SUPPORTING ARTICLE

The Myth of Application Source-Code Conformance

Stephen R. Walli
SOFTWAY SYSTEMS, INC., KITCHENER, ONTARIO

■ **Branding and certifying applications source-code conformance to POSIX standards and The Open Group specifications demands attention every few years. The general premise is that an organization purchasing POSIX.1-certified or XPG4-branded *systems* should be able to purchase corresponding *applications*. This reasoning defeats the purpose of source-code portability specifications, and has no economic foundation. It confuses applications users with applications developers, and provides information to purchasing management that is grossly out of context.**

There have been at least two cases in the past two years within the POSIX community where people outside the standards development group have claimed that without POSIX-conforming applications to run on their POSIX-conforming implementations, they see no value in these specifications. The Open Group is now demanding a similar program.

If such programs for application-conformance branding are implemented, I believe they will fail due to poor economic and technical foundations, and will confuse customers making standards-based purchases. To support this argument, I first describe source-code portability and porting and their place in current applications development practices. I examine source-code portability standards, discuss how implementation conformance is defined, and what conformance certification entails. I then apply this discussion to the problem of application source-code conformance and certification.

We define the following terms:

—Portability is an attribute of application source code that refers to its ability to be rebuilt (that is, recompiled and relinked into an executable binary form) on another machine. The more portable an application's source code is, the easier it is to rebuild it on a new machine.
—Porting source code is the act of taking the application source files to another machine and rebuilding the application to run on the new machine. This may require changes to the source which, while they would make that source code less "portable," are still done to accomplish the rebuilding of the application. Porting source code is a software engineering activity that shares many of the same concerns as configuration management and software maintenance. As a software engineering activity, source-code porting is also prone to the same problems with respect to process measurement and project comparison as other software engineering problems.

An adage in the software porting community says that there's no such thing as portable code, only code

There's no such thing as portable code, only code that's been ported.

that's been ported. This saying illustrates the difference between a code's perceived portability and the work involved in porting it. As a program is ported from machine to machine, one might suppose that its source code should become more portable, as problems are corrected with each new porting. This is not necessarily true. The demands of the project, of which source-code portability is only one aspect, may change the source code and make it less portable. If backwards compatibility or preservation of the source across multiple architectures concurrently is not required, porting can be speeded by reducing the time spent testing/fixing the source on other platforms; this may reduce source-code portability.

It is also important to understand how applications source-code portability has become more crucial as MIS technology has evolved. Computer hardware costs have fallen steadily over the past 20 years; this has caused a number of changes in the ways organizations use their data and structure their use of computers. Application strategies evolve and change, whether by careful engineering or in response to more urgent tactical needs.

Minicomputers changed the perception of mainframes; then personal computers (PCs) appeared, and the need arose to share data on these disparate systems. End users began to develop their own applications through simple databases and spread-sheets. Departmental servers have appeared, and client-server applications development has become common.

Applications source code has become one of the most expensive items in an organization's computing budget. It can also cause a bottleneck as the information needs of the organization increase. Applications are now considered "assets." As computing platforms change, companies need to move business-critical applications to the new platforms. Many applications have been moved or rewritten to a new hardware platform more than once in the past ten years. Applications source-code portability is vital to preserve a company's investment in applications development.

Programming specifications such as the ISO C-language standard and the ISO POSIX family of standards have been developed to support applications source-code portability. These standards describe programmatic interfaces and define the model to which a programmer writes source code, while a systems developer implements the underlying semantics. This is the fundamental purpose of these specifications, to be a programming source-code model for developing portable applications. The two types of users (application developers and system implementors) are identified in the scope of the POSIX.1 and ISO C standards.

These standards are the necessary first step in defining a model for developing source code that is as portable as possible. For this model to work, however, platforms that support it must be identified. To this end, the standards further describe the idea of conformance to the standard.

Each standard describes the terms "implementation conformance" and "application conformance" as they apply to themselves. In ISO C, the terms are described as follows:

—A strictly conforming program shall use only those features of the language and library specified in this standard.

—A conforming implementation (both hosted and freestanding) shall accept any strictly conforming program.

