How Do You Release a Product Implemented in Ada?

Making Compatible Interface Changes with the BiiN™ Ada Compiler

David B. Kinder

BiiN
2111 NE 25th Avenue
Hillsboro, OR 97124-5961

INTRODUCTION

In a large Ada software development project, such as an operating system, the effects of making a new product release with changes to public interfaces can cause significant recompilations of direct and indirect users of this interface. This cascading recompilation problem seems inherent with a language such as Ada — a language that has well defined interface checking rules. However, in a production environment, it is critical that updates to delivered interfaces remain compatible with existing customer's code. A product upgrade with enhanced but upward compatible interfaces, should not force users to recompile their applications.

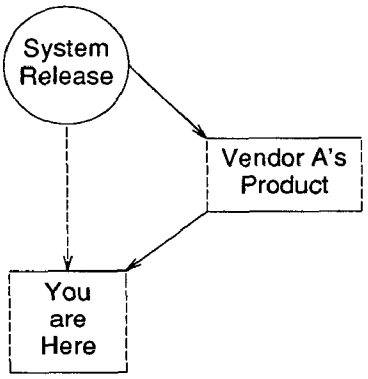
This paper describes how an intelligent Ada recompilation service was integrated into a large Ada development project (over two million lines of production Ada code). The paper also describes the uses of this service that go well beyond simply reducing unnecessary compilation, wherein it becomes a critical tool in the product development, generation, and customer release cycle.

THE PRODUCT RECOMPILATION PROBLEM

Despite the claims and goals of top-down, object-oriented, and data-abstracted designs, interfaces *do* change during the lifetime of a software project. Not only do interface changes cause delays in the work of the software developer, they can have an unacceptable effect upon the deliverable products of value added suppliers.

BiiN™ and BiiN/OS™ are trademarks of BiiN Partners.

For example, consider the simple yet typical product releasing scenario illustrated in the following figure:



Assume you're a user that has bought a product from Vendor A that provides a library of Ada utility routines built on top of operating system services (also written in Ada). Both the products released from Vendor A as well as the operating system release are distributed as Ada libraries which include the visible specifications for compiling, and the product's object code for linking.

For the initial release, the same system interfaces are used by Vendor A as are delivered to you. Later, when we deliver a system upgrade, we must guarantee that the system interfaces remain compatible with the original release so that Vendor A's product does not need to be recompiled. Indeed, these new interfaces and the delivered Ada library *must* be compatible because you probably don't even have the sources of Vendor A's product to recompile!

CHANGING INTERFACES

Let's examine why interfaces can change. For typical source control and Ada compiler implementations, a simple source change, such as adding a comment, causes recompilation managers to believe that the source needs recompiling and, transitively, all dependent units need recompiling as well. (In Ada, dependencies among compilation units are defined by context clauses (**with**

clauses). A compilation unit that mentions other library units in its context clause *depends* on those library units. Indirect dependencies exist among compilation units by transitive analysis of the direct dependencies.) In a development environment where project documentation is maintained in the program source files (and extracted by a document generation tool), these unnecessary system recompilations have a very negative effect on programmer productivity.

Similarly, when an interface is extended with an additional type or procedure declaration, the same thing happens. Even though there may be no *need* to recompile dependent units that don't depend on these new declarations, they are recompiled anyway — simply because they depend on some other declarations that were not changed but live in the same source file as those that were.

This situation is not unique to Ada. Consider a C header file where a declaration is defined and then included into C programs that use it. If the definition in the header file is changed, all C programs that use this definition *should* be recompiled. The C language does not require a recompile nor does any C compiler do a consistency check that these recompilations have been done. Instead, the user is left on his own to guarantee that all programs use consistent versions of common header files. In Ada, the language *requires* the compiler and Ada librarian to maintain unit consistency.

