DEVELOPING PORTABLE SOFTWARE

James D. Mooney
Lane Department of Computer Science and Electrical Engineering, West Virginia University, PO Box 6109, Morgantown, WV 26506 USA

Abstract: Software portability is often cited as desirable, but rarely receives systematic attention in the software development process. With the growing diversity of computing platforms, it is increasingly likely that software of all types may need to migrate to a variety of environments and platforms over its lifetime. This tutorial is intended to show the reader how to design portability into software projects, and how to port software when required.

Key words: software engineering; software portability

1. INTRODUCTION

Most software developers agree that portability is a desirable attribute for their software projects. The useful life of an application, for example, is likely to be extended, and its user base increased, if it can be migrated to various platforms over its lifetime. In spite of the recognized importance of portability, there is little guidance for the systematic inclusion of portability considerations in the development process.

There is a fairly large body of literature on aspects of portability. A comprehensive bibliography is provided by Deshpande (1997). However, most of this literature is based on anecdotes and case studies (e.g. Blackham (1988), Ross (1994)). A few seminal books and papers on portability appeared in the 1970s (e.g. Brown (1977), Poole (1975), Tanenbaum (1978)). Several books on software portability were published in the 1980s (Wallis (1982), Dahlstrand (1984), Henderson (1988), LeCarme (1989)). None of these publications provide a systematic, up-to-date presentation of portability techniques for present-day software. This tutorial offers one approach to reducing this void.

Well-known strategies for achieving portability include use of standard languages, system interface standards, portable libraries and compilers, etc. These tools are important, but they are not a substitute for a consistent portability strategy during the development process. The problems are compounded considerably by the more demanding requirements of much present-day software, including timing constraints, distribution, and sophisticated (or miniaturized) user interfaces.

This tutorial introduces a broad framework of portability issues, but concentrates on practical techniques for bringing portability considerations to the software development process. The presentation is addressed both to individual software designers and to those participating in an organized development process.

It is not possible in a paper of this length to provide a detailed and thorough treatment of all of the issues and approaches for software portability. We will offer an introduction designed to increase awareness of the issues to be considered.

2. THE WHAT AND WHY OF PORTABILITY

In this section we will examine what we mean by portability, consider some related concepts, and discuss why porting may be desirable.

2.1 What is Portability?

The concept of software portability has different meanings to different people. To some, software is portable only if the executable files can be run on a new platform without change. Others may feel that a significant amount of restructuring at the source level is still consistent with portability.

The definition we will use for this study leans toward the latter view and includes seven key concepts. This definition originally appeared in Mooney (1990):

> A *software unit* is portable (exhibits portability) across a *class of environments* to the *degree* that the *cost* to *transport and adapt* it to a new environment in the class is less than the *cost* of *redevelopment.*

Let's examine the key concepts in this definition.

- *Software Unit.* Although we will often discuss portability in the context of traditional applications, most ideas may also apply to other types of software units, ranging from components to large software systems.

- *Environment.* This term refers to the complete collection of external elements with which a software unit interacts. These may include other software, operating systems, hardware, remote systems, documents, and people. The term is more general than platform, which usually refers only to the operating system and computer hardware.
- *Class of Environments.* We use this term to emphasize that we seek portability not only to a set of specific environments, which are known a priori, but to all environments meeting some criteria, even those not yet developed.
- *Degree of Portability.* Portability is not a binary attribute. We consider that each software unit has a quantifiable degree of portability to a particular environment or class of environments, based on the cost of porting. Note that the degree of portability is not an absolute; it has meaning only with respect to a specific environment or class.
- *Costs and Benefits.* There are both costs and benefits associated with developing software in a portable manner. These costs and benefits take a variety of forms.
- *Phases of Porting.* We distinguish two major phases of the porting process: *transportation* and *adaptation.* Adaptation includes most of the modifications that need to be made to the original software, including automated retranslation. Transportation refers to the physical movement of the software and associated artifacts, but also includes some low level issues of data representation.
- *Porting vs. Redevelopment.* The alternative to porting software to a new environment is redeveloping it based on the original specifications. We need to compare these two approaches to determine which is more desirable. Porting is not always a good idea!

Note that while we concentrate on the porting of software, there may be other elements for which portability should be considered. These include related software such as libraries and tools, as well as data, documentation, and human experience.

2.2 Why should we Port?

Before we make the effort to make software portable, it is reasonable to ask why this may be a good idea. Here are a few possible reasons:

- There are many hardware and software platforms; it is not only a Windows world.
- Users who move to different environments want familiar software.

- We want easier migration to new system versions and to totally new environments.
- Developers want to spend more time on new development and less on redevelopment.
- More users for the same product means lower software costs.

The advantages of portability may appear differently to those having different roles. Here are some of the key stakeholders in software development and their possible interests in portability:

- *Users* may benefit from portable software because it should be cheaper, and should work in a wider range of environments.
- *Developers* should benefit from portable software because implementations in multiple environments are often desired over the lifetime of a successful product, and these should be easier to develop and easier to maintain.
- *Vendors* should find software portability desirable because ported implementations of the same product for multiple environments should be easier to support, and should increase customer loyalty.
- *Managers* should find advantages in portable software since it is likely to lead to reduced maintenance costs and increased product lifetime, and to simplify product enhancement when multiple implementations exist. However, managers must be convinced that the cost to get the first implementation out the door may not be the only cost that matters!

2.3 Why shouldn't we Port?

Portability is not desirable in all situations. Here are some reasons we may not want to invest in portability:

- Sometimes even a small extra cost or delay in getting the product out the door is not considered tolerable.
- Sometimes even a small reduction in performance or storage efficiency cannot be accepted.
- Sometimes a software unit is so tightly bound to a specialized environment that a change is extremely unlikely.
- Sometimes source files or documentation are unavailable. This may be because developers or vendors are protective of intellectual property rights.

2.4 Levels of Porting

A software unit goes through multiple representations, generally moving from high to low level, between its initial creation and actual execution. Each of these representations may be considered for adaptation, giving rise to multiple levels of porting:

- *Source Portability.* This is the most common level; the software is adapted in its source-level, human-readable form, then recompiled for the new target environment.
- *Binary Portability.* This term refers to porting software directly in its executable form. Usually little adaptation is possible. This is the most convenient situation, but possible only for very similar environments.
- *Intermediate-Level Portability.* In some cases it may be possible to adapt and port a software representation that falls between source and binary.

