Data Sharing Between Ada and C/C++

Matt Mark
Lockheed Martin
9211 Corporate Blvd.
Rockville, MD 20850
301-640-2861
matt.mark@lmco.com

ABSTRACT
The En Route Automation Modernization (ERAM) program is a real-time Air Traffic Control (ATC) program being developed by Lockheed Martin Corporation. The ERAM program has high availability requirements, mission critical applications, and stringent response time requirements. The estimated size of the ERAM program is 1,300 KSLOC and includes primarily Ada, C, and C++ code. Legacy code being reused in ERAM is both Ada and C. This resulted in the need for several cross-language interfaces. Standard methods for passing binary data structures between the languages have been developed and will be discussed in this paper.

Categories and Subject Descriptors
D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering Language Contructs and Features. D.3.3 [**Language Constructs and Features**]: Data types and structures.

General Terms
Performance, Languages

Keywords
Cross-language Interface

1. INTRODUCTION
Lockheed Martin ATC programs prior to ERAM have had a limited number of cases where Ada and C source code shared data structures. Ensuring the structures used in both languages are equivalent had been done manually. Code commentary was normally included next to the data structure definition in both languages indicating that if the structure was changed, the equivalent structure in the other language must also be changed. This manual maintenance strategy is cumbersome at best.

The ERAM program has many more cases of data sharing between languages than our previous programs and a more rigorous approach was needed. For years, Lockheed Martin ATC programs have had utilities that use the Ada Semantic Interface Specification (ASIS) to evaluate Ada data structures and put information about them into a code-independent format[1]. This format is commonly referred to as a dictionary entry for the data structure. The dictionary entry includes the name of the type, name of fields within the type and the type names of the fields, the bit offsets of each field, the range of each field, etc. The process of creating a dictionary entry is commonly referred to as "tooling".

Prior to ERAM, the tooling of data structures into dictionaries was mainly done for use by support (offline) software that did not have a direct interface to operational code. Two examples of this use are:

1. Utility programs that convert text to binary files used as input (adaptation) data files for operational code, and
2. Utility programs that convert binary data structures which are recorded for system analysis. Dictionary entries are used to interpret the binary data recorded and produce text.

For ERAM, the ability to create dictionary entries from C and C++ types was developed.

The approach taken to sharing of data between languages is to write utilities that:

1. Create compilable code containing data structures that have been tooled (referred to as type generation).
2. Compare two tooled data structures to determine if they are binary compatible (referred to as type matching).

With these utilities in place, the following steps are followed to address cross-language sharing of data:

- Define the structure to be shared between languages in Ada, C, or C++.
- Tool the type to create a dictionary entry.
- Run the type generator program which will create data structures in both Ada and C, regardless of the language of the original type.
- Use the "opposite language" data structure where needed.
- Tool the "opposite language" data structure.
- At system build time, the type matching tool is used to ensure compatibility of the two types.

The steps and more details about the type generator and type matching utilities will be discussed further in this paper. Also,

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

other applications of these utilities will also be discussed, including platform and/or compiler migration.

Alternatives to sharing data structures between languages include converting data to an ASCII format such as XML. XML-based schemes have the advantage of being compiler, language and machine architecture independent, but at some cost. The scheme of sharing data structures detailed in this paper was developed mainly for performance considerations; the structures being shared are included in Ada and C++ APIs that are expected to be called hundreds of times per second. This scheme is a relatively low-maintenance approach to achieve the necessary run-time performance goals.

2. TYPE GENERATOR

The type generator is only an aide to help in producing binary compatible types. The developer that maintains the code produced by the type generator has the option of changing the types. This might be done for readability (the type generator selects names which the maintainer may wish to change) or to combine types.

The type generator produces an Ada package and C header file for each type given as input. Two types being run through the type generator could, for example, share underlying data structures which should be defined once in source code used by both higher level types.

The Ada package name and the C header file name used in the generated source code files are parameters to the type generator tool.

2.1 Limitations

There are limitations to the types that can be produced by the type generator. In order to share structures between Ada and C, only structures that had common attributes between the two languages can be processed. Also, since dictionary entries are being used as input, additional limitations are imposed.

