ADA EVALUATION PROJECT

THE IMPACT OF THE ADA LANGUAGE ON SOFTWARE TESTING

Prepared for

HEADQUARTERS UNITED STATES AIR FORCE
Assistant Chief of Staff of Systems for Command, Control,
Communications, and Computers
Technology & Security Division

Prepared by
Standard Automated Remote to AUTODIN Host (SARAH) Branch
COMMAND AND CONTROL SYSTEMS OFFICE (CCSO)
Tinker Air Force Base
Oklahoma City, OK 73145 - 6340
Commercial (405) 734-2457
AUTOVON 884 - 2457 / 5152

10 JUNE 1988

Ada Evaluation Report Series by CCSO

Ada Training	March 13, 1986
Design Issues	May 21, 1986
Security	May 23, 1986
Micro Compilers	December 9, 1986
Ada Environments	December 9, 1986
Transportability	March 19, 1987
Runtime Execution	March 27, 1987
Modifiability	April 29, 1987
Testing	June 10, 1988
Module Reuse	June 20, 1988
Project Management	Summer 88
Summary	Fall 88

THIS REPORT IS THE NINTH OF A SERIES WHICH DOCUMENT THE LESSONS LEARNED IN THE USE OF ADA IN A COMMUNICATIONS ENVIRONMENT.

ABSTRACT

Testing on the Standard Automated Remote to AUTODIN Host (SARAH) project was a continuous process. Development testing provided a stabilizing environment for the project. The early design interfaces were done in the implementation code (Ada), developing confidence in both the language and software engineering techniques it supports. The Ada environment had the right tools to allow incremental building. The packaging concept provided a continuous testing platform ensuring functional isolation, verification of data integrity, and an immediate decrease in the opportunity for the introduction of errors. This supported the principles of modularity, high cohesion, and low coupling. Shared packages of tools made the subsystem developers more productive. Generics were used to reduce redundancy in the development of code. Our greatest testing problems occurred because of the tasking environment. The library management system and the vendor developed debug tools proved to be essential in development testing and integration. Testing of software is not affected as significantly by the language used as it is by the engineering applied during the design and development of the system. (KR)

FOREWORD

The ideas expressed here are the thoughts and concerns rendered by the actual staff that participated in the development of the SARAH project from its inception through the final acceptance test. These ideas and concepts are the result of the experiences and reflect the attitudes at the completion of the project. The actual implementation of the SARAH system does not necessarily reflect the best of the ideas presented in this paper. The first project using the Ada language and its associated environment and software engineering techniques, as most experts advise, should not be expected to, and does not, make proper use of all the benefits available. That development staff consisted of both experienced programmer/analysts and inexperienced programmers from varying backgrounds in computer software development.

TABLE OF CONTENTS

1. INTRODUCTION

1.1. THE ADA EVALUATION TASK

This paper is one in a series which seeks to help potential Ada developers gain practical insight into what is required to successfully develop Ada software. With this goal in mind, Air Staff tasked the Command and Control Systems Office (CCSO) to evaluate the Ada language while developing real-time communications software. The task involves writing papers on various aspects of Ada development such as training, Ada design, environments and security issues.

CCSO chose the Standard Automated Remote to AUTODIN (Automatic Digital Network) Host (SARAH) project as the vehicle basis for the Ada evaluation. SARAH is a small to medium size project (approximately 50,000 lines of executable source code) which will function as a standard intelligent terminal for AUTODIN users and will be used to help eliminate punched cards and paper tape as a transmit/receive medium. The development environment for SARAH consists of a number of IBM PC ATs and Zenith Z-248 microcomputers. Source code is developed, compiled, and integrated on these machines. The compiler and symbolic debugger used are from Alsys, Inc. The SARAH software runs on the IBM PC ATs, Zenith Z-248s, and Zenith Z-200 (TEMPEST microcomputer which is basically compatible with the PC ATs and Z-248s).

1.2. PURPOSE

The purpose of this paper is to examine the methods used to test an Ada software product. It addresses the advantages and disadvantages of testing techniques used and makes recommendations that may help future efforts take more effective advantage of the Ada language and its associated development environment. It is to provide a forum for CCSO to share knowledge about testing in the Ada software development environment.

2. SOFTWARE DEVELOPMENT ENVIRONMENT

2.1. EARLY EXPERIENCES

Working with the Ada language was a traumatic experience early in the project. The learning curve was difficult. The staff tried to use the development environment to get maximum advantage of its capabilities. Experience, however, has shown that at best, many mistakes were made in this first project in Ada. Despite the lack of experience, the effort was successful in creating a relatively good product.