2.5 Portability Myths

The portability problem is often affected by the "silver bullet" syndrome. A wide variety of innovations have all promised to provide universal portability. These include:

- Standard languages (e.g., FORTRAN, COBOL, Ada, C, C++, Java)
- Universal operating systems (e.g., Unix, MS-DOS, Windows, JavaOS)
- Universal platforms (e.g., IBM-PC, SPARC, JavaVM, .NET)
- Open systems and POSIX
- OOP and distributed object models (e.g., OLE, CORBA)
- Software patterns, architectures, and UML
- The World Wide Web

All of these have helped, but none have provided a complete solution. We will examine both the value and the limitations of these technologies.

3. INTERFACES AND MODELS

A software unit interacts with its environment through a collection of *interfaces*. If we can make these interfaces appear the same across a range of environments, much of the problem of portability has been solved. The first step in controlling these interfaces is to identify and understand them. We will make use of *interface models* to establish a framework for discussion.

A number of interface models have been defined and used by industry and governments. Examples include the U.S. Department of Defense Technical Reference Model, The Open Group Architectural Framework, and the CTRON model. Most of these are quite complex, identifying a large number of interface types classified along multiple dimensions.

A very simple but useful model was developed as part of the POSIX effort to create a framework for *open systems*. Open systems are defined as environments that are largely based on non-proprietary industry standards, and so are more consistent with portability goals. The model defined by the POSIX committees is the Open Systems Environment Reference Model (OSE/RM) (ISO/IEC 1996).

This model is illustrated in Figure 1. It defines two distinct interfaces: the interface between an application and a platform (the Application Program Interface, or API) and the interface between a platform and the external environment (the External Environment Interface, or EEI).

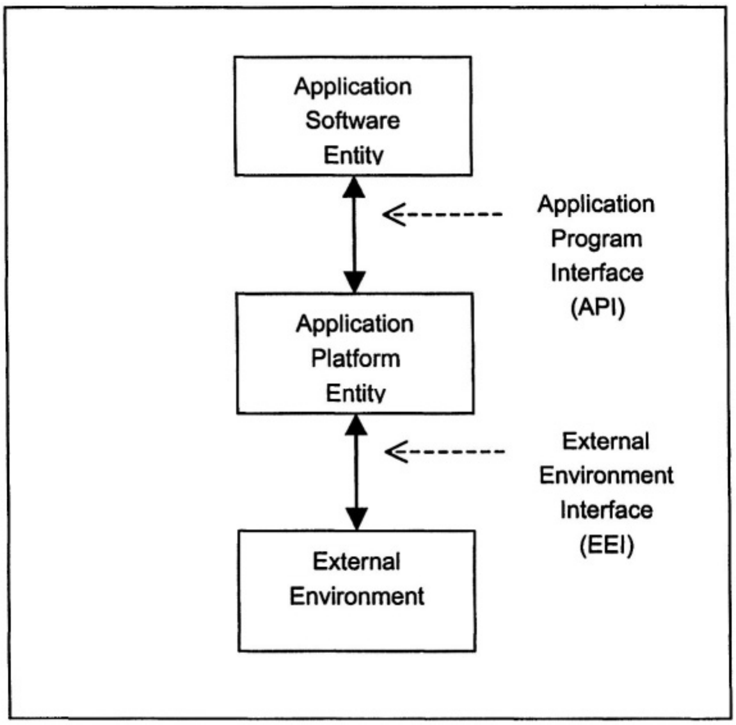
Application Software Entity
Application Program Interface (API)
Application Platform Entity
External Environment Interface (EEI)
External Environment

Figure 1. The POSIX Open Systems Environment Reference Model

The OSE/RM does not provide much detail by itself, but it forms the foundation for many of the other models.

The interface model that will form the basis for our study is the *Static Interface Model (SIM),* originally proposed by the author (Mooney, 1990). This model assumes that the software to be ported is an application program, although other software units would lead to a similar form. The application is in the upper left corner, and the interfaces with which it interacts are shown below and to the right.

The model identifies three *direct interfaces* with which the application is assumed to interact through no (significant) intermediary. These are:

- The *Processor/Memory Interface,* also called the *Architecture Interface,* which handles all operations at the machine instruction level.
- The *Operating System Interface,* responsible for all services provided to an application by the operating system.
- The *Library Interface,* which represents all services provided by external libraries.

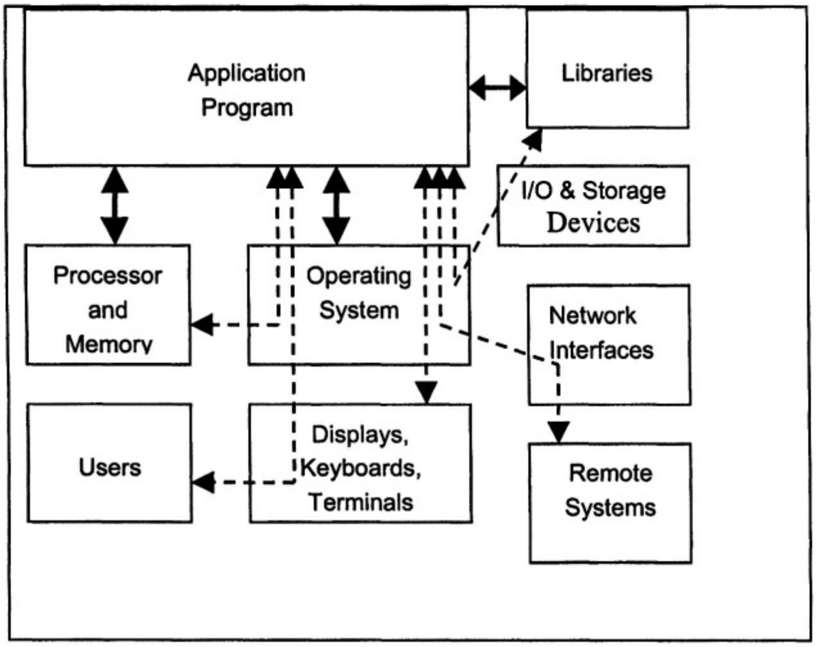

Figure 2. The Static Interface Model

The model further identifies a number of *indirect interfaces,* which are composed of multiple direct interfaces connecting other entities. For example, the user interface involves a chain of interfaces between the application, the operating system, the terminal device, and the user.

