February 1989

MTP282

David E. Emery
Karl Nyberg
(Grebyn Corporation)

Observations on Portable Ada Systems

CONTRACT SPONSOR N/A
CONTRACT NO. F19628-89-C-0001
PROJECT NO. 022A
DEPT. D74

MITRE

The MITRE Corporation
Bedford, Massachusetts

ABSTRACT

This paper presents observations from the authors' experience porting large Ada software systems. Software is ported to reproduce its functionality on a new machine/operating system. The basic reason for porting software is that it is (supposed to be) much less expensive than developing new software with the same functionality. We present eight 'lessons learned' with examples from our experience. Our primary finding is that highly portable systems have a strong model of their underlying host environment. These models are adaptive and are well documented and tested. Finally, we provide recommendations to software designers and software project managers to make the software porting task easier, and thereby much more cost-effective.

ACKNOWLEDGMENTS

We would like to thank Geoff Mendal for sharing his experiences porting the Annotated Ada (ANNA) toolset and his review of this paper. Rich Hilliard, Marlene Hazle, Chris Byrnes, Anthony Gargaro, and Olimpia Velez also provided valuable comments. Finally, we would like to recognize the Intermetrics Ada Integrated Environment (AIE) design team. This paper started as an attempt to understand what makes the AIE so portable.

TABLE OF CONTENTS

SECTION 1

INTRODUCTION

1.1 WHAT IS SOFTWARE PORTABILITY?

We define portability as the relative effort required to get a piece of software running on one host to run on another. A 'host' in this sense consists of at least the underlying computer architecture, the operating system, and the Ada compilation system (both compiler and runtime environment) on which the software is intended to execute (Kaindl 1988). Specifically, portability is a characteristic of software measured by the effort to implement the required functionality on a new host from scratch compared to the effort to rehost the software onto the new host. 'Perfectly portable' software would require no more than recompilation to get it running on the new host.

There is a strong, but not well-defined, relationship between software portability and software reuse. We make the following distinction: porting is moving an entire system to a new host environment while preserving functionality; reuse is using a system (or parts of a system) to achieve different functionality. Portable software and reusable software share many of the same characteristics; portable software is probably easier to reuse, and reusable software is probably easier to port.

It is our experience that there are two kinds of systems for which porting is difficult. The first kind requires small changes in many places. Such systems are hard to port because of the large number of units that must be identified and changed. The many disparate changes are an indication that these systems do not have well defined virtual layers, and that the separation of concerns between the various layers has not been identified. The second kind are those where there may be only one package containing all system dependencies, but porting this package requires substantial work. In some cases, it is all but impossible to port this package to the new host, and the entire port fails. This failure may be due to some necessary element that is not provided by the new system (e.g., a real-time clock with sufficient resolution) or because providing the necessary service is prohibitively expensive (in terms of effort to get the software to run, or in terms of execution costs).

Simply using Ada is not sufficient to produce portable systems. Ada has done much to reduce compiler dependencies, but Ada does not cover many host services required by software systems. Command line parameters, process creation (running another program) and in particular file naming are required by most software systems that we have ported. Ada's emphasis on modularity and its support for abstraction provide linguistic assistance for limiting host dependencies (Nissen & Wallis 1984). Ada does not mandate that the programmer use these features to support portability. As has been observed, "It is possible to write bad code in any language!"

SECTION 2

OBSERVATIONS FROM OUR EXPERIENCE

2.1 PORTABLE SOFTWARE HAS A STRONG MODEL OF ITS HOST

The common feature of portable software is its understanding of the host environment. Systems that are easiest to port have a clear understanding of their existing host, possible future hosts, and their requirements for executing on a host. Specifically, portable software uses (either on purpose or by accident) a model of the host environment that both meets the needs of the software and that is feasible on a significant set of hosts (Bardin 1987). This may be achieved by determining a "weakest implementation" (Pappas 1985) or "lowest common denominator" (KIT [no date]) for existing and potential hosts. Developing a model of the underlying host is different from simply encapsulating system dependencies. The difference is one of consistency and completeness.

There are often implicit dependencies on the host that are not expressed in a SYSTEM_DEPENDENCIES package. One example is a program that needs to know if a certain file exists in the current directory, or in some other specified directory. If the only operation on files is OPEN, there is no explicit concept of "directory," or "current directory." This can create problems when porting to a system that does not support hierarchical directories, or whose syntax for naming directories is different. The SYSTEM_DEPENDENCIES package does not present a complete model of the system's requirements and assumptions about the underlying file system.