These limitations include:

- Ada constant and C #define statements are not supported. Information about constants is not included in dictionary entries and these values are not included in generated source code.
- Ada variant records (with variant parts or array bound discriminant) are not supported.
- C unions are not supported.
- Only floating point numbers used in C are supported. This means only 4 byte floats with 7 digits of precision and 8 byte floats with 15 digits of precision are supported. Ada fixed types are not supported.
- Integer types that are a multiple of 8 bits in size are supported. In Ada, a type that has a range from 0 .. 10 can be defined to be 4 bits. Such a type cannot be defined in C. Such a type can be included in a record in both Ada and C, but a type that is simply an integer of 4 bits cannot be defined in both languages.
- Floating point numbers and strings must be on byte boundaries. In Ada, these fields can be specified to not be on byte boundaries via rep-spec. This cannot be done in C.
- Arrays whose element size is not a multiple of 8 bits (a whole number of bytes) are not supported. For example, in Ada an array of two bit elements can be defined. This cannot be done in C.
    - A special case is handled, an array of bits that is a multiple of 8 in size. In this case, the generated C structure contains an array of characters whose size matches the Ada bit array.

2.2 Special Cases

There are several cases where the generated types differ from the original. The types generated and the original types are binary compatible (i.e. all primitive fields in the original type exist in the generated type as the same bit locations). We decided to put as few restrictions on the original type as possible. Because of this, the type generator program needed to handle some special cases.

2.2.1 Subtypes

Dictionary entries do not contain information about subtypes. The generated code will contain types that are not subtypes. For example, given the following Ada type Rec_T:

```
package Test is
  subtype Int1_T is Integer range 0 .. 10;
  subtype Int2_T is Int1_T  range 0 .. 5;
  type Rec_T is record
    I1 : Int1_T;
    I2 : Int2_T;
  end record;
end Test;
```

The type generator produces the following Ada package spec:

```
package Test1 is
  type Test_Int1_T is range 0 .. 10;
  type Test_Int2_T is range 0 .. 5;
  type Test_Rec_T is record
    I1 : Test_Int1_T;
    I2 : Test_Int2_T;
  end record;
  for Test_Rec_T use record
    I1 at 0 range 0 .. 31;
    I2 at 0 range 32 .. 63;
  end record;
  for Test_Rec_T'Size use 64;
end Test1;
```

Generated type TEST_INT1_T is not a subtype of Integer and TEST_INT2_T is not a subtype of TEST_INT1_T. This behavior would be difficult to change since it cannot be determined from the dictionary entry if a type is a subtype of another. However, the generated type specifies integer fields consuming the same bits as the original type.

2.2.2 Case Sensitivity

Ada is not case sensitive while C is. In C, a field in a record can have the same name as the type, differing only by case. This cannot be done in Ada. We decided to handle this situation by appending "_F" to the field name to ensure uniqueness. For example, given the following C type Rec_T:

```
typedef unsigned char  Acid[8];
typedef struct {                    `
  Acid      acid;
  } Rec_T;
```

The type generator produces the following Ada package spec:

```
package Test1 is
  type Acid_Index1 is range 0 .. 7;
  subtype Unsigned_Char is Character;

  type Acid is array (Acid_Index1) of
     Unsigned_Char;
  for Acid'Component_Size use 8;
  for Acid'Size use 64;

  type Rec_T is record
    Acid_F : Acid;
  end record;

  for Rec_T use record
    Acid_F at 0 range 0 .. 63;
  end record;
  for Rec_T'Size use 64;
end Test1;
```

This behavior could easily be changed to reject types containing fields like these.

2.2.3 Language Keywords

Since Ada and C have different keywords, some type and field names in one language are not valid in the other. The type generator has some special code for keywords. For example, in the following type, field delay is a keyword in Ada.

```
typedef struct {
  signed    int     delay;
  } Rec_T;
```

The type generator produces the following Ada package spec:

```
package Test1 is
  subtype Int is Integer;
  type Rec_T is record
    Delay_F : Int;
  end record;

  for Rec_T use record
    Delay_F at 0 range 0 .. 31;
  end record;
  for Rec_T'Size use 32;
end Test1;
```

This behavior could easily be changed to reject types containing fields that are keywords in either language, but again, we decided to force different field names in this case.

2.2.4 C Enumeration Literals

C does not allow an enumeration literal to be used more than once in a header file. The type generator has some special code to check for this case and ensure the all enumeration literals differ in the generated C header file. For example, in the following type, field DOWN is used in two enumeration types. This cannot be done in C.

```
package Test is
  type Enum1_T is (Up, Down);
  type Enum2_T is (Down, Load,
                   Degraded, Normal);
  type Rec_T is record
    E1 : Enum1_T;
    E2 : Enum2_T;
  end record;
end Test;
```

The type generator produces the following C header file:

```
typedef enum e_TEST_ENUM1_T
  {UP, DOWN};
typedef enum e_TEST_ENUM2_T
  {DOWN1, LOAD, DEGRADED, NORMAL};
typedef struct
  {
   unsigned  char    E1;
   unsigned  char    E2;
  } TEST_REC_T;
```

This behavior could easily be changed to reject types containing multiple enumerations with the same value. Again, we decided to force different enumeration literals.