Software engineering methods and standards were established during system development. The purpose was to improve aspects of the system in the area of modifiability, efficiency, reliability, and understandability. The design method used is described in the CCSO Design Issues Ada paper dated May 21, 1987. It satisfied the requirements for modifiability throughout the design. The need for efficiency was not pressed as much because we were constantly behind schedule and efficient code was not given priority. Reliability was accomplished by having additional efforts applied during the initial development of a package. Specifically, this was aimed at circumventing failure of the software due to a poor concept of operation, to an incomplete design, or to poor architectural structure. The issue of the understandability goal was handled by developing a set of coding, designing and commenting standards. These software engineering practices were evolved as the development progressed -- some in a very formal manner and others by experience. The cumulative effect of these efforts was positive on this project. They will be better defined as we gain experience to help even more in future projects.

2.2. A CONTINUOUS PROCESS

Testing is a continuous process. Testing became an ongoing automatic process done through various support programs available in the Ada environment for the development team throughout the design and development process. It ensured the integrity of the basic parts of the system. Ada was used to develop compilable specifications. This provided the initial testing at the basic design level. The use of software engineering techniques and the Ada environment that enforced strong typing standards helped eliminate system level problems. These steps forced the programmers and analysts to recognize the effect of their decisions very early in the project, and continuously as it was developed. Corrections were made much earlier and at much less cost to the project. As a result, the stability of the product was greatly improved. This early enforcement of standards provided a positive step in the improvement of the software development environment. The final phase of the testing process was the comprehensive tests done by an independent test group using a suite of tests designed to ensure compliance with requirements.

3. <u>DESIGN</u> <u>LEVEL</u> <u>SPECIFICATIONS</u>

3.1. THE EARLY DEVELOPMENT PHASE

The development process identified eleven (11) major subsystems in the SARAH design. The subsystem interfaces were implemented in code shortly after they were defined. This proved to be a great advantage. Those interfaces were then tested (verified by the compiler) immediately, allowing an opportunity to re-design and re-code when there was only a little code to be changed to implement a "new" design.

3.2. ADVANTAGES OBSERVED

This was where confidence began to grow in the language and the software engineering techniques it supports. These early steps were the beginning of the continuous software testing process we were to later recognize as one of the most beneficial aspects of the language and its environment. This beneficial effect occurred throughout the development of the SARAH project.

3.3. ADVANTAGES OF DESIGN METHOD

Since the resulting design was already in the implementation language there was no chance for design errors to be introduced when the design was implemented into code.

3.4. OTHER ADVANTAGES OF ADA

The Ada environment had the right tools to allow incremental building of the system. It is easy to put in selected stubs, and selected segments of "real" code. This allowed the "analyze a little, design a little, implement a little, and test a little" approach (a more technical description might be "iterative development at increasingly lower levels of abstraction"). This ability to develop and compile and test an incomplete system for proper operation was vital in the development testing. Problems corrected at these higher levels of abstraction proved much easier to fix because dependent subordinate code did not exist. Also, the thorough compiler and strong typing characteristics found a lot of errors before the code could be run.

4. PACKAGING

4.1. PACKAGING CONCEPT

This provided a forum for testing that ensured functional isolation, verification of data integrity, and an immediate decrease in the opportunity for the introduction of errors. The high level subsystem designs were each developed in Ada packages. Subsequent decomposition of the SARAH system into subordinate packages and procedures re-enforced these advantages, each level providing the same advantages of verification of interface and data type that was available for the design level development work. The packaging concept thus permeates the system allowing the total segregation of subsystems and associated data structures.

The SARAH project made use of this concept. However, major coupling problems were built into the implementation. The system has some good modularity but it has plenty of room for improvement. Many of the current problems were caused by lack of time for proper design in the latter stages of the project. The packaging techniques used need improvements. These areas are currently being studied to improve this situation within the system.

4.2. ADVANTAGES OF PACKAGES

The communication package was developed using a stubbed subprogram environment. This allowed the testing of the upper level design without having to worry with the operation of the detailed low level programs that were hardware dependent. This stubbed version later became a formal version for training purposes. The simulated communication package allows practical hands-on operation of the system with responses as it would be in an actual operations environment.