Developing these models is clearly a design activity. Software architects have to determine how to meet their requirements on a particular host system and also consider potential future hosts. The goal is to produce a model of the host environment that meets the requirements of the software, and that is easy and efficient to implement on the current host as well as on future hosts. Unfortunately, such design is still a bit of "black magic" (Lampson 1984).

We observed a "good" model when trying to port a programming environment from VMS[1] to UNIX.[2] Most programming environments need to provide a mechanism for invoking a "foreign tool" whose command line parameters are not known when the environment is developed. In this programming environment, the designers came up with a model for tool parameters that consisted of two parts, 'parameters' and 'options.' This was apparently inspired by the syntax used on VMS, the initial development host. This model proved to be a very useful one, because it could be used in a variety of situations. It turned out that most programs can support this distinction, including UNIX tools that get some of their information from UNIX 'environment variables.'

When the environment was ported to UNIX or a similar system that does not differentiate between command line parameters and options, this model is still valid. Most UNIX tools support the distinction between parameters and options via a convention (options, called switches in UNIX terms, are

[1]VMS is a trademark of Digital Equipment Corporation.
[2]UNIX is a trademark of AT&T Bell Laboratories.

prefixed by the '–' character, but the UNIX shell does not enforce this convention). UNIX tools could be easily integrated into the programming environment's model. Constructing a 'command builder' that can interpret the programming environment's "call" and translate that into a operating system specific invocation is easy to do.

This model has the additional value that it can be used to support UNIX 'environment variables,' which are a special case of 'options.' The 'command builder' can interpret some options as a request to set a specific environment variable, rather than providing the value via an actual operating system command line element.

An example of a "bad" model occurred in the same programming environment. One of the services provided by the programming environment was intended to be a machine-independent (and hopefully portable) 'virtual' file naming syntax. This system was initially developed on top of VAX/VMS, but was expected to be portable to UNIX and to MS-DOS[3].

VMS provides a primitive file version numbering system. In general, whenever a file is accessed on VMS, the file is not updated in place. Instead a new version of the file is created. Versions are numbered sequentially, starting with 1. By default, if no version number is specified, VMS retrieves the version of the file with the greatest version number (which is presumed to be the most recent version). This facility is directly implemented by the operating system, and the syntax for naming versions is built into the file system.

The designers of the environment decided to adopt the VMS model for sequential versions of files. Their syntax for 'virtual file names' included an optional sequential version number. The file naming syntax was presented to the rest of the environment as a model of the host system's file system. The interpretation of these names was performed by system-dependent code.

Consider how this model can be implemented on a system that does not support version numbering and also has a fixed format for file names, such as MS-DOS. On VMS, the model can be implemented using a one-to-one mapping between 'virtual file name' versions and specific VMS version numbers of a file. On some versions of UNIX, this could be implemented by adding the version number to the 'virtual file name' to produce a UNIX file name. However, on MS-DOS, there must be a much more complex mapping function between the 'virtual file name,' and the version number of that name, to an MS-DOS file. This mapping must be maintained on disk, because it must be shared among various processes and survive system shutdowns. Such a system can be implemented on MS-DOS, but its performance is poor, because of the cost of interpreting the mappings followed by the cost to actually retrieve the appropriate MS-DOS file.

The designers presented a coherent model of the system, but the model was not efficiently implementable on one of its likely targets. Porting this software to MS-DOS required a substantial amount of work to implement the sequential version abstraction.

[3]MS-DOS is a trademark of Microsoft Corporation.

2.2 PORTABLE SOFTWARE VERIFIES ITS MODELS

It is not sufficient for portable software to identify a model of its host. This model must be correctly implemented. Portable software should verify that its host interfaces are correctly implemented. Testing is one approach to this. Portable software should provide test suites for its models of the host. Ideally, these tests should be self-diagnosing, in that they should report success or failure. Test data for a model also helps to further document the expected behavior of the model. Another way to verify (and document) host interfaces is to use a formal specification system such as ANNA (Rosenblum 1988, Luckham et al 1987).

The original release of the Ada binding to X Windows (Hyland & Nelson 1988) contained no test programs, not even a simple Hello_World program. It was difficult for us to determine if our port had been successful. When we developed a test program for the binding and it failed, we were unsure if this reflected a problem in the binding, in our test program, or in our X Windows library. The next release of the binding came with several sample programs. Although not a comprehensive test suite, these tests at least provided us with some confidence that the port was successful.