The problem then is to determine what kinds of interface changes are benign, or upward compatible, and do not introduce inconsistencies into the development system. Our goal is to totally eliminate unnecessary recompilations caused by compatible interface changes. Because of this goal, when upward compatible releases of software products (in the form of an Ada library) are delivered to customers, there will be no forced requirement for all dependent customer code to be recompiled. This is critical for software product suppliers such as Vendor A in our scenario above, that do not include the source of their software to their customers.

SOLUTION PRINCIPLES

The BiiN Ada compiler, like most other Ada implementations, uniquely identifies compilation units with a version/time-stamp that permits the compiler to verify that the same compiled version of a unit is seen by all components of a product. If a compilation unit is recompiled, whether it has changed or not, the compiler gives the unit a new version/time-stamp. In BiiN Ada, the object files produced by the compiler also contain time-stamp information, so that the linker can additionally verify the expected versions of dependent units are linked together — the defining object module version must match the time-stamp of the referenced version recorded in the object module of referencing compilation units.

In C (and other languages that support separate compilation), subprogram prototypes are written by the programmer in an header file and this file is textually included (via a #include directive) into every source file that uses the interfaces described by this header file. Each compilation pays the expense of scanning, parsing, and verifying the semantic correctness of the contents of these header files. In a large system this can amount to a significant overhead for every compilation.

In BiiN Ada however, when a specification is compiled, an interface file is produced which contains the pre-digested information about the interfaces in a form that can be quickly processed by users of the interface. Information about types, type sizes, record field bit positions, variables, data addresses, subprogram parameter passing profiles — in short, all the separate compilation information needed for a dependent unit — is recorded in the interface file.

The compiler produces an interface file for each compilation unit (CU) presented to the compiler. This file, containing separate compilation information, is stored in an Ada Library. After recompiling a CU (and before updating the library), both the old and new interface information is available. Under user control, the compiler can compare the old and new interface information, and if deemed compatible, the new file is "blessed" as a compatible version of the old interface file and is installed into the Ada library as a new (and compatible) version of the existing interface.

With blessing, multiple acceptable versions of an interface and object module are now possible — where newer versions provide superset implementations of previous versions. Blessing is integrated with the Ada librarian to handle these multiple versions in a consistent manner. When a unit is looked-up in a library (during processing of a context clause), the latest version is retrieved. When linked, there are multiple acceptable versions of the object files referenced by the user's of an interface. The linker verifies that the defining object file provides an interface that is acceptable to all the referencing object files.

A Comparison with Other Recent Work

In comparing blessing with the recent work of Tichy [1] on Smart Recompilation, and Schwanke & Kaiser [2] on Smarter Recompilation, one clear difference is evident. Both of their approaches assume that all compilation units are known when a "smart" recompilation is performed. They rely on a *reference set* from each dependent compilation unit to decide whether a recompilation of dependent units can be avoided. This approach works fine when *reference set* information is available. But notice that their solutions don't help the situation when there are users of an interface that are not known — when an Ada library of interfaces is shipped to a customer as part of a product release. "Smart" recompilation doesn't address the problem of customer source recompilation because customer's code is not part of a vendor product's Ada library.

The BiiN approach is based instead, on the assumption that information from customer's dependent units is not available — as is the case when products, and the interfaces the user can compile against, are delivered to customers. We do not depend on any specially generated information other than the normally produced interface files already used by the compiler for separate compilation support, so there's no increase in disk space used when blessing services are utilized. Blessing decides on the upward compatibility of an interface by examining the interface itself.

Blessing and Uses

Ada does put some obstacles in the way of blessing interface changes. Issues arise of static semantic incompatibilities that could be introduced because of Ada's visibility and name overloading rules. For example, our so-called "compatible" addition of a new function into a package specification could actually cause a compile-time error in a dependent compilation if that compilation does a **use** on both this new interface and a package that also defines a function with the same name. (In this case, Ada considers these functions to be homographs that hide each other, and would therefore require qualified names for a reference to either function.)

