Using ASIS to Generate C++ Bindings

Howard Ausden
Lockheed Martin TSS
9211 Corporate Blvd.
Rockville, MD 20850
1-301-640-2099
Howard.Ausden@lmco.com

Karl Nyberg
Grebyn Corporation
P. O. Box 47
Sterling, VA 20167-0047
1-703-406-4161
karl@nyberg.net

ABSTRACT
In this paper, we describe an approach to automatically creating C++ bindings to Ada libraries utilizing capabilities of the Ada Semantic Interface Specification (ASIS). We discuss language mapping issues and provide examples of usage of ASIS features during the implementation of a binding tool.

Categories and Subject Descriptors
D.2.7 **[Distribution, Maintenance, and Enhancement]**: Restructuring, reverse engineering and reengineering. F.3.2 **[Semantics of Programming Languages**]: Program analysis. I.2.2 **[Automatic Programming]**: Program synthesis; program transformation.

General Terms: Languages.

Keywords: Program Transformation, Cross-Language Libraries, Multiple Language Interfaces.

1. INTRODUCTION
The En Route Automation Modernization (ERAM) project is being undertaken by Lockheed Martin under the direction of the Federal Aviation Administration (FAA). ERAM will provide air traffic management automation services for the En Route domain at the twenty continental United States Air Route Traffic Control Centers (ARTCC). The software runs on International Business Machines (IBM) computers running the AIX operating system. The system, derived in part from an earlier project, the User Request Evaluation Tool (URET) has primarily been implemented in Ada, with additional client processes (primarily for display components) now being implemented in C++. Providing access to the underlying library services implemented in Ada for clients implemented in C++ provided the impetus for this effort. Automating the creation of language bindings became a consideration due to the quantity and fluidity of the underlying libraries during development. Utilizing ASIS as an infrastructure to support the development of a program transformation tool provided an opportunity to do that automation. (Because of the volume of already extant code, injection of a data driven approach, such as CORBA (Common Object Request Broker Architecture) [1] was not feasible to apply at this stage of the project's lifecycle.)

Generally, Ada programmers find themselves in the position of having to interface to libraries written in other languages, such as C, C++, FORTRAN and Java. In those instances scripts and / or tools (e.g., cbind [2]) are created to automate portions of the parsing of the library specifications, such as are contained in header files or intermediate object files [3]. In some cases, it is even possible to use language development tools for those languages, such as lex and yacc / bison that are designed to parse library specifications. The ASIS approach is different in that ASIS operates on an abstract view of the internal representation (as created by the compiler being used) of the program under analysis. This approach relieves the tool developer of any requirement to create a parser for the underlying software and allows attention to be focused on the generation of the library bindings rather than upon the creation of supporting technology for data access.

In the remainder of this paper, we discuss these language differences (at a high level, with a partial mapping between the languages), followed by a brief overview of ASIS and the implementation of the binding tool with examples of how ASIS capabilities have been used.

2. LANGUAGE COMPARISON
To a great degree Ada and C++ are comparable languages. The latest generation of Ada (Ada95) added features that made the language much more object oriented, and, as a result, allows easy mapping of language features into similar features in C++. In Table 1 we present some of the common features (additional detailed feature analysis and comparison can be seen in [4], [5] and [6]). Following the table we discuss how the mapping of some of these features is implemented.

Table 1. Comparison of Language Features

Feature	Ada	C++
Packaging / Name Space	Package	Namespace
Complex Typing	Private, abstract, tagged	Classes
Exceptions	Language, library and user defined	Library and user defined
Functions / procedures	Functions / procedures	Methods (as part of a class), functions and procedures

SIGAda'05, November 13–17, 2005, Atlanta, Georgia, USA.

2.1 Packaging / Name Space

Both Ada and C++ contain features for collecting items of a common nature in a single unit for managing the name space. Within Ada this feature is the package; with C++, the namespace. Both languages contain a mechanism for extending this feature through a "child" approach that allows the children to "inherit" from the parent entity.

As the requirement for developing a C++ interface was known from the inception of the project, the resulting Ada was designed with this in mind and was developed in a highly object-oriented approach. While no formal style guide or other published development approach was used to ensure the seamless integration of a C++ interface, the simplicity of achieving the result cannot be attributed to simple serendipity.