The Intermetrics AIE contains a significant number of tests for the important system interfaces (Intermetrics 1987). These tests would report failure, and in some cases the tests also reported the cause of the failure. One important set of tests is the stress tests, particularly those that test multiple access to the program library. File system dependencies often contain implicit assumptions about concurrent access to the same file by different programs, and the AIE stress test ensures that the software behaves correctly in the face of concurrent users.

2.3 PORTABLE SOFTWARE IS ADAPTIVE

"Software, port thyself!" Some of the most portable systems have performed their host analysis to the point that they are almost self-porting. Many problems occur at the point where the software exchanges data with the host. One topic for adaptive software is how to translate its data into the host's format. Ada provides excellent support for adaptability through the intelligent use of data type attributes when coding.

Another place for adaptive software comes in system tuning and performance monitoring. Portable software should come with instructions and code that analyze the performance of the software and identify hot spots for tuning during the porting process. Portable software follows the general 90%–10% rule for software, in that 90% of the time is spent on 10% in the code, and this is the code that should be tuned.

The Intermetrics AIE contains several examples of adaptive software. One of the best examples occurs in a software structure that emulates virtual memory. In this system, a reference to a virtual location is interpreted as a page number, and an offset to a location within that page. When porting the software, the user can tune the system by adjusting the page size, and the software will automatically derive the optimal packing of information on a page.

Given a page size of N storage units, the problem is to define a page as an array of integers. In Ada terms, this can be calculated as shown in Figure 2-1:

```
        -- First define a type whose size is System.storage_unit
        -- This is usually a byte or a word, depending on the
        -- architecture. This type definition should be checked for
        -- each new implementation
type stored_unit is range -2 ** (System.storage_unit  -1)
                          .. 2 ** (System.storage_unit - 1) -1;
for stored_unit'size use System.storage_unit;
        -- next figure out how many storage_units per integer
storage_units_per_integer : constant := integer'size /
                                        System.storage_unit;
        -- then how many integers fit on a page of N storage_units
integers_per_page : constant := N / storage_units_per_integer;
        -- and how much is left over
leftovers_per_page : constant := N -
                (integers_per_page * storage_units_per_integer);
        -- now we know enough to correctly define a page
        -- of virtual memory
type page_array_type is array (1..integers_per_page) of integer;
type overflow_array_type is array (1..leftover_storage_units) of stored_unit;
type page is record
   data : page_array_type;
  overflow : overflow_array_type;
end record;
for page'size use N;  -- this now serves as an "assertion check"
```

Figure 2-1. Adaptable Page Table Definition

Notice that the only thing that must be verified to port this software to another machine is the type definition for Stored_Unit. In particular, this will work for any value of N (greater than or equal to storage_units_per_integer). Different values of N can be provided during performance analysis to determine the optimum value for N for a given machine.

Another excellent example of adaptability occurs in GNU Emacs. GNU Emacs uses the UNIX make (Feldman 1979) facility for tailoring GNU to different hosts. To construct Emacs for a specific machine, there are a few 'macros' that the user provides. The makefile does conditional compilation based on the values of these macros. For example, default pathnames to Emacs library files can be provided as a makefile macro. Machine dependencies are also stored in specific C header files, and the makefile selects the appropriate file for the target machine. All these machine dependency files follow a pattern, so constructing Emacs for a new UNIX machine requires producing a new header file for that machine, based on the existing header files, and instructing make to use that header file when compiling and linking Emacs. Unfortunately, Ada does not fully support this technique, because there is no equivalent to the C preprocessor (used to insert a string from the environment at compilation time into the source code).

2.4 PORTABILITY HAS AN EFFECT (BOTH POSITIVE AND NEGATIVE) ON PERFORMANCE

The claim has been made that portable software is inherently less efficient than software that has not been designed with portability in mind. Portability has an effect on performance, but it is not necessarily negative. Adaptive software can provide the compiler with more information, because the compiler is expected to figure out a data layout, or array bounds, for instance. Given this additional information, a good compiler can perform more optimization, making the software perform better on both the current and any future host.

Performance is also directly affected by the model of the host used by the software. Choose the wrong model, and the host interface becomes a substantial bottleneck (as shown previously). On the other hand, a good model can facilitate (but not guarantee) substantial performance enhancements.

Practical experience indicates that most portable software initially exhibits some degree of inefficiency, usually because the ported software duplicates some service performed by the host (Mendal 1988). However, if this service is identified, then replacing the ported software's implementation by the host's implementation should be very easy to do, and cause significant performance improvement.