Packaging in the Ada environment aids in the testing of the resultant system because it *supports* the principles of modularity, high cohesion, and low coupling. (It does not enforce these principles.)

4.2.1. Modularity

The system is partitioned into smaller understandable units. Understandable partitioning results in easier identification of the guilty units when a bug is uncovered.

4.2.2. High Cohesion

Most of the data needed for a module is contained within the module. This results in easier to fix modules since, ideally, changes will be limited to the one module to be fixed. As an example take a bug occurring in the read routine of disk. Since most pertinent data is defined inside the routine, corrections are required only inside that one routine. This one "Ada separate module" could then be recompiled and rebound without having to change any other code within the system.

4.2.3. Low Coupling

This is the corollary to high cohesion. Inter-module dependencies are minimized. Accomplishment of this goal is definitely aided by the existence of package specifications making package interfaces explicitly defined and enforced. This is a boon to the system integration phase. Many integration errors are uncovered by the compiler. Integration errors not uncovered by the compiler are often easy to identify since system modularity provides well defined unit scope, responsibility, and effect.

4.3. CREATING TOOL PACKAGES(SETS)

Shared packages of tools were developed, tested and made available to the subsystem developers. These packages provided stability in the development environment. The eleven subsystems were analyzed to identify common data types and facilities needed throughout the system. The results were used to establish specifications for common tool packages. Packages such as the buffer manager, video display terminal manager, and disk manager were produced as a result of this process. After this formal definition of the packages, each was developed and some were tested with test harnesses to verify proper operation. Once a package was operable for the system environment it was made available for the major subsystem developers.

4.4. UTILIZATION OF TOOL PACKAGES

These packages improved the stability of the SARAH system development. They used proven logic and prevented the introduction of errors that occur if many versions of the same logic are peppered throughout the system. It centralized the areas involved if and when error corrections or modifications were needed, greatly reducing the potential for the introduction of errors. We experienced surprisingly quick and smooth subsystem implementations because of this stability.

4.4.1. ANALYST COMMENTS ON PACKAGING

These are a few of the comments made by the development staff about the Ada environment and its packaging and modular architecture:

"Ada's packaging concept aids testing because it supports the principles of modularity, high cohesion, and low coupling.

"Ada can be a very valuable testing tool during the code and checkout phase, and the system integration phase of development. It is less valuable during formal system test."

"The other problem we encountered was the fact that we were developing code before the support code(tool boxes) were written. Going back and modifying code later to use this new support code caused all sorts of problems and resulted in a complete retesting each time a new subsystem was incorporated. (i.e.. disk, buffer manager).... I believe that if we would have had the support packages ... that the basic editor could have been built and tested as each function had been coded and that the testing would have been a lot smoother and easier."

"I think that the facility to modularize the code allows for smaller chunks to be tested rather rapidly and new functions can be added and tested with ease."

"Incremental testing (i.e. the testing of individual modules or subsystems) was, however, aided by the use of Ada, primarily because of the package aspect of the language. This facilitated the testing by allowing confirmability of the operation of a particular function in isolation of the remainder of the system. This aspect is of particular importance when desiring to ensure that no previous operation has been adversely affected by a modification to a previously tested module."

"The language structures of Ada made software testing easier than previous assembler language projects done here at CCSO. The strong type checking and modularity prevented us from making costly mistakes. However, I'm not convinced that Ada provides greater advantages than other modern high order languages (Pascal, Modula2, etc.)."

"Modularity is not unique to Ada. It is a programming technique that can be applied to many (maybe all) other languages. It is a technique that could be good or bad regardless of the language used. However, if modularity had not been used, a lot of the other features of Ada could not have been used. i.e. incremental compilation, information hiding, strong typing in passed parameters, software tools."

"In the long run, if we can make reuse of modules, Ada may provide an edge in testing because each individual module will have been tested when it was developed and again when it is reused. The more testing (both in a test environment and in the field) a module gets the more reliable it becomes."

4.5. SEPARATE COMPILATIONS

The compilation process produced advantages quickly for these subsystem packages. Almost everyone took advantage of separate compilations. The strong object typing enforced by the compiler between modules and inside instructions improved the integrity of the resultant code dramatically. Basically, the programmer is forced to face the results of bad design before the software is executing as a program. Inconsistencies in these areas are not al'^wed and are brought to the attention of the programmer in the form of specific diagnostic messages. The result is that the code has greater integrity in the overall system environment.