Thus, a conforming application program is defined in such a way as to define what a conforming implementation is, by indicating what the conforming implementation (compiler) must accept. Simply put, if programmers confine themselves to writing program source code that uses only features described in the ISO C standard, an implementation that claims to conform to the standard must compile it.

POSIX.1 builds on this idea. POSIX.1 is essentially an extension of the ISO C libraries. It defines the programmatic interfaces to operating system services. It therefore expands the realm of applications to include those that manipulate and use the system services defined in the standard, as well as the ISO C libraries. Conformance is defined for both implementations and applications, again:

—A conforming implementation shall support all required interfaces defined in the standard, and these interfaces will behave as described by the standard. Any nonstandard extensions when used, may change the behavior of these standard functions, but there will always be an environment such that an application can run according to the behavior defined in the standard, and in no case shall that environment require any modification to a strictly conforming POSIX.1 application.

—A strictly conforming POSIX.1 application shall require only the facilities described in POSIX.1 and the applicable language standard (in this case, ISO C).

—A conforming POSIX.1 application using extensions is one that differs from a conforming POSIX.1 application only in that it uses nonstandard facilities that are consistent with POSIX.1

The point of portability is to protect [an] investment.

The Open Group Single UNIX Specfication is a proper superset of these standards. It thus further extends the model for source-code portability discussed here (and suffers the same fate with respect to application source-code conformance). It also adds a large number of historical interfaces (that were not incorporated into the de jure standards) to support the porting of applications developed on traditional UNIX machines to systems supporting the Single UNIX Specification interfaces.

For the purposes of system procurement, the question is whether a system claiming to conform to one of these specifications really does so. Standards development organizations do not police what does and does not conform to the specifications; that job is not within their scope or their budgets. Neither can the average procurement manager be expected to understand the technical minutia of the specifications in order to judge a system claiming to provide them.

To ensure that systems offering these interfaces adhere to some common yardstick, a number of programs were created by people who both cared about the issue and invested their own money in developing the programs. The FIPS 151-2 certification process has been evolving over a number of years for U.S. government procurement with respect to POSIX.1 (as described in FIPS 151-2). An approved test suite is run by an independent third party testing lab; NIST inspects the results, and if all is well, a certificate is issued confirming the test and inspection. This is not a guarantee of conformance, but a reasonable indication that an implementation meets certain fundamental criteria. The cost and complexity of exhaustively testing something like a operating system interface is prohibitive. Note that what is being certified is that the operating system provides the functionality in the standard; the implementation likely provides considerable extra functionality. There is also a FIPS testing and certification program for ISO C compilers.

The Open Group brand program goes further by considering the X/Open test suites as simply an indicator of conformance; the vendor is required to legally warrant that the system does and will conform. This is how The Open Group gets around the hurdle of exhaustively testing implementations, and the warranty adds a certain value over a simple certification based on test suite results. So at this point in the applications development life-cycle, systems procurement can take place with reasonable confidence in source-code portability.

Presumably these systems are being purchased by organizations that either wish to develop portable applications, or already have developed a substantial amount of source code that they wish to port to the new system. The point of portability is to protect the investment that was made in developing the application. It should also be presumed that these organizations have a model in place for portable application development that is defined and “published” in-house; that developers understand it and are rewarded for using it, and that there is a known means of deliberately ignoring the portability model under certain conditions. There would otherwise be no concern about purchasing the right system. It is quite possible, through ignorance or lack of support, to write unportable applications source code on systems supporting POSIX interfaces and ISO C compilers. The specifications simply define the model. The implementations can be certified to support the model, but that does not ensure the model’s use. Purchasing certified systems is an investment in the future of an application’s developed source code, but only if it’s used.

Having laid this foundation for the economic reasons behind porting and portability, we return to the investigation of application source-code conformance, and application source-code certification, or branding.

Application source-code conformance to a specification is a useful concept when discussing what a conforming implementation must accept. It suffers from a number of problems when brought to the real world of application development. First, there are good economic reasons why pure portability of an application’s source code should not be a goal. As with implementation conformance, there can be no guarantee for applications source-code testing, and any attempt to make such a guarantee (in the manner of an X/Open-style warranty) would mean that an application vendor was being asked to promise the impossible. An application source-code conformance certificate or brand provides no useful information to a purchaser of the application; there is no economic reason for it to exist. In fact, such a certificate or brand may give misinformation to the marketplace, and actually confuse and disappoint purchasers when they realize it does not provide them with anything real.