Note that the model only identifies interfaces between the application and other entities. Also note that the Operating System Interface could, strictly speaking, be called indirect, since a library is usually involved. However, it is useful to treat this as a direct case.

The value of these models lies in using them to identify and focus on specific interfaces that may be amenable to a particular portability strategy. The SIM provides a useful level of detail for this purpose. As we will see in Section 5, we can identify distinct and useful strategies for each of the direct interfaces of this model.

The interface models considered here are *static models;* they represent a snapshot of the state of a computing system, typically during execution. *Dynamic models* are also used to identify the various representations which may exist for a software unit (and its interfaces) and the translation steps that occur between representations. In the usual case, software ready for porting to a specific environment exists in the form of a *source program* in a common "high-level" programming language. This program may originally have been derived from still higher-level representations. This source program is translated into one or more *intermediate forms,* and a final translation produces the *executable form* to be loaded into memory and executed. Each of these representations offers a different opportunity to bridge interface differences through manual or automated modification.

Other models that may be useful include models of the porting process itself. These are beyond the scope of this paper.

4. THE ROLE OF STANDARDS

A standard is a commonly accepted specification for a procedure or for required characteristics of an object. It is well known that standards can play a crucial role in achieving portability. If a standard can be followed for a particular interface, chances are greatly increased that different environments can be made to look the same. However, standards evolve slowly, so many important interface types are not standardized. Also, standards have many limitations, and only a small number of the vast array of standards in existence can be considered a reliable solution to the problem.

Here we will briefly discuss some issues in the use of standards to help solve the software portability problem. We are interested in *software interface standards:* those that define an interface between multiple entities, at least one of which is software. A very large collection of computer-related standards fits this description.

A software interface standard will aid in the development of portable software if it:

1. provides a clear, complete and unambiguous specification for a significant interface or subset, in a form suitable for the software to be developed;
2. Has implementations that are widely available or may be easily developed for likely target environments.

Unfortunately, many standards fall short of these requirements. They may be expressed in an obscure notation that is hard to understand, or in natural language that is inherently imprecise. There are often many contradictions and omissions. A standard will only become widely implemented if there is already a high demand for it; often a major barrier is the cost of the standard itself.

Standards come into being in three principal ways, each with their own advantages and disadvantages.

- *Formal standards* are developed over an extended time by widely recognized standards organizations such as ISO. They represent a broad and clear consensus, but they may take many years to develop, and are often obsolete by the time they are approved. Some very successful formal standards include the ASCII standard for character codes, the IEEE binary floating point standard, the POSIX standard for the UNIX API, and the C language standard.
- *Defacto standards* are specifications developed by a single organization and followed by others because of that organization's dominance. These are popular by definition but subject to unpredictable change, often for limited commercial interests. Examples include the IBM-PC architecture, the VT-100 terminal model, and the Java language.
- *Consortium standards* are a more recent compromise. They are developed in an open but accelerated process by a reasonably broad-based group, often formed for the specific purpose of maintaining certain types of standards. Example standards in this class include Unicode, OpenGL, and the Single Unix Specification.

Standards will play a critical role in the strategies to be discussed, but they are only a starting point.

5. STRATEGIES FOR PORTABILITY

If software is to be made portable, it must be designed to minimize the effort required to adapt it to new environments. However, despite good portable design, some adaptation will usually be necessary. This section is concerned with identifying strategies that may be used during development to reduce the anticipated level of adaptation, and during porting to carry out the required adaptation most effectively.

5.1 Three Key Principles

There are many approaches and techniques for achieving greater portability for a given software unit. These techniques may be effectively guided by three **key principles.**

5.1.1 Control the Interfaces

As noted in previous sections, the major problems of portability can be overcome by establishing a common set of interfaces between a software entity and all elements of its environment. This set may include many different interfaces of various types and levels.

These interfaces may take many forms: a programming language, an API for system services, a set of control codes for an output device.

Commonality of the interfaces requires that each interface be made to look the same from the viewpoint of the software being developed, in spite of variations in the components on the other side. This goal may be achieved by many different strategies. Some of these strategies may successfully establish a common interface during initial development, while others will require further effort during porting.

5.1.2 Isolate Dependencies

In a realistic software project there will be elements that must be dependent on their environment, because variations are too great or critical to be hidden by a single common interface. These elements must be confined to a small portion of the software, since this is the portion that may require modification during porting.

For example, software that manages memory dynamically in a specialized way may be dependent on the underlying memory model of the architecture or operating system; graphics algorithms may depend on the output models supported; high-performance parallel algorithms may need to vary depending on the architectural class of the machine.

Notice that this is also an interface issue, since the dependent portions of the software need to be isolated behind a limited set of interfaces.

5.1.3 Think Portable

This final principle is simply an admonition to be conscious of the portability goal during all design decisions and development activities. Many portability problems arise not because there was no way to avoid them, but because portability wasn't considered when the choice was made.

5.2 Classifying the Strategies

The strategies to be studied are concerned with controlling particular interfaces. They can be identified and classified by considering the various interface models of Section 2. Static models such as the SIM or the OSE/RM allow us to identify the principal interfaces with which a typical application interacts. We have seen that the primary low-level interfaces can be classified as *architecture* interfaces (including processor, memory, and direct I/O), *operating system* interfaces (including APIs), and *library* interfaces (including support packages).

We are most interested in achieving commonality for these low-level interfaces. These interfaces in turn mediate access to higher-level interfaces such as the user interface, file systems, network resources, and numerous domain-specific abstractions. Controlling these interfaces can be viewed as special cases of controlling the underlying interfaces to the architecture, operating system, and libraries.

The first and most essential strategy for portable software development is the use of a suitable programming language. Portable programming in an appropriate language will generally provide a common model for most (but not all) of the elements of each of the three main interface classes. The remaining elements must be dealt with by considering the interfaces more directly.

There are thus four main classes of strategies that are most important to consider:

1. Language-based strategies
2. Library strategies
3. Operating system strategies
4. Architecture strategies

Regardless of the specific interface or representation we are dealing with, all strategies for achieving portability at that interface can be grouped into three main types. We examine these types in the next subsection. We will then discuss strategies of each type that apply to each of the four classes.