While on the surface this appears to be a fatal flaw with blessing, in practice, it is quite innocuous behavior given the benefits that blessing provides — all existing compiled code will continue to function identically, and newly compiled code can use the new facilities provided by the compatible interface changes. As an aside, the design (and implementation) of blessing does not prohibit the collection of further information about dependent units, as is done for Tichy's "Smart" recompilation process. With any usage information about dependent units, blessing can only become "smarter" in what changes it allows. Our intention, though, was to solve this recompilation problem when no information about dependent units was available.

For the problematic change described above, a compiler error message would be given for the use of a hidden identifier, if the unit doing the **use** were recompiled, and fully qualifying the reference would remove the problem.

Note that blessing did not create this compilation error. Even without blessing, given this change of adding a new function with the same name as a function in another package, the user would still be required to change his source to add name qualification. Blessing merely deferred recognition of the problem until the dependent unit was recompiled, for some other reason.

THE IMPLEMENTATION

The key to blessing is its integration with the Ada library manager and Ada compiler. Blessing is controlled by the compiler **:bless** option. Before a successful compilation of an interface is installed in the Ada library, the blessing tool (actually a separate pass of the compiler) retrieves the existing interface from the library and compares it with the interface that is about to be installed. If no differences are detected, or if "compatible" changes were made, the new interface will replace the old. If incompatible differences are found, then the compilation is rejected and the old interface remains in the Ada Library.

Inside an Interface File

An interface file in a BiiN Ada library is a network of *exprs*, where an *expr* is either a symbol table entry (such as for a type, procedure, or variable) or a node in an expression or statement tree. Information about types, type sizes, record field bit positions, variables, data addresses, subprogram parameter passing profiles — in short, all the separate compilation information needed for a dependent unit — is recorded in the interface file.

Interface files are complete — they contain a copy of all *exprs* from **with**'ed units that are needed by the current CU. This makes the processing of context clauses and the retrieving of separate compilation information from the Ada library very fast. Unnecessary *exprs* are pruned away to keep the size of interface files small.

Since there are copies of *exprs* from **with**'ed units within the current CU's interface file, it takes only three CUs to show how the "same" definition can enter an interface file from two different **with**'ed units.

```
package A is
   type COLORS is (
      VIOLET, BLUE, GREEN,
      YELLOW, ORANGE, RED);
end A;

with A;
package B is
   X: A.COLORS;
end B;

with A, B;
package C is
   V: A.COLORS := B.X;
end C;
```

Ada's dependency rules require these CUs to be compiled in the order: A, B, C. The *exprs* for the type A.COLORS and all the enumeration values, come into the compilation of package C, from the interface files of both A and B. Ensuring that only one copy of the definition *exprs* actually exists during the compilation of C is critical to both the semantic checks of the compiler and to the compiler's performance and resource usage. To do this, all *exprs* are assigned a unique key within their CU. All dependent units refer to the same *expr* in a **with**'ed unit by the Ada library-wide unique identifier (*CU_name, expr key*). The compiler's ability to recognize the same *expr* definition coming through multiple paths assumes that all copies of the original *expr* have

the same *expr* key. This is how the compiler knows for example, during semantic checks, that two variables are of the same type — they refer to a type *expr* with the same *expr* key.

In general, recompilation after arbitrary changes to the source can produce an interface file bearing no resemblance to the original interface file or to any copies. It is blessing's responsibility to restore order to the *expr* keys introduced by such recompilations.

After compiling A, B, and C, say we make a change to the type definition of A.COLORS and recompile A. If we attempt to recompile C (without recompiling B) the compiler will notice that the unit B is obsolete and will abort the recompilation of C. This is important to do since the definition *expr* for the type A.COLORS that would come into C from A differs from the copy that would come into C from B.

