Delivering Applications to Multiple Platforms using ANDF[1]

Stavros Macrakis

Open Software Foundation
Research Institute

June 1993

Open systems promise developers a large range of target platforms through standard API's. This promise has only partially been realized, because porting and distribution remain difficult. The Architecture-Neutral Distribution Format (ANDF) is a portability and distribution technology being investigated by OSF Research Institute, that allows a single virtual binary to run on all conforming platforms. We present the technology, discuss its applications, and examine open issues.

1. Introduction

Software developers want applications to run on many platforms. End users want free choice of both applications and platforms. Platform vendors want the freedom to improve their platforms over time.

1. To appear in *AIXpert*, August 1993. *AIXpert* is a publication for AIX developers.

Open systems standards can make this happen through standard Application Programming Interfaces (APIs). But developers have discovered that APIs by themselves cannot guarantee application portability in the absence of API conformity checking and rigorous portability checking. Moreover, even if the source code of an application is portable, each platform requires its own binary version.

ANDF technology proposes a solution to these problems. ANDF checks applications' conformity to an API and translates them into a virtual binary format which can be installed on any platform supporting that API.

2. Defining the ANDF Technology

As an independent organization promoting open systems, the Open Software Foundation (OSF) is the natural forum for providing portability and distribution technology for open systems applications.

OSF thus initiated a request for technology (RFT) for an architecture-neutral distribution format and its associated tools. After defining the problem and design goals, a variety of proposed solutions were evaluated, and one was chosen for further investigation.

The Problem

Software and hardware development, marketing, and purchasing decisions are closely linked today. Software vendors pay heavy porting and distribution costs for each additional hardware platform, so concentrate their efforts on a few, and are restricted to the least common denominator of programming methods and languages available on target platforms. Hardware vendors avoid innovation for fear that applications will not be available, and must build development environments for each new combination of architecture and language. End users are locked into software-hardware combinations where individual components cannot be procured competitively.

Design goals for ANDF

ANDF was thus specified to provide technology to support the development of portable code and to distribute it in an architecture-neutral format.

The major design goals for ANDF were:

- Architecture neutrality—easy to install on new architectures;
- Language neutrality;
- Easy extension to any given API;
- Protection from reverse engineering—hard to reconstruct a source program;
- Efficient code—comparable to native compilers;
- Small size—comparable to usual executables; and
- Openness to future evolution and innovation in software, hardware, and APIs.

Proposed solutions

Three classes of solution were proposed: shrouded source, compiler intermediate languages, and annotated object. Shrouded source is C code where meaningful identifiers are replaced with generated arbitrary symbols and control structure is flattened. It solves the reverse engineering problem, but not the language and architecture neutrality problems. Annotated object is machine code (typically for the 80x86 family) with additional information added to make it decompilable and recompilable. This clearly does not solve architecture neutrality in a satisfactory way.

The ANDF Solution

The solution chosen for further study was a compiler intermediate language, the TDF technology from the U.K. Defence Research Agency[2]. TDF/ANDF is a compiler technology that has been designed from the start to guarantee neutrality.