There were several instances where namespace issues were slightly more complicated than an obvious approach might expect. For example, within a C++ namespace, all enumeration literals must be unique, whereas within in Ada package, all enumeration literals must be unique within a type. For example, where Ada would permit the following enumeration declarations in a package:

```
type Stop_Light is (Red, Yellow, Green);
type Rainbow is (Red, Orange, Yellow,
                 Green, Blue, Indigo,
                 Violet);
```

the generation of the obviously equivalent C++ enumeration literals:

```
enum stop_light (RED, YELLOW, GREEN);

enum rainbow (RED, ORANGE, YELLOW, GREEN,
              BLUE, INDIGO, VIOLET);
```

will result in the **BOLDED** literals being flagged as duplicates by the C++ compiler. Several options existed for making the enumeration literals unique – mangling the literal, changing case, pre- or appending additional text or mapping to some other uniquely generated symbol. However, since this is text that the C++ client application programmer uses, it was felt that such changes should maintain meaningful names for the literals. Hence the Ada programmers were encouraged to change the names to ensure uniqueness and yet maintain meaning. This had the side benefit of keeping the literals the same in both languages as well. The above example could be solved by simply making the following change to the Ada enumeration and continuing to use the obvious mapping to C++ enumerations.

```
type Stop_Light is (Red_Light,
                    Green_Light,
                    Yellow_Light);
```

Additionally, at link time, it must be possible to uniquely resolve all names in use. When specifying the name in each language for crossing the language boundary interface (`pragma export` in Ada and `extern` in C++), the names must be globally unique. Prepending a version of the package / namespace name will ensure that no cross-package clashes exit. Within an individual package / namespace, conflicting link names due to overloaded functions / procedures / methods with different parameter profiles were simply disambiguated by appending sequentially increasing number suffixes ("`_01`", "`_02`", etc.) as needed. This approach was considered acceptable because none of the names were visible to either the client application programmer or the original library implementer, as the generated symbols were wholly contained within the wrapper routines.

2.2 Type Mapping

There are essentially three groups of data types that need to be mapped between the two languages. These are the "intrinsic" types (integer, boolean, strings, etc.), the "composite / structured" types (arrays / vectors, records / structs) and the "abstract" types (not only actual `abstract` types, but also variant records, discriminant, tagged and private types and classes).

The implementation of the type mapping can be thought of in two aspects – the specification and the implementation. The implementation is commonly attempted through the use of representation specifications on the Ada types to ensure that the data layout in the Ada implementation matches that of the C++ compiler being utilized on the project. While this approach worked effectively during the early use of Ada in creating interfaces to existing straightforward C libraries, the complications of discriminant records, variant types, union structures, arrays and vectors, null termination requirements of strings in C++ and the whole collection of issues associated with tagged and private types / classes has led us to consider alternative approaches to implementing the exchange of data across the language boundary. Additional project requirements for data, such as in simulations, messaging and logging during the operation of the system has provided the opportunity to consider additional formats. These implementation issues are described in greater detail in later sections.

2.2.1 Intrinsic Types

The intrinsic types match quite directly between the languages. However the semantics of some of the types such as integer subtypes only become evident at the time of the implementation of the language boundary. For example, where an Ada integer subtype such as:

```
subtype Teenage_Years is
    integer range 13 .. 19;
```

provides bounds information that the Ada compiler enforces on access and assignment, the corresponding C++ declaration:

```
typedef int teenage_years;
```

has no attendant range checking. Thus, to ensure that an object, or component of an object that is passed from C++ to Ada is within the range of the declared subtype) requires either checking the range of the object within C++ before crossing the language boundary or checking the `'Valid` attribute on the Ada side.

Using the principle of least surprise, checking the values in the C++ side of the binding before crossing the language boundary has been implemented.

2.2.2 Composite / Structured Types

The composite types, records and arrays, while also mapping quite well directly at the language level, suffer from even more complication during the implementation. Within Ada, arrays with non-static bounds are often maintained in memory preceded by "dope vectors" that contain information on the bounds of the array; within C++, no such information is carried along. This required a little more effort during the transmission of objects between the two languages (see below), but was otherwise straightforward.

2.2.3 Abstract Types

The "abstract" types in Ada consist of those for which either access to objects of the type has been limited (e.g., private types) to the functions and / or procedures provided or for which extensibility (inheritance) of the object is desired. These are mapped into C++ classes.

The mapping of variant records provided an engineering tradeoff. A case could be made to either map to a C++ union struct or to a class (both mappings were, in fact, implemented). As the C++ programmers on the project felt that the class approach was more natural, that selection was made.

2.3 Additional Language Boundary Issues

The primary focus of the foregoing discussion has been the creation of the C++ header file (the *.h file). This section discusses the issues associated with crossing the language boundary in terms of both the mapping of the functions / procedures and methods and the data and exceptions.