5.3 Three Types of Strategies

The object of an interface strategy is to enable a software unit at one side of the interface to adapt to multiple environments at the other side. If we can arrange for the software unit to have the same predictable view of the interface for all environments, the problem has been solved. This can occur if there is a well-known common form that most environments will follow, or if the element on the other side of the interface can be kept the same in each environment.

If there is no common model known when the software unit is designed, then interface differences are likely to exist when porting is to be done. In this case, a translation may be possible to avoid more extensive modifications to the software.

These considerations lead us to identify three fundamental types of strategies. All of the more specific strategies we will consider can be placed into one (or occasionally more) of these types.

5.3.1 Standardize the Interface

If an existing standard can be identified which meets the needs of the software unit and is likely to be supported by most target environments, the software can be designed to follow this standard. For example, if most environments provide a C compiler, which adequately implements the C standard, it may be advantageous to write our programs in standard C.

This strategy must be followed in the initial development of the software. It relies on the expectation that the standard will truly be supported in (nearly) identical form in the target environments. Figure 5 depicts the general form of this strategy.

5.3.2 Port the Other Side

If the component on the other side of the interface can be ported or reimplemented in each target environment, it will consistently present the same interface. For example, porting a library of scientific subroutines ensures that they will be available consistently in the same form.

This strategy may be chosen during initial development, or selected for a specific implementation. Note that in this case we are actually "extending the boundaries" of the ported software, and trading some interfaces for others. The interfaces between the extra ported components and their environment which is not ported, must be handled by other strategies.

5.3.3 Translate the Interface

If the interfaces do not match, elements on one side may be converted into the equivalent elements on the other. This may be done by an overall translation process when porting, or by providing extra software that interprets one interface in terms of the other during execution.

An example of the first variation is the usual compiling process. The common representation for the architecture interface of a program (in the source language) is translated to the specific architecture of the target. The second variation is illustrated by a library that converts one set of graphics functions to a different set provided in the target environment.

This strategy may be planned during development but must be carried out in the context of a specific implementation.

There is one alternative to all these approaches; that is to redesign the software to fit the interface of the target environment. This is not a portability strategy, but an alternative to porting. Sometimes this is the best alternative, and we must know how to make a choice. This issue will be discussed later.

We now are prepared for a look at the various strategies associated with each of the main classes: language, library, operating system, and architecture.

5.4 Language Based Strategies

Effective approaches to portable design start with the choice and disciplined use of a suitable programming language. If the language allows a single expression of a program to be understood identically in all target environments, then portability will be achieved.

In practice, it is very often possible to express many of a program's requirements in a common language, but not all of them. If the language includes no representation for certain concepts, they must be handled by other strategy classes.

Sometimes the language in which the program is currently expressed is not supported for the target environment. In this case it becomes necessary to translate the software from one language to another.

The source language representation of a software unit is the most convenient starting point for both manual adaptation (e.g. editing) and automated adaptation (e.g. compiling) to prepare it for use in a given target environment. Therefore language-based strategies are the single most essential class in our collection.

Language strategies for portability may be classified according to the three types identified in the previous section: standardize, port, translate.

5.4.1 Standard Languages

Programming languages were among the first types of computer-related specifications to be formally standardized. Today formal standards exist for over a dozen popular general-purpose languages, including FORTRAN, COBOL, Ada, Pascal, C, and C++ (note that standardization for Java is not yet achieved).

Writing a program in a standard language is an essential step in achieving portability. However, it is only a starting point. The language must be one that is actually available in most expected target environments. No standard is clear, complete and unambiguous in every detail, so the programmer must follow a discipline (think portable!) that avoids use of language features which may have differing interpretations. No language covers all of the facilities and resources that particular programs may require, so portability in some areas must be achieved by other means.

Effective use of standard languages is crucial to achieving portability. Each of the most widely-used languages presents a somewhat different set of opportunities and problems for effective portable programming. For example, C does not fully define the range of integers; many Java features continue to vary as the language evolves. Standard language strategies, and the issues raised by specific languages, are often the subject of books and are beyond the scope of this paper.

5.4.2 Porting the Compiler

One of the potential problems of the use of standard languages is the fact that different compilers may use different interpretations in areas where the standard is not completely clear. If the same compiler is used for each target environment, then its interpretation of a software unit will not vary, even if it is non-standard!

To exploit this situation, we may choose to write a program for a specific compiler, then "port the compiler" to each new environment. Porting the compiler, of course, may be a daunting task, but it needs to be done only once to make all software for that compiler usable on the new target. Thus the payoff may be great, and if we are lucky, someone has already done it for us.

It is actually misleading to speak of *porting* the compiler; the essential requirement is to *retarget* the compiler. The compiler's "back end" must be modified to generate code for the new machine. Many compilers are designed to make this type of adaptation relatively easy. The retargeted compiler *may* also be ported to the new environment, but it does not actually matter on which system it runs.

In some cases the compiler is a commercial product, not designed to allow adaptation by its users. In this case we must rely on the vendor to do the retargeting, or else this strategy is unavailable. Many open source compilers, such as the GNU compilers, are designed for easy retargeting.

5.4.3 Language Translation

A compiler translates software from a human-oriented language to a language suitable for execution by machine. It is also possible to translate programs from one human-oriented language to another. If we are faced with a program written in a language for which we have no compiler for the target, and no prospect of obtaining one, then "source-to-source translation" may be the best porting strategy available.

Translating software from Pascal to FORTRAN or C to Java is considerably more challenging than compiling, though not as difficult as natural language translation. Several tools are available which perform reasonable translations among selected languages. Usually these tools can do only an imperfect job; translation must be regarded as a semi-automated process, in which a significant amount of manual effort may be required.

Translation can also be used as a development strategy, when compiler variations are anticipated. Software may be originally written in a "higher-level" language that can be translated into multiple languages or, more likely, language dialects, using a preprocessor. This is more likely to be a strategy that can be fully automated.

5.5 Library Strategies

No programming language directly defines all of the resources and facilities required by a realistic program. Facilities that are neither defined within the language nor implemented by the program itself must be accessed explicitly through the environment. This access may take the form of procedure or function calls, method invocations, messages, program-generated statements or commands, etc. Whatever the mechanism, the aim is to obtain services or information from software and physical resources available in the environment.