2. formerly the Royal Signals and Radar Establishment

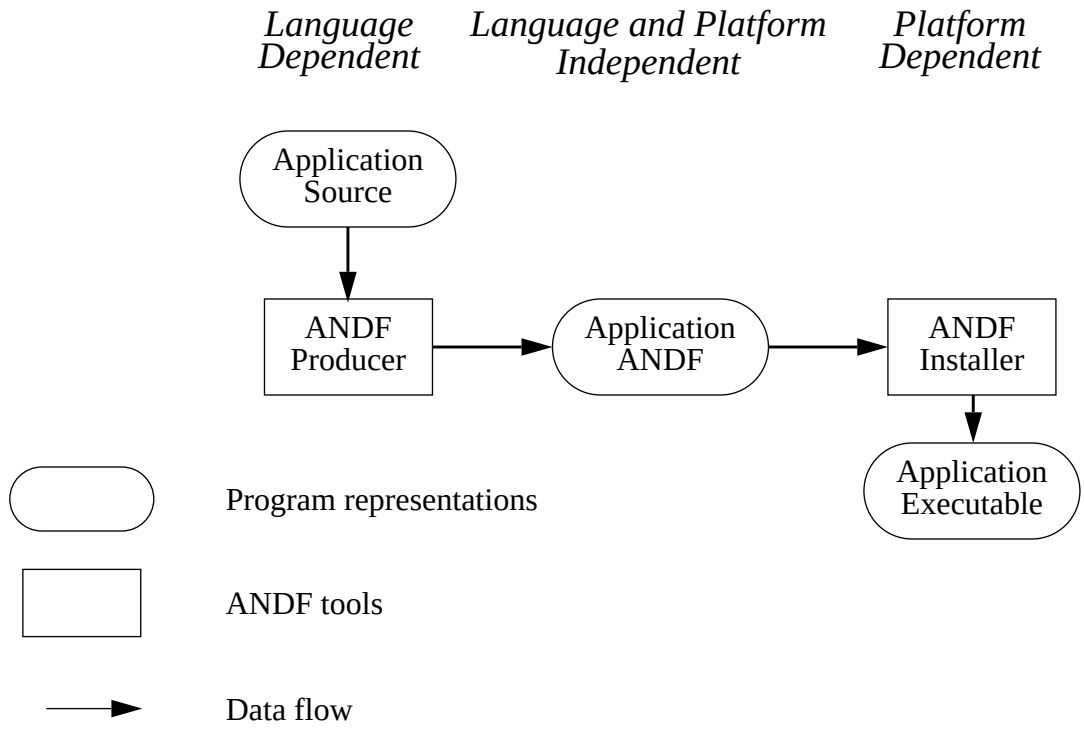

Figure 1. Simplified block diagram of the ANDF technology

3. How ANDF works

An ANDF producer translates from the source language into the ANDF intermediate language, as shown in Figure 1. The ANDF producer is language-dependent but machine-independent. The ANDF intermediate language is language-independent as well as machine-independent. Finally, an ANDF installer translates from ANDF to machine language. Typically, the producer runs on the developer’s machine, and the installer on the user’s machine.

Application source may be written in any programming language supported by some ANDF producer; it must of course be portable code. The ANDF producer transforms this source code into ANDF form. This form removes all language dependencies by translating them into ANDF constructs. The ANDF format is language- and platform- independent, is tightly encoded,

and does not contain the information necessary to reverse engineer the application. The ANDF installer is platform-specific, and creates a platform-specific executable.

This simple scheme would work if programs did not refer to outside libraries. But any realistic program refers to a multitude of libraries defined by the API. And each platform defines the library interfaces concretely, that is, with platform dependencies. But the true definition of interfaces as defined by standards is abstract. It contains only that information necessary to use them, and no information about their implementation.

An important innovation in ANDF is a mechanism for separating the abstract interface into two concrete interfaces: the language-dependent interface, and the platform-dependent interface, as shown in Figure 2. The producer uses the abstract interfaces as expressed for the programming language to check the application's use of the interfaces, and generate appropriate ANDF code. The installer uses a platform-dependent version of the interfaces to translate the ANDF code into native code.

Besides using standard APIs, an application may also wish to take advantage of specific non-standard interfaces on particular platforms. The ANDF technology manages this by providing for platform-dependent libraries and conditional compilation.

The ANDF producer can thus be considered to be a compiler for an ideal platform, which supports strictly standard programming language semantics, and strictly standard library interfaces. The ANDF representation of a program contains only the information needed to run on that ideal platform. The ANDF installer implements the ideal platform by filling out the abstract form with the concrete details of its platform-specific implementation.

The internal structure of the ANDF intermediate language

As we have seen, programs are translated from the source language (*e.g.* C) into the ANDF language. ANDF is a tree-structured language with unambiguous and clean semantics. This makes it architecture- and language-independent, and easy to process internally.