In order to isolate the C++ binding interface from the existing Ada packages, child packages were created to contain all the necessary interface information. Procedures were created that wrapped original procedures with an additional exception integer variable (to represent any exception generated by the original procedure) and those derived from functions had both an exception integer added as well as an OUT variable of the return type of the original function. Calls to the original function / procedure were further wrapped in an exception handler with an others clause to ensure no Ada exception would propagate across the language boundary. (An alternative, compiler-dependent, solution that cleared the execution stack and replaced it with C++ exception material was considered but not selected because of both its compiler dependency and a requirement to register all exceptions in advance, both cumbersome and a potentially error prone situation.)

The exception passing mechanism effectively operates as a UNIX "errno" feature, passing back an integer indicating the "success" or "failure" of the underlying operation. This integer is used at the language boundary to indicate the exception that has been raised (on the Ada side) and which should be thrown (on the C++ side). Each possible exception declared in the Ada package as well as each predefined and "with'ed" package exceptions are mapped to a positive integer; unspecified exceptions (caught by a where clause on the Ada side) are mapped to a negative number.

Such an approach provides more than may actually be necessary so when the final implementation (the package bodies) is available for analysis and determination, a reduction in the number of actual exceptions being raised may be possible.

2.4 Data "Conversion"

As mentioned earlier, the selection of a data passing mechanism was motivated in part by a requirement for data logging (for checkpointing, error logs, etc.) in the operational system. Other components of the system utilize a logging tool, which places restrictions on the components of the type that may be "tooled" with respect to use of this tool. Such "toolable" types may not include any components that contain variant records, discriminant types, tagged types, etc. Such logs and the records contained in them also suffered from lack of versioning control as changes to the underlying types were not always kept synchronized because of the manual effort required to update interface routines for the tool when type changes were made.

Utilizing ASIS to automate the generation of the interfaces makes it possible to both maintain versioning control and to overcome the limitations of the tooling and logging tools. It is possible to automatically create routines for generating and parsing data objects in either language into or out of a format that the other language can generate or use. Both binary (for internal use and performance) and readable ASCII (in an XML like or data "dump" style format) could be supported. At the moment, the only implementation is with the internal Collection mechanism mentioned below.

3. ASIS OVERVIEW

ASIS was developed as an international "secondary" standard (dependent upon the Ada standard):

> The Ada Semantic Interface Specification (ASIS) is an interface between an Ada environment as defined by ISO/IEC 8652 (the Ada Reference Manual) and any tool requiring information from this environment. An Ada environment includes valuable semantic and syntactic information. ASIS is an open and published callable interface which gives CASE tool and application developers access to this information. ASIS has been designed to be independent of underlying Ada environment implementations, thus supporting portability of software engineering tools while relieving tool developers from having to understand the complexities of an Ada environment's proprietary internal representation. [7]

ASIS is implemented as a set of public package specifications (an Application Program Interface, or API with compiler dependent bodies for individual platforms:

> The ASIS interface consists of a set of types, subtypes, and subprograms which provide a capability to query the Ada compilation environment for syntactic and semantic information. Package Asis is the root of the ASIS interface. It contains common types used throughout the ASIS interface. Important common abstractions include Context, Element, and Compilation_Unit. Type Context helps identify the compilation units considered to be analyzable as part of the Ada compilation environment. Type Element is an

abstraction of entities within a logical Ada syntax tree. Type Compilation_Unit is an abstraction for Ada compilation units. In addition, there are two sets of enumeration types called Element Kinds and Unit Kinds. Element Kinds are a set of enumeration types providing a mapping to the Ada syntax. Unit Kinds are a set of enumeration types describing the various kinds of compilation units. [8]

The basic feature that ASIS provides is the ability to abstractly access the pre-digested compiler internal information (similar to an Abstract Syntax Tree) on a program under consideration. An ASIS application can then be thought of as a report generator written against data in a database.

The primary ASIS interfaces used in this project include:

- Context – the basic program library containing the compilation units.
- Compilation Units – package specifications, containing context clauses ("with"s) and various declarations (types, functions, procedures).
- Declarations – functions and procedures to obtain types and parameter profile information.
- Definitions – functions and procedures to query type declarations for detailed information.

Further details on the development of ASIS and tutorials can be found at the ACM's SIGAda web site [9].

