Ada Bindings for C Interfaces: Lessons Learned from the Florist Implementation

T.P. Baker, Dong-Ik Oh

Department of Computer Science, Florida State University
Tallahassee FL, USA 32306-4019, Internet: baker@cs.fsu.edu

Abstract. There is an acknowledged need for wider availability of Ada application program interfaces to commercial off-the-shelf software components. One instance of work that is being done to address this need is Florist, the most recent in a series of implementations of the standard POSIX Ada bindings. Experiences with Florist and its predecessors illustrate the strengths and weaknesses of some of the available techniques for implementing an Ada binding based on an existing C-language API.

1 Introduction

Florist is a free implementation of the POSIX Ada bindings packages for use with the GNAT compiler. It provides application program interfaces for both the basic system services (POSIX.5, also known as ISO/IEC 14519-1:1995) and the real-time and threads extensions (POSIX.5b, also known as IEEE Std 1003.5b-1996).

Florist is a second-generation implementation of the POSIX Ada bindings, which draw on experiences with several earlier implementations. One immediate predecessor is Forest, a free implementation done by Kenneth Almquist of AT& T, which has been distributed with GNAT for several years. The other immediate precursor of Florist is a prototype implementation done at the Florida State University by Wan-Hua Lin[4] and Yi-Gang Li [3].

All these POSIX Ada bindings consist of "glue" code that accesses the functionality of an underlying operating system using the OS vendor's C-language application program interface (API). This is not the only way the POSIX.5 standards can be implemented. It is possible that an operating system could be written in Ada, and support only the Ada interfaces. The U.S. Navy has sponsored prototyping of some of the POSIX Ada interfaces in that form. It is also possible to implement the POSIX Ada bindings by direct operating system calls using the assembly language (trap) interface to the local operating system. However, the quickest and cheapest way of obtaining a POSIX.5 implementation is to construct it as a layer over the standard POSIX C-language API.

Timeliness and low cost are important considerations in the development of Ada bindings. Most large software systems today make use of several commercial off-the-shelf (COTS) components, including at least an operating system, a database engine, and a graphical user interface. COTS software components typically include a C-language API. A recent report of the U.S. National Research

Council[2] points out that the additional cost and lag-time in development of Ada API's are obstacles to wider use of the Ada language. The problem of producing an implementation of POSIX.5 is therefore an instance of an important general problem– how to efficiently produce a good Ada binding for an existing C API.

In this paper, we report some of what we have learned from the Florist implementation and its predecessors, about the design, implementation, and maintenance of Ada bindings for COTS C-language API's.

2 Automatic Generation

One of the methods that has been used to accelerate the generation of Ada bindings for C-language API's is automatic translation. There are several tools that will translate C-language header files into equivalent Ada packages. Silicon Graphics, Inc. has used such a tool to provide Ada interfaces that correspond to the C-language header files supported by their operating system. Intermetrics has successfully used another toolset, called `c2ada`, to produce Ada 95 bindings to X Windows, Microsoft Windows, and GCCS.

The essential character of such translations is that they mirror the C-language header files. This is both a strength and a limitation. It is a strength because no separate documentation is required; the documentation of the C-language header files should generally be sufficient to use the corresponding Ada packages. It is also a strength because the Ada packages can be easily updated to match changes in the C header files. It is a limitation because there is no information hiding. The Ada package interfaces must mutate over time to follow mutations in the C headers. Even minor differences in the C headers that would not require modification of a C application may result in visible changes to the Ada package specifications. This is not good for a long-lived Ada application, where stability of the Ada API is important.

Automatic translation techniques are also limited by the nature of the C language. There is considerable "noise" in C header files. Even among operating systems that support standards like POSIX, the standards permit gross variations between header files with the same names. This makes direct translation of C header files impractical for standardized open interfaces, such as the X/Open and POSIX OS API's. These variations include such things as extra declarations, conditional "includes" of other header files, selection between alternate versions of declarations based on user-definable flags (such as `_POSIX_C_SOURCE` and `_XOpen_Source`), variations in order and number of `struct` type components, variations in representation of opaque types, and replacements by macros of apparent names of functions, types, and structure component names. (See Figure 3 for an example of some of these features.)

For example, consider changes in the header file `signal.h` between releases 2.4 and 2.5.1 of SunSoft's Solaris(TM) operating system (not a very major revision, by OS standards). The differences in the two versions of the file itself amount to 230 lines of code (ignoring changes in comments and whitespace).