These resources are organized into packages, collections, or subsystems that we may view uniformly as *libraries.* Examples may include language-based libraries such as C standard functions, scientific computation libraries, graphic libraries, domain-specific classes and templates, mail server processes, network interfaces, or database management systems.

These libraries provide a class of interfaces, and programs that rely on them will be most portable if they are able to access these facilities in a common form. When this is not possible, adaptation will be required.

We must assume, of course, that the target systems are capable of supporting the services and facilities to which the libraries provide access. No portability strategy can enable a program that plays music to run on a system that has no hardware support for sound!

Once again we can identify three classes of library strategies according to our three principal types: standardize, port, translate. We will overview these strategies in the following subsections

5.5.1 Standard Libraries

Many types of library facilities are defined by formal or defacto standards. This group is led by libraries that are incorporated into specific language standards, such as the C standard function library, Ada standard packages, standard procedures and functions of Pascal, standard templates for C++, etc. Software written in these languages should use the standard facilities as far as possible, taking care to distinguish what is actually standard and what has been added by a particular language implementation.

Additional standard library facilities are not bound to a specific language but are widely implemented in many environments. This is especially likely for libraries providing services of broad usefulness. Some examples here include GKS libraries for low-level graphics functions, MPI for message passing, CORBA for distributed object access, or SQL for database access. Portable software may rely on such libraries if they are expected to be widely available, but must make use of an alternate strategy when they are not.

If the absence of a library supporting an important standard is a major drawback for the target environment, it may be worthwhile to consider implementing such a library. This is likely to be a major effort but could significantly improve the target environment as well as aiding the immediate porting project.

5.5.2 Portable Libraries

Instead of relying on the wide availability of library implementations which conform to a common standard, we may rely on a single implementation, not necessarily standardized (although it creates a de facto standard), which is or can be ported to a wide variety of environments.

A few examples include the mathematical libraries of the Numerical Algorithms Group (NAG) and the linear algebra library LINPACK for high-performance computing.

If the library is non-proprietary and its source code is available, then we may rely on porting the library ourselves when faced with an environment which does not support it. Again, this may be a large task, perhaps larger than porting the rest of the software, but the benefits may apply to many projects. If the library is proprietary, the only hope is to appeal to the vendor.

5.5.3 Interface Translation

In some cases the target environment will provide a library with the necessary functionality, but not in the expected form. In this case an additional library must be created to "bridge" the difference. This library becomes a part of the porting effort, and must present the required services in the form expected by the program, using the facilities provided by the native library. The effort to create such a bridge library can range from minimal to extensive, depending on the extent of the difference between the two interfaces. Once created it may provide benefits for multiple projects, as though the library itself had been ported.

5.6 Operating System Strategies

Many of the services which a program accesses from its environment are provided or mediated by the operating system (OS). As can be seen from the Static Interface Model, the OS may directly provide services such as process management, memory management, file access, timing services, security services, etc. It is also a key mediator in the user interface, and in interfaces to networks and I/O devices.

Some of these services, such as simple file access, may be defined directly by the programming language. Others may be defined by standard libraries such as the C library. However, a variety of services may be obtainable only by direct request to the OS. This is especially true of many newly important services such as thread management, multimedia, or

Internet access for which higher-level standards are still evolving. The OS interface is thus a key issue for portability in a large number of programs.

Since portability is most commonly considered as a proper expectation of application programs (more than specialized system programs), the operating system interface is referred to as the Application Program Interface, or **API.** It would perhaps be more accurate to speak of the "OSAPI", identifying the entity on both sides of the interface, but this term has not caught on.

Most OSs support a number of programming languages and must make their services available in a uniform language-independent form. This creates the need for two representations of the API: a language-independent form, as presented by the OS, and a representation in terms of the particular programming language used, called a *language binding.* A small library is needed to provide services in the form specified by the language binding and convert them to the form provided by the underlying operating system. In this discussion we will ignore this extra layer and focus our strategies on the language-independent API.

As before, we can consider three main classes of strategies: standardize, port, or translate.

5.6.1 Standard APIs

As recently as the early 1980s there was no such thing as a "standard" API. Each specific OS presented its services in its own way. Even when the services were equivalent, there was no effort to represent them by a common model.

A great deal of variation often existed (and still does) even within versions of the "same" OS. Many subtle differences in UNIX APIs have made portability a problem even from one UNIX to another. This created a strong motivation for the POSIX project. Similar problems across versions of proprietary OSs led vendors to create their own internal standards.

Today there are a variety of established and developing standards for APIs, both formal and defacto. Important examples include the POSIX system interface for "UNIX-like" environments, and the Win-32 API for Microsoft Windows systems.

Unfortunately, there are few standard APIs which span distinctly different types of operating systems, such as UNIX *and* Windows *and* z/OS *and* Palm OS, etc. In some cases standard APIs can be implemented (less naturally and efficiently, of course) by libraries on top of a different type of OS. The POSIX API, in particular, has been implemented for a wide variety of environments which are not actually similar to UNIX.

If the set of target systems anticipated for porting, or a significant subset, is covered by a standard API, then that standard should probably be followed. If not, we must continue with other strategies.

5.6.2 Porting the Operating System

The idea of porting an operating system may seem completely unreasonable. The purpose of the OS is to manage resources, including hardware resources, so its design must be tied closely to the machine architecture. Because of the need for efficiency and compactness, many OSs have been written in assembly language. The OS is designed from the ground up to suit a particular machine; moving it to another just doesn't make sense.

In spite of this, a number of operating systems have been successfully ported, and some have been designed to be portable. A few early research systems in this category include OS/6, MUSS, and THOTH. These systems were designed in ways that allowed hardware access but remained somewhat architecture-independent, often using the generic architecture strategy discussed below. They were programmed in medium-level "system implementation languages" such as BCPL. As a result, they were successfully ported to multiple hardware environments.

The quintessential example of a portable OS today is UNIX. UNIX has been ported to, or reimplemented for, almost every known hardware environment suited for general-purpose computing. UNIX and all of its related programs are written in C, so porting is greatly facilitated by the creation of a C compiler. The various implementations represent many slight variations, but they all share a common core of UNIX concepts.

Porting a compiler is a project that is likely to have high costs but also high benefits. This is true to a much greater degree for OS porting. The effort required may be enormous, but the result is the ability to support a whole new collection of software in the new environment.