Continuing with our example, what if instead of changing A.COLORS, we add a new type definition HOT_COLORS to the package:

```
package A is
   type COLORS is (
      VIOLET, BLUE, GREEN,
      YELLOW, ORANGE, RED);

   subtype HOT_COLORS is
      COLORS range YELLOW .. RED;
end A;
```

There is no need for dependent CUs of A (namely B and C) to be recompiled, since no change was made to an existing definition. We would like to avoid recompiling dependent units of A. We want to bless the new A interface as being compatible with the old A interface.

Making Interfaces Internally Compatible

Now, what is involved in making interfaces compatible should become more clear. When a specification is recompiled, each item in the interface must be given the same unique key it previously had. This is normally true for a recompilation only if the sources are identical (except for white-space and comment changes).

If however, a source change *were* made, say by adding a new type declaration, then the unique key scheme would be thrown askew. All items after the newly inserted declaration would not have the same key as in the previous interface, and other compilation units that referenced the old-key item would no longer refer to the correct declaration! If any unit that did a **with** on this changed unit were recompiled, then all of these users of this changed interface would need to be recompiled even though they did not reference the new declaration. The new interface would be internally incompatible with the old. To bless the new interface, the compiler must perform a transformation of the items in the new interface to give them the corresponding old interface unique keys.

Without blessing, the new file would simply replace the old interface and would obsolete dependent units. The secret to interface blessing is to know which interface differences are important and which are benign or upward compatible. For a useful set of changes to an interface, the new interface can be *made* compatible with the old interface.

For blessing to be successful then, each *expr* in the new interface file must be given the same *expr* key as the corresponding *expr* in the old interface file. To do this, the blesser must find these corresponding *expr*s and set the new *expr*'s key to be the same as the old. Finding the corresponding *expr* is essentially a tree matching problem.

An *expr* in the new interface file that does not have a corresponding *expr* in the old interface file is considered an addition to the interface. In this case, a compiler note message is produced to remind the user that a compatible change was detected, and the blessed CU is put into the Ada library.

An *expr* in the old interface file that does not have a corresponding *expr* in the new file is considered to have been deleted. An *expr* in the old interface file that has been deleted or is not the same as its counterpart in the new file, is grounds for not blessing the new interface file. For these, a compiler error message is given to the user, and the Ada library is not updated.

PERFORMANCE

Some interesting statistical information was gathered to help decide upon the algorithms used for blessing. Over two hundred BiiN/OS Ada package specification interface files were analyzed. Of these, over two-thirds of the interface files contained under 100 *exprs* for declarations made by this package.

```
Average is 98.4 exprs per Interface

# exprs  # interfaces
---------------------+--------------------
   0- 24: 34 (16.8%) | ******************
  25- 49: 39 (19.3%) | *******************
  50- 74: 39 (19.3%) | *******************
  75- 99: 24 (11.8%) | ************
 100-124: 19 ( 9.4%) | *********
 125-149: 10 ( 4.9%) | *****
 150-174: 10 ( 4.9%) | *****
 175-199:  6 ( 2.9%) | ***
 200-224:  5 ( 2.4%) | **
 225-249:  2 ( 0.9%) | *
 250-274:  1 ( 0.5%) |
 275-299:  2 ( 0.9%) | *
 300-324:  1 ( 0.5%) |
```

Guided by these statistics, blessing was implemented with a rather straight-forward linear searching algorithm for finding the corresponding *expr* in the old interface for each *expr* in the new interface. Remarkably, compilation time statistics also show that for a 500 line package specification's interface file (containing over 150 *expr*s),

the time to perform the blessing service was less than one CPU second. This was true not only when comment and white space changes were made, but also when a new function was added right in the middle of the package specification.

WHAT ARE COMPATIBLE INTERFACE CHANGES?

For common cases of interest, this is a straightforward question to answer. An interface change is compatible, from an object code point-of-view, if code compiled against the old and new interfaces will behave identically. This pragmatic answer is the key to determining the compatibility of a given interface change.

In its design, blessing was very conservative, and permitted only very controled changes to an interface. As we observed its use and behavior, and the kinds of changes being made to interfaces within BiiN's development projects, we modified blessing rules to permit more extensive changes. These compatibility rules continue to evolve.