This includes 14 new constants, and both removal and addition of entire type and subprogram declarations. The code includes conditional compilation commands, which support different C views of the system API (e.g. traditional, POSIX, X/Open). Thus, to produce an Ada package specfication one would need to run it through the C preprocessor first. Unfortunately, this has the side-effect of pulling in various other header files. The differences in header files pulled in by `signal.h` are shown in Figure 1. Expansion of some such embedded headers could be avoided by explicitly pre-expanding them, under the assumption that they are shared by several C header files, and so should be mapped to separate Ada packages. However, that leads to further difficulties, since the exact set of these auxiliary packages will change from one version of the OS to another.

Solaris 2.4	Solaris 2.5.1
	`time.h`
	`sys/time.h`
	`sys/select.h`
	`sys/procset.h`
`sys/stdtypes.h`	
`sys/signal.h`	`sys/signal.h`
`sys/types.h`	`sys/types.h`
`sys/isa_defs.h`	`sys/isa_defs.h`
`sys/siginfo.h`	`sys/siginfo.h`

Fig. 1. Differences in other header files included by `signal.h`.

Some constructs in C header files simply cannot be translated automatically, and so must be worked around using techniques such as pre-editing of the C sources, hand-written glue code, or specification of special-case translation rules to the tool. This hand work needs to be redone each time there is a new release of the COTS product to which the API applies.

Automatic translation tools themselves require porting and maintenance. The maintainer of a binding that is implemented with such a tool relies on the tool's continued availability. For example, the `c2ada` tool has been successfully ported to several systems, but porting it to a new system is the user's responsibility. Moreover it depends on the programming language Python, so a long-term user depends on the the continued availability, portability, and maintenance of Python. For example, when we attempted to install `c2ada` on our system, we found and corrected OS-dependent C header files conflicts, then linker conflicts, then direct dependences on the code and directory tree structure of the Python 1.3 source distribution (we had Python 1.4); it is not clear how long it would have taken to get it to work if we had persisted.

Finally, direct Ada translations of proprietary C header files are subject to the copyright of the original files. In the case of commercial products, this can limit the value of the Ada binding.

In summary, automatic translation is a valuable time-saver for constructing Ada interfaces to COTS products, where copyright can be obtained, some hand-tailoring is acceptable, and there is no requirement to preserve a stable Ada package interface across different vendors' products or across different releases of a single vendors' product. Thus, the technique makes very good sense for a COTS vendor who wants to provide Ada bindings for the vendors' own products. Automatic translation is not so well suited for direct implementation of a portable Ada binding such as POSIX, that must provide a stable Ada package interface across different implementations. Automatically generated package specifications may be useful indirectly, as a lower layer, with hand-written glue code to project the desired Ada interface. However, such an implementation falls short of the ideal of automatic translation, as it is likely to require hand re-work to port it to each version of the product to which it interfaces.

Because Florist is an implementation of a standardized set of Ada package specifications, to which we wanted to avoid adding unnecessary glue code, and which we wanted to make automatically portable to any POSIX-compliant operating system, we did not attempt to automatically translate the C headers to Ada package specifications.

3 Translation of Fragments

An alternative to translation of the full C header files is to translate just small fragments, which are embedded within hand-written Ada code. This technique is used by Almquist's Forest implementation of the POSIX.5 packages, which includes a program called `ctoada`[1] and several scripts. The `ctoada` program makes use of the front end of the Gnu C compiler (`gcc`) to read in C header files, parse them, and translate them into semantic trees. It also reads in a file describing the intended correspondences of names of type and function declarations expected to be in the C header files with names and forms of Ada declarations. It puts out a file containing a macro definition for each pair of C and Ada names, which when processed and expanded by the `m4` macro processor will produce the text of an Ada declaration that is equivalent to the C declaration in the header files. There is another program that produces similar `m4` macro definitions that produce Ada expressions corresponding to the values of C preprocessor constants. The Forest package specifications and bodies contain `m4` macro calls for declarations of interface types, functions, and constants. They also use `m4` macro calls for conditional compilation of target-dependent bits of glue code.

For example, the POSIX Ada bindings define an implementation-dependent subtype, `Child_Processes_Maxima`, which specifies what is known at compile time about the maximum number of simultaneous processes that may have the same user ID. Forest defines this via a call to a macro, `CHILD_PROCESSES_RANGE`, and the macro definition is generated by the tools.