Unfortunately, though, most environments can only run one OS at a time for all users. Porting a new OS means removing the old one. This will have a very strong impact on users; we do not recommend that you change the OS on a system that is used by many people unless there is broad agreement that this is a good idea!

5.6.3 Interface Translation

If it is not possible to ensure that the API for which the program is designed will be supported in the target environment, then a translation library may be necessary. This library can range from trivial to highly complex, depending on the differences in the resource models expected by the program and those supported on the target platform. For example, the Windows interface is implemented on UNIX systems, and POSIX libraries are available for environments as diverse as OpenVMS and MVS.

5.7 Architecture Strategies

The first and most fundamental of the three main direct interfaces is the interface to the machine architecture. At its lowest level this is manifest as a set of machine instructions together with other hardware resources such as registers and a memory model.

It is generally expected that the programming language will hide the details of the architecture; this is after all its primary purpose. However, there are often architectural details that are not encapsulated in programming languages, such as the precision of floating point operations, or the organization of special memory models. Some languages include structures biased toward a particular architectural model, such as C with its orientation toward Digital PDP-11 and VAX architectures. Even if the language can hide the architecture completely, providing one or a few common architecture interfaces can greatly simplify compiler design. In the extreme, identical architectures across platforms can eliminate the need for recompilation, allowing for binary portability.

For all of these reasons, we may want to consider strategies that provide greater standardization of the lower-level architectural interface.

As usual we consider the three principal strategies of standardization, porting, and translation. Here we run into a problem. It is clear what is meant by standardizing an architecture, but how do we "port the machine?" Architecture translation may also seem impractical, but there are two different types of strategies that fit this description.

In the end we can identify three distinct types of strategies at the architecture level. However, their relation to the three primary categories is a little more complicated.

5.7.1 Standard Architectures

The straightforward concept of a standard architecture is that a large collection of computers should have the same "machine-level" architecture (i.e., instruction set, registers, data types, memory model, etc.) even though they are produced by various companies. The clearest example of this concept is the de facto standard IBM-PC architecture, which is copied precisely by numerous "clones" made by companies other than IBM. Because the architecture is identical (except perhaps for a few details related only to maintenance) all of the same software can be run. There have been clones of systems as diverse as the IBM S/360, the Intel 8080 processor chip, and the Macintosh.

A few formal architecture standards have also been developed. Japan's TRON project defined a microprocessor architecture which has actually been implemented by over a dozen companies. The Sun SPARC architecture has been approved as a formal standard, although it is not yet implemented outside of Sun.

Today few users care greatly about the architecture of their computers, as long as they run the desired software and achieve good performance. However, companies that sell computers must be able to point to unique advantages of their product, which necessarily means differences. Makers of IBM clones try to meet this need by higher performance, lower cost, or better I/O devices. Other implementors may add extended features such as better memory management, but programs that rely on these features lose the benefits of portability.

Occasionally success can be achieved by standardizing a limited part of the architecture. The IEEE binary floating point standard is now almost universally used in floating point hardware, and has greatly relieved a major portability problem for numerical software.

5.7.2 Generic Architectures

As an alternative to a standard architecture that is to be implemented by computing hardware directly, a common architecture may be defined which is intended to be "easily translated" into the physical architecture of a variety of computers. A common compiler can produce code for the generic architecture, and a machine-dependent translation converts this code into native instructions for each specific system. This may be an attractive approach if the translation step is simple and if the final performance of the software is not greatly reduced.

The generic representation of the program may be viewed as a low-level intermediate form in the translation process. It may be translated to native machine code before execution, or it may be interpreted "on the fly." Microprogrammed architectures may have the option of interpreting the generic machine code by a special microprogram. This option has become less common since the advent of RISC processors, which are usually not microprogrammed.

5.7.3 Binary Translation

In previous discussions we have noted that significant adaptation is generally not practical for a program in “binary” (executable) form. In spite of this, there are times when it becomes essential to convert software already compiled for one architecture into a form that can be used on a very different architecture. Two well-known examples of this approach have arisen as major computer vendors migrated to newer, RISC-class architectures:

- The change in Digital systems from the VAX to the Alpha
- The change in Macintosh systems from the 68000 to the PowerPC

In these situations a great deal of application software, already in executable form for the older environments, must be made to work in the newer one. To meet this need, strategies have evolved for effective binary translation as a transitional strategy. Typically, this approach uses a combination of translation before execution where possible, and run-time emulation otherwise. The success of the approach may rely on strong assumptions, such as the assumption that the program being translated is a well-behaved client of a particular operating system.

6. THE SOFTWARE DEVELOPMENT PROCESS

The previous section has identified a wide range of strategies for increasing portability by controlling the interfaces of a software unit. To put these strategies to work we must see how portability concerns fit into the software development process.

The discussion in this section is focused on incorporating portability in a large-scale software development process. However, most of the recommendations may be applied to small projects as well.

6.1.1 The Software Lifecycle

A number of models of the software lifecycle are used both to understand the lifecycle and to guide the overall development strategy. These are surveyed in many software engineering texts, such as Sommerville (2000). Most widely known is the waterfall model, in which activities progress more or less sequentially through specification, design, implementation, and maintenance. Recently popular alternatives include rapid prototyping and the spiral model, with frequent iterations of the principal activities. Testing (and debugging) and documentation may be viewed as distinct activities, but are usually expected to be ongoing throughout the process.

Each of the principal activities of the lifecycle is associated with some distinct portability issues. However, the sequencing and interleaving of these activities, which distinguishes the models, does not substantially affect these issues. Thus our discussion is applicable across the usual models, but will focus primarily on the individual activities.

6.1.2 Specification

The purpose of a specification is to identify the functionality and other properties expected in the software to be developed. There are many proposed structures for such a specification, ranging from informal to fully formal, mathematical notations. Formal notations in current use express the *functional* requirements of a software product, but are not designed to express non-functional requirements such as reliability, performance, or portability. If such requirements exist they must be expressed by less formal means.