Let’s consider these items. The “best” port would be to recompile and link the application source code on a new machine and have it all simply work. This is the ultimate goal of these source-code portability specifications; to define as complete an environment as possible based on proven ways of doing things to make the porting job easier. There are, however, a number of reasons why applications developers will deliverately choose to break the portability model.

For example, database vendors compete based on their products’ speed and functionality. They may develop as much of their database engine as possible to be as portable as possible, and choose to implement certain key functions in very platform-dependent ways on each platform they support to achieve some maximum performance. A commercial MIS development organization may choose to diverge from its portability model because the application is needed quickly, and will initially be deployed only on certain platforms. In that case, some aspects of the application will be written in a less portable manner.

There can be no guarantees associated with application source code conformance. No technology exists that can scan source code and give a 100% assurance of its future portability to systems that implement the specifications. There are two problems here, one technological, the other to do with the specifications. First, a static compile-time scan does not give any indication of some of the run-time configurable issues with which a program may need to concern itself. Such scanning tools should be part of any development process that seeks to make the source code as portable as possible. Running tools such as the classic UNIX lint utility or ensuring the compiler warning level is set to the most pedantic level are important steps in such a process, but they are not a guarantee.

Second, even specifications as large and robust as the Single UNIX Specification have “holes” in them where implementations are allowed to differ. These holes exist for good reasons, having to do with the economics of developing specifications and gaining support for them from the vendor community, as well as ensuring that interfaces, not implementations, are specified. The holes are indicated by the use of words like “may,” rather than “shall,” and are sometimes flagged as “implementation-defined” or “unspecified.” There are even more subtle holes where nothing at all is said. An example of the first instance is the undefined return for a call to the POSIX.1 function to get the current working directory of a process (getcwd()) with a null buffer pointer. Some systems consider this an error; the standard says that the behavior is implementation-defined. As an example of the implementation versus interface problem, the X/Open Single UNIX Specification is silent on the initialization of master and slave sides of pseudo-terminals. This job is done differently depending upon the way pseudo-terminals are implemented. Berkeley systems do it differently from System V systems, but all systems use calls within the Single UNIX Specification to create and manage the pseudo-terminal. These standards must not dictate an implementation, but rather define the interface to reach the functionality. The implication is that a source-code scanner would see nothing in the source but correct C-language calls to Single UNIX Specification interfaces, but if the source-code was developed on an XPG4 UNIX-branded Berkeley derivative, it would not port with a recompile to an XPG4 UNIX-branded System V derivative.

One might argue that if application vendors ensured that their source code was strictly portable, they would be able to maintain their application less expensively, and the cost of the application would fall. This supposition is simplistic. It presumes that the cost of maintenance and development on new platforms is based largely or mainly on the source-code’s portability—but many other aspects of the software maintenance and development process must be taken into account. In any case, any cost savings will not necessarily be passed on to the customer, but may be reinvested in the functionality of the application.

Developing an application to be as portable as possible (and no more portable, to paraphrase the old simple quote) may help reduce maintenance costs, and likely helps reduce the porting burden to future architectures that provide a similar programmatic interface. Software vendors, however, do not decide to deploy their applications on other architectures based solely on portability. A product is much more than its source code in their eyes, and the port will not happen simply because some potential customers have suggested it. The software vendor needs a complete business case based on the potential size of the market, ability to ship the application and to support it on new and different platforms, distribution channels, and so on.

Sometimes an application customer purchases source code; this is typically done to ensure the availability of the application should the application vendor go bankrupt, and not from a desire of the customer to maintain the source code directly. If the source-code purchaser wants to check the portability of the application’s source code, this should be done in some agreed-upon format as part of the acceptance test. Running lint-like tools provides a useful measure of the source code’s expected portability to similar systems, but it provides no guarantees. Future portability requirements must be balanced against current functionality needs and timeliness considerations.

Against what yardstick of portability is an application’s source code measured? Many interesting applications present some form of graphic user interface. Some use calls into a relational database. While broad in scope, even the Single UNIX Specification does not cover embedded SQL or the X11R6 interface.

