 85] STARS Joint Service Team.
STARS Software Engineering Environment (SEE) Operational Concept Document (OCD).
Proposed Version 001.0, Department of Defense, October, 1985.

[TASC 85] The Analytical Sciences Corporation.
E&V Classification Schema Report, Draft Version 1.0.
Technical Report TR-5234-2, TASC, November, 1985.

[Texas Instruments 85a]
Texas Instruments.
APSE Interactive Monitor -- Final Report on Interface Analysis and Software Engineering Techniques.
Naval Ocean Systems Center Contract No. N66001-82-C-0440, Equipment Group - ACSL, July, 1985.

[Texas Instruments 85b]
Texas Instruments.
APSE Interactive Monitor -- Program Design Specifications.
Naval Ocean Systems Center Contract No. N66001-82-C-0440, Equipment Group - ACSL, July, 1985.

[U.S. Department of Defense 80]
U.S. Department of Defense.
Requirements for Ada Programming Support Environments.
Technical Report ADA100404, DoD, February, 1980.

[U.S. Department of Defense 85a]
U.S. Department of Defense.
Military Standard Common APSE Interface Set (CAIS).
Technical Report MIL-STD-CAIS, DoD, January, 1985.

[U.S. Department of Defense 85b]
U.S. Department of Defense.
Defense System Software Development.
Technical Report DoD-STD-2167, DoD, June, 1985.