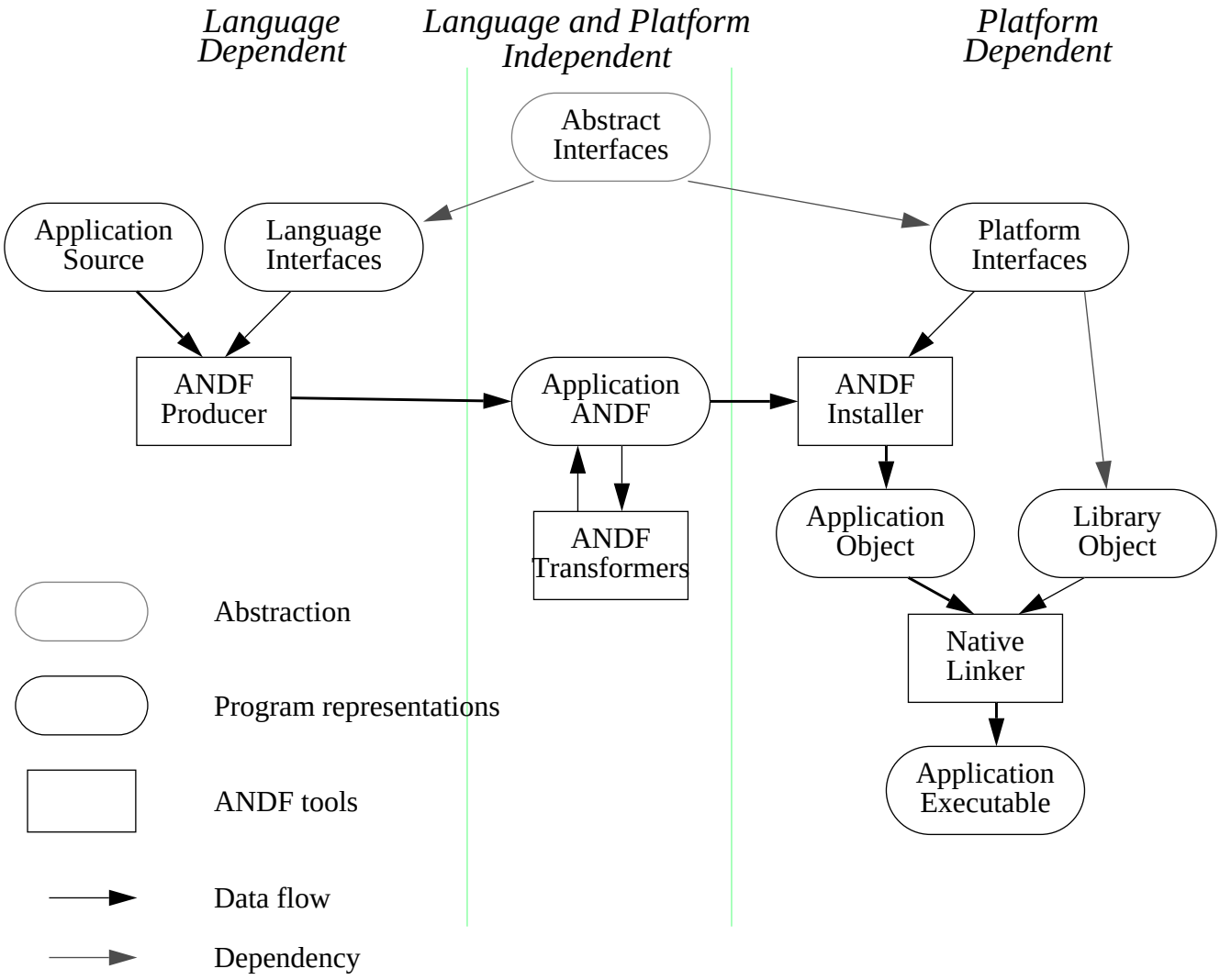

Figure 2. Abstract and concrete interfaces in ANDF

For instance: when a construct has options or attributes, they are all explicit; all operations are completely specified for both normal and exceptional cases independently of target machine; there are no scope rules, since all names are guaranteed unique; ANDF only represents the concrete aspects of types.