4.5.1. ANALYST COMMENTS ON COMPILER DIAGNOSTICS

"These were good by themselves; but when added to strong typing in passed parameters, modularity, constraint error exceptions, limited data manipulation, private types, incremental compilation; they were even more useful. This is a good example of where the whole is greater than the sum of its parts. The combination allowed test and debug solutions to be determined easier."

5. USE OF GENERICS

5.1. STABILITY OF CODE

Another facility of the Ada environment that proved beneficial to the validity of the final product was the use of generics. Basically that is the use of a template program that can be created (instantiated) in a module to accomplish a common type function on data structures unique to that module. Generics were not used to any great extent in this implementation due to our lack of understanding of their usefulness and to the fact that it appears that they cost an excessive amount of memory to instantiate.

Generics were used by some programmers to improve the stability of frequently used procedures. They did effectively prove to be a stabilizing factor. After the generics were tested they provided no problems. The major difficulty encountered with the generic was that our debug tool Alsys "AdaProbe" would not operate inside a generic instantiation. This made the debugging of complex generics very difficult compared to straight non-generic coding.

5.2. REDUCTION OF COMPLEX CODE

Within the SARAH system there is a screen masking system developed for the entry of different message formats. Each message type requires different sets of screen masks or templates because of varying requirements for the different formats. A data structure was developed to satisfy these differing mask requirements in a non-generic form. After the message mask was implemented, a need for additional screen masks was defined for the SARAH configuration package. There was no way the implemented code could be used. The new requirement functionally operates the same way as the original mask. The new requirement also had to satisfy screen input requirements for both a large routing database file management system and the site terminal specific table entries. The screen functions were to be identical but the application and type of data was entirely different. To meet these requirements, and to prevent the creation of two more screen management systems, a generic solution was developed. The resulting generic was instantiated into each application. Problems were encountered in the first instantiation and corrected in a timely manner. The second instantiation worked with no problems. The screen operation was stable and met all requirements.

5.3. EASE OF CORRECTION AND IMPROVEMENT

The screen requirements were changed to improve the capability of the generic and to reduce the coupling of the application programs to the data structures in the generic. The changes were made in the generic and the implementation was tested in both instantiations. The fix in the generic took care of the problem in both applications of the packages. This provided a great amount of confidence in the techniques being used and the developing product because the product had less chance for the introduction of errors during the correction and enhancement processes.

6. USE OF TASKING

6.1. TASKING WAS DIFFICULT

Testing of the SARAH system was greatly inhibited at the development level because of tasking. Since little was known or understood about it, it was a trial and error process. The communication and input/output systems had to use tasking. Our greatest testing problems occurred because of this environment. As noted by the analyst quote below, the diagnostic debugger did not function well in the tasking environment. Therefore our testing and isolation of logic problems required the programming of diagnostic logic. This was difficult and time consuming.

6.2. ANALYST COMMENTS ON THE USE OF TASKING

"One feature of Ada that hinders testing is tasking. Because the timing of tasks (which task will become active if several are waiting, exactly how long a task will stay active, etc.) is not exact, we experienced what were apparently different results from the same sequence of events. Tasks are particularly hard to debug because we could not use the Alsys AdaProbe. Trying to use it changes the timing and many times the problem does not occur (you may even be unable to breakpoint at the spot where you need to)."

7. THE ADA PROGRAMMING SUPPORT ENVIRONMENT(APSE)

The Ada Programming Support Environment (APSE) consists of the configuration management, command processor, editor, compiler, debugger, and linker/loader. These facilities of the Ada development environment provide capabilities that greatly influence productivity and the stability of the software product. The concept of a programming support environment is probably more valuable to testing than the Ada language. A proper APSE provides the software engineer a cadre of interactive development and testing tools. The Alsys system we used provided a library management system, a compiler, a debugger, and a linker/loader. It is not a complete APSE. It did provide a good library management system that resolves some configuration problems and an excellent debug tool set that assists tremendously in integration testing.

7.1. ADA LIBRARY MANAGEMENT SYSTEM

7.1.1. AUTOMATIC LIBRARY FUNCTIONS

A strong reason for the improvement of the software product is the library management system. It prevents the use of old compilations of subordinate programs when a specification is recompiled. This automatically prevents the creation of executable code from outdated subordinate program files. This feature is another continuous step throughout the development process that tests and ensures the integrity of the software being produced. In addition, the compiler checks the usage of a package by other packages and automatically requires users of a recompiled package to be recompiled. This prevents the inadvertent creation of a system with the wrong version of packages included in the executable code.