For another example, the C-language signal interfaces use a system-dependent data type called *struct sigaction*. Forest needs a corresponding Ada type declaration to implement the body of the package `POSIX_Signals`. This is done by a call to the macro DECLARE_signal_action, whose definition is generated by the program ctoada. Figure 2 shows the input scheme provided to ctoada, the m4 macro definition produced by ctoada from this scheme under the Solaris 2.5.1 operating system, and the expansion of the macro call into Ada code.

```
define DECLARE_signal_action indent 0
    type signal_action is struct sigaction
end define
```

The macro definition scheme for DECLARE_signal_action.

```
define([DECLARE_signal_action], [type array_type_9 is array (integer
range 0 .. 1) of Interfaces.C.int;
type signal_action is record
    sa_flags:    Interfaces.C.int;
    X_funcptr:   Bad tree;
    sa_mask:     POSIX_Private_1.signal_mask;
    sa_resv:     array_type_9;
end record;
pragma convention(C, signal_action);])
```

The macro definition generated by ctoada.

```
type array_type_9 is array (integer range 0 .. 1) of Interfaces.C.int;
type signal_action is record
    sa_flags:    Interfaces.C.int;
    X_funcptr:   Bad tree;
    sa_mask:     POSIX_Private_1.signal_mask;
    sa_resv:     array_type_9;
end record;
pragma convention(C, signal_action);
```

The expansion of the macro call.

Fig. 2. Forest treatment of *struct sigaction*.

Note that this is a case where the tools fail. The immediate reason is clear if one examines the corresponding C delcarations, which are shown in Figure 3. There is a C union type declaration, which the program could not handle. This is minor problem, which could be solved by expanding the capabilities of the ctoada program. However, there is a more difficult problem that is not so easily solved. The C structure declaration does not comply with the POSIX specifiations. The component names *sa_handler* and *sa_sigaction* are not present. Instead, they are given as macros, that refer to subcomponents in a nested structure.

This technique does not eliminate the need for hand work, but it greatly simplifies the job of porting an Ada binding to new versions of the underlying C-language API. It has the benefit that one can ignore much of the content of the C-language header files and just focus on bits that are of interest. The combi-

```
struct sigaction {
        int sa_flags;
        union {
#ifdef  __cplusplus
                void (*_handler)(int);
#else
                void (*_handler)();
#endif
#if defined(__EXTENSIONS__) || ((__STDC__ - 0 == 0) && \
       !defined(_POSIX_C_SOURCE) && !defined(_XOPEN_SOURCE)) || \
        (_POSIX_C_SOURCE > 2)
                void (*_sigaction)(int, siginfo_t *, void *);
#endif
        }       _funcptr;
        sigset_t sa_mask;
        int sa_resv[2];
};
#define sa_handler      _funcptr._handler
#define sa_sigaction    _funcptr._sigaction
```

Fig. 3. Solaris 2.5.1 declaration of *struct sigaction*.

nation of automatically translated pieces, conditional compilation, hand-written code permits the implementation to be thin. A disadvantage is that porting the binding to a new environment means porting the tools and maintaining the binding means maintaining the tools, through changes in the operating system, `gcc` compiler, and `m4` macro processor.

At an early stage, we intended to use `m4` with `ctoada` and the other Forest tools to implement Florist. As a warm-up for this project, we used the Forest mechanisms to reimplement the package interfaces that the the GNAT Ada runtime system (GNARL) uses to obtain services operating systems that have a POSIX-like C-language API. This was successful in reducing the amount of glue code as compared to the original implementation, and in allowing one to more easily port the GNARL to new OS's and new OS versions[5].

However, we ran into several problems. When we tried the `ctoada` program on new operating systems, we found that it died on some of the more complicated type definitions in the header files, such as the *struct sigaction* example described above. By patching the tool, we enabled it to get through the header files, but there was still a need for some hand editing. This was needed for some type declarations that the tool could not translate, and for the cases where the C header used macros to stand in for functions and record component selectors. Then, when we went to a new version of the `gcc` compiler, we found that `ctoada` would no longer compile, due to dependences on `gcc` installation configuration information that had changed since the `gcc` version from which `ctoada` was derived. We also found problems with some identifiers in the Ada sources being incorrectly interpreted as calls to `m4` macros. This was correctable by using different names for the `m4` macros. We also needed to use features that varied across implementations of `m4`, and so relied on a particular version of `m4` being portable to each system.

Our goal for Florist was that it should automatically configure itself to new operating system versions, and that users could be easily port is to a new operating system without our help. We judged that maintaining and porting Almquist's Forest tools would require more effort than we wanted to ask of Florist users.