One of the major departures of ANDF from conventional compiler intermediate languages is the provision of a general syntax macro facility[3]. Unlike ordinary C compilers, where macro expansion precedes all further

3. Similar to Lisp macros, where a program subtree expands to a subtree; and unlike C macros, where a sequence of tokens expands to another sequence of tokens.

processing, ANDF producers can pass along unexpanded macros to the installer. Thus platform-dependent macros can define platform-dependent implementations of standard interfaces. These syntax macros are also used to save space in the ANDF representation by abbreviating common sequences.

ANDF also supports static conditions (`#if`), which can be used to parameterize program behavior depending on platform.

Besides being powerful, the ANDF representation is also compact, and is comparable in size to machine binaries.

4. Portability and ANDF

The ANDF technology cannot make a non-portable program more portable; however, it performs the important function of checking portability and defining standard interfaces portably.

The installation environment

ANDF itself is independent of the installation environment. This does not automatically imply that all programs distributed in ANDF will be independent of the installation environment. For instance, a program may require particular numerical libraries or operating system interfaces. It may require floating-point precision higher than that provided on the target platform. In these cases, the ANDF installer will report that it cannot install the application.

Alternatively, a non-portable program may depend on dynamic behavior left explicitly undefined by the ANDF specification, such as the transient behavior of a variable which has not been declared as volatile or the result of an unsafe conversion (a C ‘cast’). The ANDF translator does do a good job of flagging unportable constructs, but cannot guarantee identical behavior if programs depend on undefined constructs.

Standard interfaces

ANDF is only useful in the presence of standard interfaces. Today, standard interfaces exist for operating system primitives (XPG, *etc.*) and for certain classes of graphic user interfaces (*e.g.* Motif). There is a natural synergy between ANDF technology and wider interface standards. Since the ANDF technology can deal with source-code conditionals, it can handle multiple incompatible interfaces, but at the cost of complicating application programming.

5. The History of ANDF

The idea of a universal intermediate language goes at least as far back as 1958, under the name “UNCOL” (UNiversal Computer-Oriented Language). The primary motivation for UNCOL was the high cost and slow delivery of compilers for new machines. At the time, compilers were universally written in machine language from scratch, and were thus neither easy to write nor portable.

Regrettably, the project was too ambitious for the technology of the day. UNCOL was never fully defined and certainly never implemented. UNCOL is remembered as the first attempt at a universal intermediate language, but none of its particular ideas have survived it.

Since UNCOL, a great deal of progress has been made in compiler technology in general, and intermediate languages in particular. Most compilers today are divided into two major pieces, the front end, which handles language syntax and semantics, and the back end, which translates to machine language. Compiler porting has become a routine software engineering activity: front ends require only minor modifications; back ends are often table-driven, so retargeting them is relatively straightforward (although tuning for performance is still difficult).

Although intermediate languages are machine-independent in their overall design, almost all practical compilers assume a particular target during front end processing and intermediate code generation, since architecture-neutrality has not been a goal.

TDF/ANDF uses the same fundamental approach, but applies it directly to the problem of neutrality. The ANDF front end is completely target-independent, as is the intermediate language. This is not a radical departure from today's standard compiler technology, but a goal-directed evolution.

6. Distribution of software

ANDF's motivating application is software distribution. Distribution of standard applications by software vendors is the most visible use, but many others are possible.

Distribution of Applications

Before ANDF, a software vendor had to produce a different version of software for each target platform. Even on the same target machine, different operating systems or versions of the operating system might require different binary versions.

With ANDF, a software vendor can distribute a single package, which can then be installed on a wide variety of platforms. This greatly simplifies the software vendor's development and reduces marketing and distribution costs. At the same time, it provides real benefits for end users and hardware vendors, creating a free market by making execution platforms interchangeable and encouraging innovation and competition.

Distribution of Libraries and Modules

Because of its linking mechanisms, ANDF can also be used to distribute software libraries.

Thus end user organizations can build new software out of modules provided by vendors. This would be useless if interfaces were not well-defined, but the increasing importance of object-oriented software design promises to make such standard interfaces more and more common.