7.1.2. DISADVANTAGES IN THE LIBRARY SYSTEM

File dating is not managed by the library management system under the environment for the SARAH project. Thus generations of a file, i.e. father, grandfather, greatgrandfather, etc., are not managed by this system. Current methods use MSDOS BAT files to control configuration compilations. This allows the creation of executable code using older versions of a program module without being aware that more current versions exist. It has proven to be a source of major problems for our configuration environment.

7.1.3. ANALYST COMMENTS ON ADA LIBRARY MANAGEMENT

"The object code library concept that Ada provides can aid greatly in subsystem and integration testing. A separately compiled unit can quickly be compiled in, rebound and retested with confidence that the only code changed was the one unit that was recompiled (don't have to recompile the entire system or subsystem)."

7.2. ADA CODE EXECUTION TEST TOOLS

7.2.1. WHEN TO USE TEST TOOLS

Debug tools proved to be essential in the development testing and integration process. It was also necessary for diagnostic, isolation and correction of problems when the independent testing group identified them. In addition to all the previous procedural and environmental circumstances that have helped create a more stable product, the system could not have been developed without the vendor's debug tool system.

A tool to breakpoint and step through the executing program is an absolute necessity. There is no other way to check reason and logic automatically. Development work was done during the early stages of the system when it was very simple without the aid of a debugging tool. Great difficulties were encountered in trying to isolate the simplest problems. Debug tools were ordered when their criticality to the project was realized . An example of the problem is an error that was present in the "EDIT" program. It was studied and tested for three days without solution. Then the Alsys "AdaProbe" arrived and the problem was located in about 5 minutes.

7.2.2. ALSYS ADAPROBE

The debug tool used was the Alsys "AdaProbe." This tool provided direct visibility into the source code for walking through problem code as it executes. This debugging tool, however, did not function well in the tasking environment.

7.2.3. TEST HARNESSES

A technique used by some developers was the development of a subsystem driver that presented a series of controlled input parameters to a subsystem package. The output from the package was checked for the expected result to confirm proper operation by the driver. This proved to be a beneficial exercise for packages such as the disk, vdt and buffer packages.

7.2.4. ANALYST COMMENTS ON THE PROBE DEBUG TOOL

"Adaprobe is a very powerful symbolic debugger. Without this tool SARAH development may well have taken an additional 6 to 12 months."

"This was an excellent tool. Debugging would have been much slower and more cumbersome if this tool had not been available."

"Some systems do not have good debug tools. With them, it is necessary to stop the system at specific memory locations (which may involve hex or octal addition or subtraction with a hardcopy of the software at hand); look at registers and memory locations; and translate data to a recognizable form (with a hardcopy of the software at hand)."

"With probe, we were able to view the software on line, use the cursor to mark lines/data (without determining memory locations), and examine data easily and quickly without having to translate it."

8. LESSONS LEARNED

The continuous checking for conformance to interface specifications and data typing done by the compiler forced adherence to design. This helps improve the validity of the software product constantly throughout the development process.

Packaging in the Ada environment can greatly improve the correction and maintenance situations in the system and reduce the opportunity for the introduction of errors into the system.

The actual implementation of modular, highly cohesive, and low coupled environments is not easy and requires a deeper understanding of the complete environment than was available on the project.

The debug tools should be ordered with the compiler. They are a necessity for diagnostic work to isolate problems.

Test harnesses are good for verifying the correctness of a set of packaged procedures. It helps ensure a good product before other portions of a system become dependent on it.

The tasking and generic environments make testing difficult since the debug tool used did not work well in these type of programs when trying to diagnose runtime problems.

9. <u>SUMMARY</u>

In summary, the Ada environment provides the tools to make testing a more continuous, efficient and complete process. The Ada language can be used in the absence of proper design and proper tools. You may be no better off than with FORTRAN or any other language.

The Ada environment causes very early identification of errors and forces corrections. The reuse of modules and creation of selected tool boxes for commonly used functions provide increased reliability in the product. Generics are a potentially powerful facility that can be used to great advantage but it must be done correctly and administered properly to be beneficial. The programming support environment provides the capabilities to greatly improve the products.

Ada is not a panacea. It is difficult to use to your advantage because of its versatility and infinite number of ways to apply it, but it can improve the stability of software products if applied using proven software engineering practices.