We offer three guidelines for the specification activity to maximize portability, regardless of the form chosen for the specifications:

1. *Avoid portability barriers.* It is important that a specification should not contain statements and phrases that arbitrarily restrict the target environment, unless those restrictions are conscious and intentional. For example, “the program shall prompt the user for an integer value” is better than “the program shall display a 2 by 3 inch text box in the center of the screen”.
2. *State constraints explicitly.* It is important to know, for example, if the application must process a database with one million records or must maintain a timing accuracy of 5 milliseconds. This can be used in part to determine which target environments are reasonable.
3. *Identify target classes of environments.* After consideration of the constraints and necessary portability barriers, the specification should identify the broadest possible class of target environments that may make sense as candidates for future porting.
4. *Specify portability goals explicitly.* If the form permits, it is desirable to identify portability as a goal, and the tradeoffs that can be made to achieve it. An example might be “the program shall be developed to be easily ported to any interactive workstation environment, supporting at least thousands of colors, provided that development costs do not increase by more than 10% and performance does not decrease by more than 2% compared to non-portable development.”

6.1.3 Design

Design is the heart of software development. Here our understanding of what the software is to do, embodied in the specification, directs the development of a software architecture to meet these requirements. At this stage the developer must select the approach to portability, and choose appropriate strategies.

A large software project may require several levels of design, from the overall system architecture to the algorithms and data structures of individual modules. A systematic design method may be used, such as Structured Design, SADT, JSD, OOD, etc. The various methods have widely differing philosophies, and may lead to very different designs. However, they share a common objective: to identify a collection of elements (procedures, data structures, objects, etc.) to be used in implementing the software, and to define a suitable partitioning of these elements into modules.

The resulting design (perhaps at various levels) has the form of a collection of interacting modules that communicate through interfaces. It is well understood that clear and careful interface design is a crucial element of good software design.

Ideally, a software design is independent of any implementation and so is perfectly portable by definition. In practice, the choice of design will have a major impact on portability.

Portability issues in design are focused on partitioning. We identify four guidelines:

1. *Choose a suitable methodology.* Some design methods may be more favorable to portable design. For example, object-oriented design provides a natural framework for encapsulating external resources.
2. *Identify external interfaces.* A systematic review of the functionality required by the software unit from its environment should lead to a catalog of external interfaces to be controlled.
3. *Identify and design to suitable standards.* Standards should be identified that address interfaces in the catalog, and that are likely to be supported in the target environments. The design should organize these interfaces, as far as possible, in accordance with these standards.
4. *Isolate system-dependent interfaces.* By considering the interfaces with no clear standard or other obvious strategy, and the intended class of target environments for porting, the developer can make reasonable predictions that these interfaces will need system-specific adaptation. These interfaces then become strong candidates for isolation.

6.1.4 Implementation

Implementation is concerned with transforming a design into a working software product. If good design practice has been followed, the design in most cases should not be platform-specific, even if it is not explicitly portable.

In most cases, the implementation targets one specific environment. Occasionally, versions for multiple environments are implemented simultaneously. During portable development, it is also possible to envision an implementation that has no specific target, but is ready for porting to many environments.

Developers who strive for portability most frequently concentrate their attention on the implementation phase, so the issues here are fairly well understood. We offer three guidelines:

1. *Choose a portable language.* If the language or languages to be used were not determined by the design phase, thy must be chosen now. Many factors go into good language choice, including programmer experience, availability of tools, suitability for the application domain, etc. An additional factor should be considered: is the language well standardized, widely implemented, and thus a good choice for portability?
2. *Follow a portability discipline.* It is not enough to select a good language; the language should be used in a disciplined way. Every language has features that are likely to be portability problems. Any compiler features that check for portability should be enabled.
3. *Understand and follow the standards.* The design phase and language choice have identified standards for use. The programmer must study and understand those standards, to be sure that the implementation actually matches what the standard says, and what will be expected on the other side of the interface.

6.1.5 Testing and Debugging

Testing is an essential activity for any type of software development. Many projects also make use of formal verification, to demonstrate a high likelihood of correctness by logical reasoning. However, this does not remove the need for testing.

The goal of testing is to verify correct behavior by observation in a suitable collection of specific test cases. It is not possible to test all cases, but there are well-known techniques to generate sets of test cases that can cover most expected situations and lead to a reasonably high confidence level in the correct operation of the software.

Guidelines for the testing activity are:

1. *Develop a reusable test plan.* A written test plan is always important. For portable software the test plan should be designed to be reused for new ported implementations. Based on design choices and experience to date, the plan should cleanly separate tests of system-dependent modules from tests of the modules that are common to all (or many) implementations. It should be anticipated that after porting, the same tests will be applicable to the common modules (and should produce the same results!).
2. *Document and learn from errors.* A record should be kept, as usual, of all errors found, and the debugging strategies used to correct them. Again these records should be divided between common and system-dependent parts. Errors that have been corrected in common modules should not usually recur after a port.
3. *Don't ignore compiler warnings.* Warnings from a compiler are often taken lightly, since they generally indicate a construct that is questionable but not considered a fatal error. If the program seems to work, the warning may be ignored. It is highly likely, though, that the problem identified by warnings means an increased likelihood of failure when running in a different environment. An unitialized variable may be harmless in one implementation, but cause incorrect behavior in the next.
4. *Test portability itself.* If portability has been identified as an intended attribute in the specifications, it is necessary to test if this goalhas been achieved. This may require the use of portability metrics, discussed briefly below.

6.1.6 Documentation

Many types of documents are associated with a well-managed software process. Portability will have an impact on the documentation activity as well as the other development phases.

Portability guidelines for documentation are:

1. *Develop portable documentation.* The documentation phase offers an opportunity to take advantage of the commonality of portions of a software unit across multiple implementations. Technical documentation can be separated between the common part and the system-specific part. The common part will not change for new implementations. The same is true for user documentation, but with a caution: Users should be

presented with documentation that is specific for their environment, and avoids references to alternate environments.

2. *Document the porting process.* The technical documentation should explain the aspects of the design that were provided for the sake of portability, and provide instructions for those who will actually port the software.

6.1.7 Maintenance

The maintenance phase is the payoff for portable development. Each requirement to produce an implementation in a new environment should be greatly facilitated by the efforts to achieve portability during original development. Other maintenance activities, such as error correction and feature enhancement, will not be impeded by portable design and may possibly be helped.

The only complicating factor is the need to maintain multiple versions. Where possible, clearly, common code should be maintained via a single source, if the versions are under the control of a common maintainer. Issues of multiversion maintenance are widely discussed in the literature and will not be repeated here.