Snippets and discussion of ASIS-related code relevant to the binding generator will be interspersed throughout the presentation of the binding process in the remainder of this paper. (In some of these instances, the code has been stripped to only show the relevant portion of ASIS being utilized.)

4. BINDING TOOL

4.1 OVERVIEW

The creation of a C++ binding to Ada libraries breaks down naturally into several reasonably managed areas. These include the creation of the interface specifications (in C++, the *.h header files corresponding to the Ada specification files, commonly the *.ads files) and the implementation of such specification as well as a "wrapper" in both C++ containing `extern` declarations corresponding to the interface specifications that implements the calls to the underlying Ada with corresponding `pragma export` declarations.

In this effort, the development approach has called for the release of a compilable Application Program Interface (API) in both Ada and C++ with dummy "stub" implementations to allow client programmers to begin to compile and link their applications prior to the full implementation of the underlying libraries. In order to meet this requirement and to permit a compiler-independent approach to cross-language exception management, this implementation is divided into a C++ implementation wrapper and an Ada implementation wrapper (on "top" of the existing Ada packages). Thus, for any existing individual Ada package foo.ads (and potentially foo.adb), there will be generated six corresponding files:

1. foo.h – the C++ header
2. foo.cpp – the C++ implementation
3. foo-wrap.ads – the Ada wrapper interface specification
4. foo-wrap.adb – the Ada wrapper implementation
5. foo-wrap.h – the C++ wrapper interface specification
6. foo-wrap.cpp – the C++ wrapper implementation

This approach has the advantage of keeping the existing Ada library uncluttered by placing all information associated with the C++ binding in separate files. Thus, client applications continuing to be developed in Ada will not have any of the additional material required for the C++ binding included.

In the creation of the header files, some consideration was given as to whether to attempt to provide representation specifications (see LRM (13)) as a means of passing data between calls in the two languages. However, the project had already defined a separate Collection package / namespace, implemented in each language, for inter-language data exchange. It was decided to use this existing functionality rather than invent something new. Use of such a mechanism requires eight additional files for the packaging of the data and specification of the interfaces for both the "tooling" approach and the "serialization" approach:

7. foo-tool.h – the C++ header
8. foo-tool.cpp – the C++ implementation
9. foo-tool.ads – the Ada specification
10. foo-tool.adb – the Ada implementation
11. foo-serial.h – the C++ header
12. foo-serial.cpp – the C++ implementation
13. foo-serial.ads – the Ada specification
14. foo-serial.adb – the Ada implementation

Now, as should be quite obvious, the explosion in the number of files necessary to implement an interface in this manner could be quite a concern, especially as the number of packages and libraries increases. It should be noted, however, that the only file that is of concern to the C++ programmer is in fact the original foo.h file, as all the other files represent material that is "under the hood" of the binding. It might be possible, even desirable, to coalesce some of the other files together to reduce the quantity, but for simplicity of development of the binding generator by separation of concerns, this has not yet been attempted.

In the creation of a secondary language binding, you must always address the issue of the level of the binding: whether to stay true to the original language definition or whether to allow the programmer to operate in the natural development environment of the derived language. We have striven to allow the programmer to operate in the natural development environment of the derived language (C++). This decision implies that we draw a "line in the sand" when requiring language-specific libraries that is closest to the language of the derived language, rather than of the original language and, where necessary, do low level manipulation to achieve these more native types. For example, we defer to the C++ `string` and `ctype` libraries rather than require the C++

programmer to use bindings to `Ada.Strings` or `Ada.Characters.Handling` packages.

In a similar vein, other components within ERAM also have the requirement to provide both Ada and C++ interfaces. In those cases, where we might have created an automated interface to the underlying Ada, we have deferred to any available C++ interfaces where they already existed. In some cases, these are bindings to underlying Ada libraries and in others, actual complete implementations.

In several cases of the existing C++ interfaces, differing approaches had been used to create the C++ interface from the Ada itself. Thus, any changes at the API level of those interfaces would require modifications by the client application programmers. To accommodate these existing libraries, whether project- or language-specific, the binding tool approach was used during generation and textual substitution was implemented where possible to make such substitutions while manual editing was performed where necessary.

Once complete API generation had occurred, this investment of manual labor in editing (which included insertion of comments in the generated code, since they were not propagated from the original Ada and needed to be recast into the C++ paradigm) made developers unwilling to regenerate their entire API. In maintenance mode the generator, used in conjunction with a differencing too, allows developers to manually verify the matching of the Ada and C++ APIs.