4 Binding-specific Generators

The current Florist implementation is mostly hand coded, with a few automatically generated packages. It uses a combination of two mechanisms to configure itself to a new system. The first phase is execution of a configuration script, similar to that used by all the Gnu software products. We generate the script using our own derivative of the Gnu `autoconf` `m4` macro set, but the user does not need `m4` to run the script. The shell script that the Gnu tools produce is executable on virtually all UNIX-like systems. The configuration script searches for the POSIX C header files, specific object libraries, and for specific names within the header files. For each POSIX function, it attempts to compile and link a dummy program that calls the function. The result is a set of C preprocessor `#include` directives and macro definitions that specify which POSIX features are supported by the underlying C-language interface of the underlying OS. The second phase is compilation and execution of a portable POSIX C program, which we call `c-posix`. That program generates the complete Ada source code of a few Ada package specifications, principally the one called `POSIX.C`.

The `POSIX.C` package provides a complete direct Ada binding for all of the standard POSIX C-language interface. For features that are not supported by the local operating system, it provides dummy declarations to permit compilation of code that refers to the interface. This is to support a special requirement of POSIX.5, that lack of OS support for any feature does not cause failure of a compilation, but only causes an exception to be raised at run time. There are also Ada constants that can be interrogated to determine whether a given feature group is supported, and for C interface functions that are not supported by the OS, a dummy body is provided that returns a failed status value with the appropriate error code. The `POSIX.C` package is almost entirely interface declarations. The only code in the body is for a few type conversion functions, and that code is completely portable.

`POSIX.C` is self-contained. It does not use the standard `Interfaces.C` package, but instead provides its own Ada declarations for all the C types that are needed. Very few of the C-language types that are used by the POSIX C interfaces are covered by `Interfaces.C`, so we were forced to define Ada version many C types ourselves. We also found that the `Interfaces.C.Strings` package was not suitable for our purposes. The POSIX.5 implementation packages need direct visibility of the Ada equivalent of C's `char *` and `char [ ]` types. The Ada type `chars_ptr_array` is not equivalent to `char [ ]`, since its representation includes dope. We originally used the types from `Interfaces.C.Strings`, with unchecked conversions and various constrained subtypes of `chars_ptr_array`. The code was very hard to read. We also had trouble remembering which package contained which type declaration, among all the C interface types. We tried adding our own declarations for the few interface types we had been using from `Interfaces.C` and `Interfaces.C.Strings`, to `POSIX.C`. This simplified our code and made it much more readable.

The `c-posix` program also generates the specifications of the packages `POSIX`, `POSIX.Limits`, and `POSIX.Options`. The rest of the 40 package specifications, and all of the 37 package bodies of Florist are pure portable Ada code.

For example, in Florist the declaration of the subtype `Child_Processes_-Maxima` in Florist is directly generated by `c-posix`. There is no Forest-style macro-call in the source code. The Florist treatment of the C-language*struct sigaction* is shown in Figure 4. This and all other system-dependent declarations are contained in the single implementation package `POSIX.C`, whose specification is generated by `c-posix`.

```
type struct_sigaction is record
   sa_handler : System.Address;
   sa_mask : sigset_t;
   sa_flags : int;
end record;
for struct_sigaction use record
   sa_handler at 4 range 0 .. 31;
   sa_mask at 8 range 0 .. 127;
   sa_flags at 0 range 0 .. 31;
end record;
for struct_sigaction'Alignment use ALIGNMENT;
```

Fig. 4. Florist treatment of *sigaction*.

.

Note how the difficulties with `ctoada` have been solved. Nonstandard parts of the C structure definition, that are not specified by the POSIX standard, are simply left out of the Ada declaration. Nested structure definitions are not visible; the field-names that are implemented as macros in C appear as normal Ada component names. The Ada representation clause does this, using information extracted by `c-posix` using the standard C-language operations to discover the sizes and addresses of component objects. The code of the program `c-posix` that deals with such declarations is very repetitive. We have made it simple to modify, by appropriate use of C macros. For example, the code that generates the declaration above is shown in Figure 5. The macro `GT2` provides all the information needed to generate a single *struct* component. The `#ifdef`s allow us to provide a dummy delcaration for systems that do not support the POSIX standard.