Even for standard applications such as X clients and numerical libraries, ANDF makes possible a plug-compatible replacement business.

Single-user “distribution”

A single user nowadays may use many different machines. In particular, in the high-performance computing area, users will often develop and debug programs on one machine as preparation to executing them on another. Today, such users have problems with uniform behavior of their applications across these platforms.

ANDF solves this problem, and gives them even greater flexibility in choice of execution machines.

In particular, a user could call for benchmarks of the execution of a program on a wide variety of target architectures. When the most cost-effective solution was found, the production code could be moved to it with no risk of anomalous behavior.

ANDF for Cross-development

All ANDF development is, in some sense, cross-development: the development environment is different from the delivery environment. This means that, unlike most traditional compilers, the ANDF tools are tuned to cross-development. In particular, the ANDF Producer incorporates one of the most sophisticated portability checkers available. The very fact that a single ANDF producer is used for all targets guarantees that language semantics are uniform across platforms.

7. Some Open Issues

ANDF has thus met most of the goals defined by OSF and its members at the beginning of the RFT process. Yet ANDF has not yet been adopted by industry. Continued analysis at the OSF Research Institute, and discussions with OSF member companies have helped us identify several key open issues, and to take steps to resolve them.

Performance

OSF has performed rather extensive performance measurements on the ANDF technology. For a wide range of platforms and applications, performance is comparable to native compilers (±5%). However, certain applications (notably floating-point intensive ones) are systematically slower on some platforms.

OSF and its partners are investigating the causes of this discrepancy and working towards improving installers.

Language neutrality

ANDF was originally designed to be usable for a wide range of languages, including C, C++, Fortran, Cobol, *etc*. But as of today, ANDF producers are only available for C. Although C is the most important programming language for open systems applications, it is not the only important one.

OSF and its partners are thus actively developing Fortran 77, C++, Ada, and other ANDF producers. Several issues have been identified in these languages which may require ANDF extensions in order to provide good performance while preserving language neutrality. On the other hand, it appears that no changes will be needed to the base technology.

Confirmation of this will have to await validation and performance testing of the completed implementations.

Precision of the definition

ANDF is currently defined informally by an English-language specification. It is also implicitly defined by the existing producers and installers.

As ANDF gains wider use, more precise specifications will be necessary. For this reason, OSF's partners are developing a formal definition of ANDF, and OSF is developing validation methods and suites for ANDF producers and installers. As these efforts progress, confidence in the precision of the specification increases.

Data portability

ANDF does not address the issue of data portability and binary file formats. In particular, ANDF provides no support for hiding the byte ordering of the platform.

OSF is currently investigating this issue.

Reuse of existing compilers

Until recently, all ANDF producers and installers came from the same source code base. Hardware vendors typically have large investments in existing compilers and especially the optimizer phase, so would like to reuse components in their ANDF implementations.

One widely distributed compiler is the Gnu C compiler (`gcc`), which runs on many different platforms. An OSF demonstration project called `gandf` reuses the `gcc` compiler's back end by translating ANDF into `gcc`'s intermediate language. It then generates native code using the `gcc` back end. Not only does this demonstrate the feasibility of reusing back ends, it has also permitted the rapid creation of new back ends. Currently, this back end has a performance penalty of approximately 10% compared to native `gcc` compilers. OSF has also begun work in reuse of existing language front ends.

8. Conclusion

ANDF provides a way of defining an applications programming interface in an architecture-neutral way, then compiling an application to the abstract interface. Compiled programs are thus guaranteed to conform to the interface, and not just some implementations of it.

The resulting ANDF capsule is an architecture-independent representation of the program which can be installed on any of a wide range of machine architectures. ANDF thus helps realize the potential of standard APIs.

Although a great deal of progress has been made in ANDF, it has not yet been adopted by industry. OSF's ongoing research program continues to examine the potential and the limitations of ANDF technology.

The OSF Research Institute encourages hardware vendors and ISV's to work with us to realize ANDF's potential.