4.2 HEADER FILES

For a small contrived example: (exceptions declared within a package are not included in this example for brevity, nor are more complicated data structures and interfaces). Note that we also are not interested or required to know what the functions or procedures are intended to accomplish – all implementation details in the body of the package are irrelevant at this juncture.

```
--
-- foo.ads
--
package Foo is

   subtype Teen_Years is
      integer range 13 .. 19;

   procedure Proc (I : in out Teen_Years);
   procedure Proc (I : in out Integer;
                   J : in out Integer);
   function Func (I, J : Integer)
      return Boolean;

end Foo;
```

Within the binding generator, the list of declarations is retrieved from the visible declarations and iteratively processed:

```
declare
   My_Declarations :
      Asis.Declarative_Item_List :=
         Asis.Declarations.
         Visible_Part_Declarative_Items
            (My_Element);
begin
   for I in My_Declarations'Range loop
      Process_Element (My_Declarations (I));
   end loop;
end;
```

The resulting individual declarations are then processed according to their kind:

```
procedure Process_Element
             (E : in Asis.Element) is
  D : Asis.Declaration_Kinds :=
      Asis.Elements.Declaration_Kind (E);
begin -- Process_Element
  case D is
    when Asis.Not_A_Declaration =>
      null;
    when Asis.A_Component_Declaration =>
      Process_A_Component_Declaration (E);
    when Asis.A_Constant_Declaration =>
      Process_A_Constant_Declaration (E);
     ...
  end case;
end;
```

Creating a header file:

```
#ifndef FOO_H
#define FOO_H

namespace foo {

typedef int teen_years;
const int teen_years_first = 13;
const int teen_years_last = 19;

void proc (teen_years & i);

void proc (integer & i,
           integer & j);

boolean func (integer i,);

} // end namespace foo
#endif // FOO_H
```

4.3 WRAPPER FILES

As mentioned earlier wrapper files were generated as child packages to the existing Ada source code in order to encapsulate the language boundary information and keep it separate from the existing implementation. For each Ada package specification (*.ads file), a corresponding wrapper child package in Ada (*-wrap.ads and *-wrap.adb) and C++ (*-wrap.h and *-wrap.cpp) were generated. This was accomplished in ASIS by walking the visible declarations of the package, generating components of each of the four wrapper files along the way.

```
My_Element : Asis.Element  :=
  Asis.Elements.Unit_Declaration (My_Unit);
Declarative_Item_List :
   Asis.Declarative_Item_List :=
     Asis.Declarations.
        Visible_Part_Declarative_Items
           (My_Element);
...

for I in Declarative_Item_List'Range loop
   case Asis.Elements.Element_Kind
      (Declarative_Item_List (I)) is
      when Asis.A_Declaration =>
         case Asis.Elements.
           Declaration_Kind
          (Declarative_Item_List (I)) is
         when Asis.A_Function_Declaration |
             Asis.A_Procedure_Declaration =>
            ...
         end case;
    end case;
end loop;
```

4.3.1 Ada Wrapper

```
--
-- foo-wrap.ads
--
package Foo.Wrap is

   procedure Proc (I : in out Teen_Years;
                   Except : out Integer);
   pragma export ("C", Proc,
     "foo_wrap_proc");

   procedure Proc (I : in out Integer;
                   J : in out Integer;
                   Except : out Integer);
   pragma export ("C", Proc,
      "foo_wrap_proc_01");

   procedure Func (I, J : Integer;
                   Result : out Boolean;
                   Except : out Integer);
   pragma export ("C", Func,
      "foo_wrap_func");

end Foo.Wrap;

--
-- foo-wrap.adb
--
package body Foo.Wrap is

   procedure Proc (I : in out Teen_Years;
                   Except : out Integer) is
   begin
      Except := 0;
      Proc (I);
   exception
      when others =>
         Except := -1;
   end Proc;

   procedure Proc (I : in out Integer;
                   J : in out Integer;
                   Except : out Integer) is
   begin
      Except := 0;
      Proc (I, J);
   exception
      when others =>
         Except := -1;
   end Proc;

   procedure Func (I, J : Integer;
                   Result : out Boolean;
                   Except : out Integer) is
   begin
      Except := 0;
      Result := Func (I, J);
   exception
      when others =>
         Except := -1;
   end Func;
end Foo.Wrap;
```