The Intermetrics AIE presents three models of its Host Interface (HIF). The External Interface is used by the rest of the software to obtain HIF services. Below it sits the Indexed-Sequential Access Method (ISAM) Interface, so the external interface software is implemented in terms of ISAM files. Below this interface is an implementation of ISAM that uses fixed-size file pages. This lowest layer is implemented in terms of Ada's DIRECT_IO. This software provides its own buffering scheme, which can be eliminated if the host's implementation of DIRECT_IO provides efficient buffering. However, should the new host operating system provide an equivalent ISAM, it is very easy to replace the implementation of the ISAM model with calls to the host's ISAM facility, eliminating calls to the Ada DIRECT_IO package.

2.5 PORTABLE SOFTWARE USES STANDARDS

There is an obvious gain in portability from using directly applicable standards. But there are other uses for standards in portable software. Designing a good interface is hard work. Most standards have undergone substantial rigorous public review, and represent a clear model of their underlying system. Often the model embedded in a standard can be used as the basis for defining the software's model of a different host.

POSIX (the IEEE Standard Portable Operating System Interface based on UNIX) presents a clear example of this. Both Microsoft and Digital Equipment Corporation (DEC) have announced that they will implement the POSIX interface on OS/2 and VMS (respectively). There is at least one proposal (Harbaugh 1988) for a common file naming package that uses a variation on the UNIX file naming scheme. Software that uses a UNIX style of file naming should be easily ported to a wide variety of systems that support hierarchical directories.

Another example of this is illustrated by the many compilers that use some variant of Descriptive Intermediate Attributed Notation for Ada (DIANA) (Evans et al 1983) as their internal representa-

tion. Almost every compiler vendor has modified DIANA to meet his individual requirements (so no two DIANA implementations are the same). Several implementors have publicly acknowledged the savings by adapting DIANA, rather than developing a new representation from scratch (Milton 1983, Mendal 1988).

2.6 PORTABLE SOFTWARE COMES WITH A PORTING MANUAL

The hardest part of the porting process is identifying system dependencies. Therefore, software that is expected to be portable should come with a Porting Manual that identifies these dependencies.

Figure 2-2 is a Table of Contents for a Porting Manual:

1. Overview of This Manual
2. The Porting Process
   a. Resources Required
   b. Sequence of Events
   c. Estimated Schedule
3. Structure of the Software
   a. Compilation Order
   b. Compilation Dependencies
   c. System Dependencies
4. Modifying Dependent Units for Each Unit
   a. System Dependencies
   b. Porting this Unit
   c. Performance Considerations
5. System Test and Integration
   a. Overview of Test and Integration Process
   b. Unit/Subsystem Test Suites
   c. Integration Test Suites
6. Performance Monitoring and Tuning
   a. Overview of Performance Monitoring and Tuning
   b. Potential Performance 'Hot Spots'
   c. Tuning Parameters
   d. Performance Tests and Tuning Suites

Figure 2-2. Porting Manual Contents

This Porting Manual identifies the system dependencies, permitting the person doing the port to concentrate on those units (and ignore other units). The required model of the interface is documented, both on paper and also through the use of tests that provide an 'operational verification' of the behavior of the model. The manual identifies the sequence of operations needed to port a complex system to the new host, and should provide resource and schedule estimates. The manual provides information to support the task of tuning and adaptation needed to make the system perform adequately on the new machine, factors that are often ignored when trying to get software to compile on the new host.

2.7 PORTING SOFTWARE STRESSES COMPILERS

An amazing number of compiler bugs and dependencies can be found when moving code from one mature compiler to another, even on the same computer and operating system. Ada has gone a long way towards reducing the number of implementation dependencies, and the ACVC tests try hard to ensure conformance with the language Standard. However, what works on one compiler often breaks another.

There are several different categories of language problems that porting can uncover. First are implementation freedoms in the language, such as the order of evaluation of expressions. Another class of problems are compiler bugs. These usually occur as a result of interactions of Ada features. Finally there are the set of issues that concern **pragma** INTERFACE and similar services. Some compilers do not support **pragma** INTERFACE at all, while others place restrictions on the subprograms that may be called from Ada. Passing data across an inter-language procedure call can also cause problems.