Compatible Changes

An interface is compatible if code generated by users of both the old and new interface behaves identically. Specifically, this means that the compiler-known information about basic declarations stored in the interface files must not be affected by the changes made to the new interface.

The most common compatible change is adding or changing a comment, and it is always considered compatible since no new *expr*s are added and no old *expr*s are removed from the existing interface.

Additions to the end of a CU, such as adding a new type, procedure, or variable are allowed, and are also considered compatible. With a few exceptions (described below), additions may also be made at places other than the end of the library unit, but more care must be used to maintain compatibility.

- New object declarations (such as variables) and type declarations that cause implicit declarations of objects (such as temporaries used to freeze the bounds of a dynamic subtype declaration) must be placed after all existing object declarations (explicit or implicit). The reason for this is simple. The generated code references a variable by a compiler-known offset within the public (specification-defined) data for the interface. If a new variable is introduced in the middle, then all subsequent variables would be referenced with an incorrect offset by any existing code.

- New overloaded subprograms must be placed after all existing subprograms with the same name. The compiler generates unique linker names to resolve references to overloaded subprograms. These names are based on the order in which the overloaded subprograms are declared. Changing this order would result in the wrong subprogram being called by any existing code.

Blessing detects all of these incompatible changes and reports the problem to the user.

The following are *compatible* changes to an interface:

- No change at all (i.e., recompiling the same source file)

- Adding (or changing) comments, space_characters, or format_effectors

- Adding new basic declarations

As described above, new declarations may be placed anywhere within the package so long as they do not change the interface information about other basic declarations. In general, new types, scalar constants, and non-overloaded subprograms can be added anywhere in the package specification. New variables (both explicit and implicitly defined by subtype declarations) may only be added after all existing variables. With-lists changes are always compatible.

Incompatible Changes

Altering an existing type definition or subprogram specification is generally incompatible. In the design of the blessing service, interesting type changes were investigated for their compatibility. For example, adding an element to an enumeration type was considered, but finally rejected. To see why, consider this example:

```
package COLOR_MGR is
   type COLORS is (
      VIOLET, BLUE, GREEN,
      YELLOW, ORANGE);

   function P return COLORS;
end COLOR_MGR;
```

The type COLORS defines:

- 5 enumeration literals (VIOLET=0, BLUE=1, GREEN=2, YELLOW=3, ORANGE=4)

- overloaded operator symbols ("=", "/=", "<", ">", "<=", ">=")

- attributes (SUCC, PRED, FIRST, LAST, RANGE, POS, VAL, IMAGE, VALUE, SIZE)

Adding a new enumeration literal to the type COLORS affects all of these definitions. For example, consider this user of the package COLOR_MGR:

```
with COLOR_MGR; use COLOR_MGR;
procedure USER is
   type PALETTE is
      array( COLORS) of BOOLEAN;
   PAL: PALETTE;
   B:   BOOLEAN;
begin

   for C in COLORS loop
      PAL(C) := FALSE;
   end loop;

   PAL( COLOR_MGR.P) := TRUE;

   case COLOR_MGR.P is
      when VIOLET..YELLOW =>
         null;
      when ORANGE =>
         null;
   end case;
end USER;
```

Consider the effects on procedure USER if the type COLORS were changed by adding a new enumeration literal (say adding RED after ORANGE), and we allowed this new interface to be blessed. If the specification and body of COLOR_MGR were recompiled and the procedure USER were relinked with these changes, then the type PALETTE would still be an array of 5 BOOLEAN values, the loop would still iterate through the values VIOLET (0) to ORANGE (4), but the invocation of the function P could return the value RED (5) which would index beyond the end of the PAL array! Runtime checks would not catch this error because, as far as the compiler is concerned, the function P's return type is the same as the index type of PAL so no runtime check would be emitted. The user's program would fail in a very mysterious way by overwriting whatever was allocated after the end of the array PAL (the variable B in this example). Similarly, the case statement would behave unpredictably if the function P returned a value of RED since there is no entry in the branch table for this value (since the compiler didn't expect there could be any value greater than ORANGE).