6.1.8 Measuring Success

An important management concern is to demonstrate with facts and figures that making software portable is a good idea, as well as to show that this goal has been achieved. Metrics are required to evaluate portability in this way. One useful metric is the degree of portability, defined as:

$$\mathbf{DP = 1 - (cost\ of\ porting) / (cost\ of\ redevelopment)}$$

This metric may be estimated before beginning a porting project by comparing the estimated cost of porting with that of redevelopment, using standard cost estimation techniques. Note that the elements of the cost must be considered in the context of a specific target environment or class of environments. Degree of portability has no meaning without this context.

The main difference between the two cost alternatives is that porting begins with adaptation, while redevelopment begins with redesign and reimplementation.

If DP < 0, porting is more costly and should be avoided. If DP >= 0, then it will range between 0 and 1. In this case porting is the preferred solution, and the vale of DP is proportional to the expected cost of porting.

This metric may be estimated before initial development, to determine if portable development is worthwhile. It may also be estimated after initial development to characterize the portability that has been achieved.

7. OTHER ISSUES

This section briefly overviews two additional areas of concern that need to be considered for a more complete understanding of the software portability problem.

7.1.1 Transportation and Data Portability

We have identified two major phases of porting: adaptation and transportation. So far we have focused on adaptation issues. Transportation addresses problems of physical movement of software and associated artifacts, whether by means of transportable media or a network. This phase also must contend with a number of problems of data representation.

Transportation issues can be identified in several categories:

- *Media Compatibility.* There may be no common media format between the source and target environments. Even if both accept floppy disks, for example, there are different sizes, densities, and formats. The physical drive must accept the media from a different system, and it must further understand the information that is on it.
- *Network compatibility.* A similar problem can occur if two systems are connected by a network. In this case differences in network protocols can present effective communication.
- *Naming and file systems.* The problem is more complex if the data to be transported represents a set of files for which names and relationships must be maintained. There are dozens of file system types, and no standard format for data transport. Each environment understands only a limited number of "foreign" file systems, and may have different rules about file naming.
- *Data compatibility.* Low level data issues may occur due to differences in character codes supported, different strategies for indicating line endings, different rules on byte order for multibyte integers, etc. The problems are more complex if data is to be transported in formats such as floating point or structures or arrays.

7.1.2 Cultural Adaptation

It is not always desirable that ported software behave in exactly the same way as the original. There are many reasons why different behavior may be wanted. Many though not all of these are related to the user interface. We define the process of meeting the varying behavioral needs of each environment as *cultural adaptation.* This may take several forms:

1. *Adapting to user experience.* The type of user interface preferred by a travel agent for booking airline flights is very different than that preferred by a casual user. In the same way, a user experienced with Macintosh systems will not want a new application to behave like a Windows program, *unless* they are even more familiar with the Windows version of that application.
2. *Adapting to human cultures.* This involves many processes identified under the heading of internationalization and localization. It may be necessary to translate all text, including labels, etc., to different languages with very different structure from the original. In addition, issues as diverse as the sort order for databases or the use of color to convey certain meanings must be reconsidered.
3. *Adapting to environment capabilities and constraints.* One example of this is the need to use different computational algorithms for different high-performance parallel computers. Another is the problem of *economic portability* (Murray-Lasso 1990). Many users in schools, non-profit agencies, or less developed countries continue to work with computers much less capable than today's state-of-the-art. To avoid leaving these users behind, software should be adaptable to these older environments, even if its performance and functionality are reduced.

8. CONCLUSION

This paper has surveyed a broad range of issues to be considered in the development of portable software. A range of strategies has been proposed for addressing the problems. We have also examined ways in which portability may be incorporated into the software development process.

We have only been able, however, to explore the surface of the problem. Some of the issues that have not been discussed are:

- Tools for developing portable software
- Analysis of costs and benefits
- The porting process itself
- Portability for special domains, such as parallel and real-time software
- The relationship between portability and reuse

Portability vs. reuse is discussed by Mooney (1995). Most of these issues are examined in a course taught by the author at West Virginia University (Mooney 1992). More information is available on the course website (Mooney 2004).

REFERENCES

Blackham, G. Building software for portability. 1988, *Dr. Dobb's Journal,* **13**(12):18-26.

Brown, P.J.,(ed), 1977, *Software Portability,* Cambridge University Press, Cambridge, U.K.

Dahlstrand, I., 1984, *Software Portability and Standards,* Ellis Horwood, Chichester, U.K.

Deshpande, G., Pearse, T., and Omar, P., 1997, Software portability annotated bibliography, *ACM SIGPLAN Not.,* **32**(2):45-53

Henderson, J., 1988, *Software Portability,* Gower Technical Press., Aldershot, U.K..

ISO/IEC, 1996, *Guide to the POSIX Open System Environment,* TR 14252

Lecarme, O., Gart, M. P., and Gart, M., 1989, *Software Portability With Microcomputer Issues,* Expanded Edition, McGraw-Hill, New York.

Mooney, J. D., 1990, Strategies for supporting application portability, *IEEE Computer,* **23**(11):59-70.

Mooney, J. D. 1992, A course in software portability, in *Proc, 23rd SIGCSE Tech. Symp.,* ACM Press, New York, pp. 163-167.

Mooney, J. D., 1995, Portability and reusability: common issues and differences, *Proc. ACM Comp. Sci. Conf.,* ACM Press, New York, pp. 150-156.

Mooney, J. D.. 2004, CS 533 (Developing Portable Software) course website, West Virginia University, Morgantown, WV, http://csee.wvu.edu/~jdm/classes/cs533

Murray-Lasso, M., 1990, Cultural and social constraints on portability, *ISTE J. of Research on Computing in Education,* **23**(2):253-271.

Poole, P .C. and Waite, W. M., 1975, Portability and adaptability, in *Software Engineering: An Advanced Course.* F. L. Bauer, ed., Springer-Verlag, Berlin.

Ross, M., 1994, Portability by design, *Dr. Dobb's Journal,* **19**(4):41 ff.

Sommerville, I. 2000, *Software Engineering,* 6th ed, Addison-Wesley, Reading, Mass.

Tanenbaum, A. S., Klint, P., and Bohm, W., 1978, Guidelines for software portability, *Software -- Practice and Experience,* **8**(6):681-698.

Wallis, P. J. L., 1982, *Portable Programming,* John Wiley & Sons, New York.