We observed several examples of implementation freedoms when porting the X Ada binding. For instance, the following compiled correctly on several compilers, but generated a compilation warning on another:

```
type X_Integer is range -2**31 .. 2**31 -1;
type X_Natural is new X_Integer range 0 .. 2**31 -1;
```

The compiler flagged the definition of X_Natural, warning that evaluating the upper bound on the range of X_Natural would yield a value (2**31) out of the range of X_Integer. In this case, there was an implicit dependency on evaluating an expression.

Another example from the X Binding concerns **pragma** INTERFACE. Most UNIX-based compilers provide the ability to specify the linker name of an 'interfaced' subprogram, but they do it differently. One compiler supports an optional third parameter to **pragma** INTERFACE, and others support a separate pragma, called INTERFACE_NAME. Compiling one version of the binding on another compiler causes these pragmas to be rejected but the compilation itself succeeds. This then causes an error when trying to link a program using the X Binding packages.

2.8 TO PROVE PORTABILITY, DELIVER ON MULTIPLE HOSTS

If portability is a major concern, then the best (and perhaps only) way to achieve it is to require delivery on multiple hosts. This is an approach that was taken in the early Software Technology for Adaptable Reliable Systems (STARS) work, and appears to be effective (STARS 1988). STARS contractors are required to deliver code that executes on multiple host configurations using compilers from different vendors. There are two approaches to multiple deliveries. The first way is to develop on one host, and then port the software to another host. This is commonly the way software is developed and, as a result, ported (Bowles 1988). The other way is to require parallel development and delivery.

There are benefits to developers from the parallel development approach. By using two different hosts, a failure (e.g., compiler bug) on one does not stop development. Bad design decisions can be

identified early on, particularly via prototyping interfaces on both hosts. Additionally, the use of multiple hosts forces the designers to deal with two possibly different systems as current hosts for the software, and not unknown potential future platforms. This will help in the development of ap propriate models of the execution environment through practical experience. Finally, one of the hosts may prove to be a much more cost-effective development host, providing some real financia benefits to the project (Carstensen 1987).

SECTION 3

RECOMMENDATIONS

3.1 RECOMMENDATIONS TO DESIGNERS

- Know your hosts.
- Spend the time to develop a good model of your requirements on the host.
- Use standard interfaces, even if it requires some work to implement your specific requirements on top of the standard.
- Prototype the models on several machines.
- Document these models from a portability perspective.
- Include portability as a topic during Design Reviews.
- Use the capabilities of Ada and adaptive algorithms to let your software port itself to a new host.
- Adopt design and coding standards and guidelines that emphasize portability.
- Develop tests of your interfaces that are both comprehensive and self-diagnosing
- Identify performance 'hot spots' in the interfaces.

3.2 RECOMMENDATIONS TO MANAGERS

- Plan to spend much more time during design to support portability.
- Provide the resources to design, prototype and evaluate host models.
- Try to use multiple systems during development; do parallel development if possible.
- Be prepared for compiler problems.
- Deliver a Porting Manual.

SECTION 4

CONCLUSIONS

Portable software starts with portable architecture and designs. Thus the emphasis is upon the software designer, not the implementor. The key technical issue in portability is producing the right model of the host(s) to which the software will be targeted. This is not something that can be solved by coding standards or style guides, but is a design (perhaps even a requirement analysis) activity that needs to be undertaken before coding.

Management has its role in portability, too. Management must decide to pay the additional up-front cost and schedule to achieve a high level of portability. The best way to develop portable software is to do parallel development on multiple hosts. This requires management commitment, in terms of time and money.

Our experience indicates it is possible to develop software systems that are easy to port. Ada assists in this by providing facilities to help the designers and implementors, but Ada is not sufficient. This paper presents some observations from our successful porting efforts. We hope software developers will apply these observations, to make our next software port easier.

SECTION 5

REFERENCES

Bardin, B., *Layered Virtual Machines + OOD — a Balanced Refinement Methodology*, Boston, MA: Proceeding AIAA Computers in Aerospace VI, 1987.

Bowles, K., *Lessons Learned in Ada Developments*, Washington, DC: Presented to ACM DC Local, SIGAda, 1988.

Carstensen, H., *Magnavox Lessons Learned from AFATDS*, Boston, MA: Presented to Ada Expo, 1987.

Evans, A., et al., *Descriptive Intermediate Attributed Notation for Ada (DIANA) Reference Manual Revision 3*, Pittsburgh, PA: Tartan Labs, Inc., 1983.

Feldman, S., *Make — A Program for Maintaining Computer Programs*, Software — Practice & Experience, Vol 9, No 4, 1979.

Harbaugh, S., *Universal FileNames Package*, in Proceedings 1988 STARS Workshop, Washington, DC: STARS Joint Program Office, 1988.