The following example illustrates the confusion and misinformation that beset a customer when an application source-code conformance certificate or brand is used. A software vendor currently ships its product line on three different types of traditional UNIX operating systems, plus DEC’s VMS, and Microsoft’s Windows NT. The first four of the implementations (the three UNIX-type systems and VMS) have X/Open XPG4 Base brands, and all of the implementations have FIPS 151-2 POSIX.1 certificates. The exact same application functionality is provided

Groups demanding . . . brands . . . will [soon] be looking for ways around their own rules.

on all five systems. What does it mean to the purchaser when the application's source code is the same on the first two traditional UNIX machines, but a little different on the third; if it uses the same GUI calls on the VMS machine as the traditional UNIX machines, but otherwise uses very different calls for file-access; and if it was completely rewritten to the Win32 interfaces on NT.

Can this application be certified or claim applications source-code conformance to anything? It is the same on two of the UNIX-type machines, but not the third. If it could be made the same on all the systems that support the XPG4 Base brand (and let's further say that the application on these four systems uses the same Xlib calls for the user interface) can the application be branded as conformant to something, even though its source code is still different on another platform that does not support certain implementation brands? If so, then the entire exercise of checking and certifying application source code is not worth the time and effort it takes. Obviously, customers care that the application is available and runs as fast as possible on each platform, and don't care how the source code appears. If not, then the whole point of certification or branding becomes questionable—an application runs on every platform I care about, plus several that I don't (let's say I have no NT machines) yet it is not "approvable."

And what about the unpredictable future? If an implementation vendor changes his system implementation to support the standards and specifications, is a application vendor with source code that is approved in one set of standards-related environments now obligated to change his source in other environments to maintain a seal of approval? At what cost, and to what economic purpose? If a new platform arrives on the scene, as it surely will in the next five to ten years, and the approved application's source code needs to change to be moved to the new platform (despite the fact that the platform may support the implementation certificate or brand), is it still an approved application? What if an application vendor wishes to take advantage of new features presented on a platform to improve the functionality of the application for his customers and to compete? Can he do so without losing his application source-code certificate?

If any form of strict application source-code certification or branding is established, it will affect the ability of software development organizations to compete. Absurdities may afflict the market, such as software vendors competing based on how their source code is written rather than on performance, functionality or creative presentation. (Past calls for this sort of application conformance have come from places like the U.K. National Health Service and certain U.S. federal government agencies; these groups are large enough to demand attention.) If this happends, vendors who do not embroil themselves in the mess will do exceedingly well, providing fast functional applications; and soon the same groups demanding application source-code brands will be looking for ways around their own rules to purchase these faster, more functional, applications.

If any form of weak application source-code certification or branding is established, it will only create a great deal of "busy work," and will confuse customers as to what exactly they are purchasing. Companies will hop through the hoop in front of them, and compete the way they have always competed for customer dollars. The program will not benefit the industry, and may even generate sufficient bad press as to reflect poorly on the current necessary efforts in implementation conformance that provide the base for portable applications development.

In summary, source-code portability is a difficult software development issue. It is complex, as the requirements for source-code portability are different for each application. Customer needs differ depending upon whether one is currently deploying the application on multiple platforms, or developing with future platforms in mind. Providing robust standards and specifications like the ISO POSIX.1 and C-language standards and the Single UNIX Specification, and trusted implementations (i.e., certified or branded systems) is the first and necessary step in the development of more portable applications. It is far from a complete solution, and the complete solution will vary from organization to organization.

Application source-code conformance is very different from implementation conformance. Implementation conformance certification asks a system vendor who has complete control over his implementation to build something based on a stable, known model.

Application source-code conformance, on the other hand, asks an application developer to commit to being portable to systems (some not yet created) that support a particular set of interfaces, regardless of whether or not the application developer will ever need or want to build the application on that platform.

There is no reason to assume that partially measuring an application's source-code portability is of any value to the consumer. The software industry should not be prevented from competing based on its source code implementation strategies. There is no economic rationale to demonstrate that a particular source-code implementation style affects a software vendor's ability to ship products on any platform for which there is a business case. Application source-code conformance registration and certification schemes can only complicate matters and sow confusion in the marketplace. **SV**