4.3.2 C++ Wrapper

```
//
// foo-wrap.h
//
namespace foo {
namespace wrap {

extern "C" {
   void foo_wrap_proc (int i,
                       int & except);
   void foo_wrap_proc_01 (int & i,
                          int & j,
                          int & except);
   void foo_wrap_func (int i,
                       int j,
                       boolean & result,
                       int & except);
};

void proc (teen_years & i);
void proc (int i,
           int j);
boolean func (int i,
           boolean & result);

} // end namespace wrap
} // end namespace foo

//
// foo-wrap.cpp
//
namespace foo {
namespace wrap {

   void proc (teen_years i)
   {
      int except;

      //
      // Note range checking on C++ side
      // BEFORE the call to Ada
      //
      if ((i < teen_years_first)
       || (i > teen_years_last))
      {
         throw Constraint_Error_Equivalent;
      }

      foo_wrap_proc (i, except);
      if (i != 0)
      {
         throw Corresponding_Error;
      }
   }

   void proc (int i, int j)
   {
      int except;

      foo_wrap_proc_01 (i, j, except);
      if (except != 0)
      {
         throw Corresponding_Error;
      }
   }

   boolean func (int i, j)
   {
      int except;
      boolean result;

      foo_wrap_func (i, j, result, except);
      if (except != 0)
      {
         throw Corresponding_Error;
      }
      return result;
   }

} // end namespace wrap
} // end namespace foo
```

4.4 TOOLING AND SERIALIZATION

In order to achieve data object passing between systems, the project utilized an existing set of utilities, collectively implementing what was known as a "tooling" approach. These utilities operated on the type definitions (either C++ or Ada) and created corresponding definitions in the other language which could then be inserted into files by the programmers. These types can then be used with a Collection package that managed data objects by the insertion of tag / value pairs into a serialized data object and transferred to another system (in another address space on the same computer or on another computer). An implementation of this approach was also developed for this project.

4.5 OTHER ISSUES

Generation of interfaces for packages with dependencies on other packages was accomplished by processing those packages that were included in context clauses ("with" clauses). A Clause_List containing the clauses for a particular unit would be iterated and interfaces for the corresponding packages also generated.

```
declare
   Clause_List : Asis.Context_Clause_List :=
     Asis.Elements.Context_Clause_Elements
        (My_Unit_Lists (I));
begin
   for Each_Clause in Clause_List'Range loop
      case Asis.Elements.Clause_Kind
         (Clause_List (Each_Clause)) is
         when Asis.A_With_Clause =>
            Process_Withs (
               Asis.Clauses.Clause_Names (
                  Clause_List
                     (Each_Clause)));
         when others =>
            null;
      end case;
   end loop;
end;
```

5. INTEGRATION INTO DEVELOPMENT

In order to run the tools and generate the bindings and associated files, the code upon which the bindings depend must be compiled first. Since the project uses a configuration management system, in order to perform automated builds, the bindings must themselves be generated and stored in the configuration management system as well, so that they can be retrieved during the build process, rather than generated and compiled during the build process. Because of additional requirements on documentation being placed within the delivered software, this is not a problem, since some manual editing of the generated files is allowed with this approach. Care must be taken when the base software is updated to compare generated (and now commented / edited) versions of the code.

6. SUMMARY

This paper has discussed the use of ASIS as a basis for a program transformation tool to automate the generation of C++ bindings to Ada packages. Issues associated with the language binding process have been interspersed with examples of ASIS-related code used to accomplish individual tasks.

7. ACKNOWLEDGMENTS

This work was performed under contract from the Federal Aviation Administration, DTFA01-03-C-00015. The support and review from Jeff O'Leary is especially noted and appreciated.

8. REFERENCES

[1] http://www.omg.org/gettingstarted/corbafaq.htm

[2] http://unicoi.kennesaw.edu/ase/ase02_02/tools/cbind/readme

[3] Emery, David E.; Mathis, Robert F.; and Nyberg, Karl A. "Automating the Ada Binding Process for Java - How Far Can We Go?'' Reliable Software Technologies: Proceedings of the Ada Europe 1998 conference, June 1998, Uppsala, Sweden.

[4] Jesper Jørgensen, A Comparison of the Object-Oriented Features of Ada 9X and C++, Proceedings of the 12th Ada-Europe International Conference, p.125-141, June 14-18, 1993

[5] http://www.adahome.com/articles/1997-03/ada_vs_cpp.html

[6] http://www.adaic.com/whyada/ada-vs-c/ada-vs-c.html

[7] http://www.acm.org/sigada/wg/asiswg/ASIS_Background.html

[8] http://www.acm.org/sigada/wg/asiswg/intro.html

[9] http://www.acm.org/sigada/wg/asiswg/asiswg.html