Hyland, S., and Nelson, M., *The Ada Binding to X Window System*, Princeton, NJ: Presented to ACM SIGAda, 1988.

Intermetrics, Inc., *Rehost/Retarget Manual for the Ada Integrated Environment (IR-MA-826-0)*, Cambridge, MA: Intermetrics, Inc., 1987.

Kaindl, H., *Portability of Software*, ACM SIGPLAN, Vol 23, No 6, 1988.

KAPSE Interface Team (KIT), *Ada Tool Transportability Guide*, Washington, DC: Ada Joint Program Office.

Lampson, B., *Hints for Computer System Design*, IEEE Software, Vol 1, No 1, 1984.

Luckham, D., et al., *Anna — A Language for Annotating Ada Programs*, Berlin, FRG: Springer Verlag, 1987.

Mendal, G., Experience Porting Anna, personal communication, 1988.

Milton, D., *Lessons Learned from Developing the Verdix Ada Compiler*, Tinton Falls, NJ: Presented to ACM Princeton Local SIGAda, 1983.

Nissen, J. C. D. and Wallis, P. J. L., *Portability and Style in Ada*, Cambridge, England: Cambridge University Press, 1984.

Pappas, F., *Ada Portability Guidelines (ESD-TR-85-141)*, Waltham, MA: SofTech, Inc., 1985

Rosenblum, D., *Design and Verification of Distributed Tasking Supervisors for Concurrent Programming Languages, Ph.D. Thesis*, Stanford, CA: Stanford University, 1988.

Stallman, R., et al., *GNU Emacs Documentation*, Cambridge, MA: Free Software Foundation, 1987.

STARS, *Proceedings 1988 STARS Workshop*, Washington, DC: STARS Joint Program Office, 1988.

SECTION 6

ADDITIONAL BIBLIOGRAPHY ON ADA PORTABILITY

Brosgol, B., and Cuthbert, G., *The Development of the Ada Binding of the Graphical Kernel System*, Ada UK Ad User, Vol 8 Suppl, 1987.

Brown, P. J., ed., *Software Portability*, Cambridge, England: Cambridge University Press, 1972.

Emery, D., *Experience Using Pragma Interface*, Princeton, NJ: Presented to ACM SIGAda, 1988.

Fisher, G., *A Universal Arithmetic Package*, ACM Ada Letters, Vol 3, No 6, 1983.

French, S., *Transporting an Ada Software Tool: A Case Study*, ACM Ada Letters, Vol 6, No 2, 1986.

Gargaro, A., and Pappas, T. L., *Reusability Issues and Ada*, IEEE Software, Vol 4, No 4, 1987.

Genillard, Ch., and Ebel, N., *Reusability of Software Components in the Building of Syntax-Driver Software Tool in Ada*, Cambridge, England: Cambridge University Press, 1986.

Goodenough, J., and Probert, T., *Designing and Testing Interfaces for Portable Software: Ada Text_IO as a Example*, in Ada Software Tools Interfaces, P. J. L. Wallis, ed. Berlin, FRG: Springer Verlag, 1983.

Goodenough, J., *The Ada Compiler Validation Capability*, ACM SIGPlan Notices, Vol 15, No 11, 1980.

Koh, J., and Sym, G. T., *A Proposal for Standard Basic Functions in Ada*, ACM Ada Letters, Vol 4, No 3, 1984

Kurtel, K., and Pietsch, W., *A Portable Ada Implementation of Indexed Sequential Input/Output*, ACM Ada Let ters, Vol 6, Nos 2 and 3, 1986.

Lecarme, O., and Pellisier Gart, M., *Software Portability*, New York, NY: McGraw-Hill, 1986.

Matthews, E., *Observations on the Portability of Ada IO*, ACM Ada Letters, Vol 7, No 5, 1987.

Nyberg, K., *Porting Ada Applications Between a PC and a VAX*, Washington, DC: Presented to Capital Area PC Users Group Special Interest Group for Ada, 1988.

Rehmer, K., *Development and Implementation of the Magnavox Generic Ada Basic Math Package*, ACM Ada Letters, Vol 7, No 3, 1987.

Vines, D., and King, T., *Gaia: an Object-Oriented Framework for an Ada Environment*, ACM SIGMOD Record, Vol 17, No 3, 1988.

Willman, H., *APSE Portability Issues — Pragmatic Limitations*, Boston, MA: Presented at ACM AdaTEC, 1982.