If the type COLORS had 8 values and then was changed to have 9, the size of variables of type COLORS would change from 3 bits to 4 bits, another source of mysterious program errors.

Blessing must maintain the run-time integrity of user's programs. The user's program would remain correct only if, after changing the type COLORS, the subprogram USER were forced to be recompiled. Therefore allowing COLOR_MGR to be blessed after a change to the type COLORS should not be allowed.

Similar problems would occur if we allowed a subtype range to be changed (e.g., from INTEGER range 0..4 to INTEGER range 0..5) and the interface blessed.

Further examples of interface changes that are incompatible (and are therefore not allowed) are:

- Deleting an existing basic declaration
- Changing an existing basic declaration:
  - Adding, deleting, reordering, or changing the type of a record field or formal subprogram parameter
  - Adding, deleting, or reordering enumeration literals of an enumeration type
  - Changing the element type of an array
  - Changing the base type or constraints of a subtype
- Changing the default value for a formal parameter or discriminant, or the value of a constant or named number
- Applying (or changing) a representation pragma or representation clause (except when the actual representation is not affected by the change)
- Reordering overloaded subprograms in a package

Although not all changes are considered compatible, interface blessing is often able to contain the effects of an incompatibly changed interface to the immediate users of this interface, thereby containing the effects of an unblessed change to the directly dependent CUs.

CONSISTENCY IN THE PRESENCE OF CHANGE

The programmer can take advantage of the multiple versions allowed by blessing, to install upwardly compatible interfaces with varying levels of support, all under the watchful eye of the Ada librarian and linker. Because multiple version/time-stamps are maintained in the interface and object files, these tools will catch any attempt to link together incompatible implementations, such as compiling against a new enhanced interface but linking with an older subset implementation. At the same time, these linking tools will allow you to compile with an old interface and link against a new compatibly blessed version of the object code.

BLESSING USED FOR PROJECT CONTROL AND SYSTEM RELEASE

The ability to compare interfaces has important applications for project management and maintenance. For example, as a project control tool, blessing can be told (by specifying the option **:bless=identical** to the Ada compiler) to allow only *identical* interfaces to be installed into a project library. Normally this would be done by restricting modify-access to the source files, but this restriction would also prohibit documentation changes. With blessing, source files can remain available while changes to the declarations and code can be detected and prohibited.

By shipping public interface files together with product source files, the regeneration of a product at a foreign site can be *guaranteed* to be identical to a baseline site's generation. This is vital for releasing Ada systems written by different vendors, to the same user.

By far the most important benefit from interface blessing is in generating product releases. By compiling with the blessing option, the compiler will *guarantee* that the new interface is indeed compatible with the existing interfaces of the previous release. Project management need not rely on manual inspection of source changes to verify compatibility — with blessing the checking is done automatically. Surprises caused by incorrectly perceived compatible changes will be immediately caught, an error message given, and the changes not allowed — the Ada library will not be updated with the incompatibly changed unit. Blessing solves the problems of making upward compatible updates to product interfaces by providing a powerful tool to verify that old and new interfaces are truly compatible.

CONCLUSION

The ability to keep Ada library semantics consistent, even in the presence of compatible interface changes, is vital not only for rapid product development, but also for product generation, maintenance, and releases. As our system project use has demonstrated at BiiN, the interface blessing techniques and tools we've developed are proven and efficient, and make sense in a world of changing interfaces.

REFERENCES

[1] Tichy, W.F. Smart Recompilation, *ACM Transactions on Programming Languages and Systems*, Vol. 8, No. 3, July 1986.

[2] Schwanke, R.W. & Kaiser, G.E. Smarter Recompilation, *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 4, October 1988.