At the time of this writing, Florist has been compiled and tested on Solaris2.4, Solaris2.5.1, and a Linux-based Gnu system with kernel version 2.12. The configure script and the `c-posix` program have been tested on versions of the IRIX, AIX, OSF/1, and HPUX operating systems. We have not yet been able to get access installations of GNAT on the latter systems to compile and test the Ada code. We are hoping that by the time of the conference we will have more to report.

```
/* generate the declaration of subprogram g_sigaction
 */
#ifdef HAVE_struct_sigaction
  GT1(sigaction,1)
#else
struct sigaction {
    void (*)() sa_handler;
    sigset_t sa_mask;
    int sa_flags;
    void (*)(int,siginfo_t *, void *) sa_sigaction;
  };
  GT1(sigaction,0)
#endif
  GT2(sa_handler, void (*)())
  GT2(sa_mask, sigset_t)
  GT2(sa_flags, int)
#ifdef HAVE_sa_sigaction
  GT2(sa_sigaction, void (*)(int,siginfo_t *, void *))
#else
#endif
  GT3
...
/* call the subprogram, to generate the Ada type declaration
 */
g_struct_sigaction();
```

Fig. 5. Source code from *c-posix* for *sigaction*.

.

5 Conclusions

Our experience leads us to believe that one must approach the design and implementation of Ada bindings in more than one way. There are at least two distinguishable cases, that will benefit from different treatments.

For a vendor of a COTS product, automatic translation of C headers provides a cost-effective method of providing Ada bindings for the vendor's customers. Copyright and maintenance considerations make it more cost-effective for this work to be done by the vendor, rather than by a third party. Therefore, in this case Ada users will be better served if they can persuade the COTS vendor to take on the responsibility of providing an Ada binding.

The second case is that of a standard Ada binding, such as POSIX.5. The overhead of developing and maintaining such standards does not make sense unless they are for an interface that will be supported by multiple COTS products, and will be used by many applications. If there is an underlying portable C-language interface, the implementation of the Ada binding should use this, providing its own tools to perform any local configuration that may be needed. For this case it might be practical for the implementation of the Ada binding to be provided by a third party.

This leaves cases where a specific application needs an interface to a particular COTS product, for which there is no standard interface and for which the vendor does not provide an Ada binding. Automatic translation of C header files may make sense if the system is not expected to be maintained across many upgrades of the COTS component. However, if it is expected to be maintained over a term that is long enough to encounter a series of COTS component versions or require porting to a completely different COTS product, it may make more sense to develop an application-specific package, that hides the specifics of the COTS API. By isolating an limiting the visibility of COTS-dependent interfaces, such a package is likely to pay for the cost of rewriting its internal glue code. Moreover, since most applications do not use more than a small subset of the typical COTS component interface, writing an application-specific binding may be less hand work than is required to make up for the limitations of the automatic tools if one attempts to translate a full set of C header files.

Florist Availability

The Florist 1.1 implementation is available in source-code form via anonymous from the Florida State University Computer Science Department (via URL `ftp://ftp.cs.fsu.edu/pub/PART/FLORIST`). There are plans to improve and extend this implementation, at least to support the draft POSIX.5c socket interfaces. We welcome electronic mail correspondence, including defect reports, suggestions for improvements, and offers of help porting Florist to other systems.

Acknowledgments

This work is part of the FSU POSIX/Ada Real-Time (PART) Project. PART has been funded by the Ada Joint Program Office, through the HQ U.S. Army CECOM, Software Engineering Directorate.

References

1. Almquist, Kenneth: The Forest Software Library. available from New York University via URL `ftp://ftp.cs.nyu.edu/pub/gnat/contrib/forest`.
2. Boehm Barry *et al.*: Ada and Beyond: Software Policies for the Department of Defense. U.S. National Research Council Report. (1996)
3. Li Yi-Gang: A Prototype Implementation of the POSIX/Ada Realtime Extension. Technical report, Computer Science Department, Florida State University. (1995) available from the Florida State University via URL `ftp://ftp.cs.fsu.edu/pub/PART/publications/lireport.ps.gz`.
4. Lin Wan-Hua: A Prototype Implementation of the POSIX Ada Interface. Technical report, Computer Science Department, Florida State University. (1995) available from the Florida State University via URL `ftp://ftp.cs.fsu.edu/pub/PART/-publications/linreport.ps.gz`.
5. Moon Seung-Jin, Baker T.P., Oh Dong-Ik: Low-level Ada tasking Support for GNAT – Performance and Portability Improvements. in *Proceedings of the Washington Ada Symposium*, also available via URL `ftp://ftp.cs.fsu.edu/pub/PART/-publications/wadas96.ps.gz`. (1996)