2.2.5 C Enumerations in Records

In C code created by the type generator, enumerations are not included as fields in records. Instead, an integer field of the appropriate size is put in its place. The reason for this is that in C, enumerations cannot be specified to use fewer than eight bits while in Ada they can. For simplicity of the type generator program, we decided to always include integers in C structures where the original type had an enumeration. The previous example has been changed so that the enumerations in the original type are specified to be fewer than eight bits.

```
package Test is
  type Enum1_T is (Up, Down);
  type Enum2_T is (Down, Load,
                   Degraded, Normal);

  type Rec_T is record
    E1 : Enum1_T;
    E2 : Enum2_T;
  end record;

  for Rec_T use record
    E1 at 0  range  0 ..  3;
    E2 at 0  range  4 ..  7;
  end record;
end Test;
```

The type generator produces the following C header file:

```
  typedef enum e_TEST_ENUM1_T
    {UP, DOWN};
  typedef enum e_TEST_ENUM2_T
    {DOWN1, LOAD, DEGRADED, NORMAL};
  typedef struct
  {
   unsigned  int     E1: 4;
   unsigned  int     E2: 4;
  } TEST_REC_T;
```

This behavior is needed to process enumerations contained in structures, where the enumeration consumes fewer than eight bits.

2.2.6 Array Index Types

For simplicity of the type generator program, array indices in Ada generated types are always defined as integers. This is done regardless of the original array index type (two discrete values, an integer type, an enumeration type, etc). For C, the array is simply declared to have the correct number of elements.

Example 1: The below type is an array with a range of 1 .. Max_Elem (where Max_Elem is 3).

```
package Test is
  Max_Elem : constant := 3;
  type Arr1_T is array
              (1 .. Max_Elem) of Integer;
end Test;
```

The type generator produces the following Ada package spec:

```
package Test1 is
  type Test_Arr1_T_Index1 is range 1 .. 3;
  type Test_Arr1_T is array
          (Test_Arr1_T_Index1) of Integer;
  for Test_Arr1_T'Component_Size use 32;
  for Test_Arr1_T'Size use 96;
end Test1;
```

The following is the contents of the C header file produced:

```
  typedef INTEGER  TEST_ARR1_T[3];
```

Neither the generated Ada or C code contains the constant Max_Elem. Constants values are not in the dictionaries.

Example 2: The below type is an array with an enumeration, Enum_T, as the index.

```
package Test is
  type Enum_T is (Up, Down);
  type Arr1_T is array (Enum_T) of Integer;
end Test;
```

The type generator produces the following Ada package spec:

```
package Test1 is
  type Test_Arr1_T_Index1 is range 0 .. 1;
  type Test_Arr1_T is array
           (Test_Arr1_T_Index1) of Integer;
  for Test_Arr1_T'Component_Size use 32;
  for Test_Arr1_T'Size use 64;
end Test1;
```

The following is the contents of the C header file produced:

```
    typedef INTEGER  TEST_ARR1_T[2];
```

Neither the generated Ada or C code contains the enumeration Enum_T. With a change to the type generator, it would be

possible to use Enum_T as the array index in the generated Ada code, but this has not been done.

Example 3: The below type is an array with an integer type, Int_T, as the index.

```
package Test is
  type Int_T  is range 5 .. 10;
  type Arr1_T is array (Int_T) of Integer;
end Test;
```

The type generator produces the following Ada package spec:

```
package Test1 is
  type Test_Arr1_T_Index1 is range 5 .. 10;
  type Test_Arr1_T is array
           (Test_Arr1_T_Index1) of Integer;
  for Test_Arr1_T'Component_Size use 32;
  for Test_Arr1_T'Size use 192;
end Test1;
```

The following is the contents of the C header file produced:

```
  typedef INTEGER  TEST_ARR1_T[6];
```

As with the enumeration type above, neither the generated Ada nor C code contains the type Int_T.

2.2.7 Anonymous Types as Fields in Records

Ada allows an unconstrained array. Ada allows the bounds on these arrays used as a field in a record to be specified when record is declared. The tool that builds dictionary entries only supports strings as fields within records this way. In the generated type, the strings are not anonymous. For example:

```
package Test is
  type Rec1_T is record
    S1 : String(1..10);
    S2 : String(1..5);
  end record;
end Test;
```

The type generator produces the following Ada spec:

```
package Test1 is
  subtype Str_Type_Len5_T is String
                                (1 .. 5);
  subtype Str_Type_Len10_T is String
                                (1 .. 10);
  type Test_Rec1_T is record
    S2 : Str_Type_Len5_T;
    S1 : Str_Type_Len10_T;
  end record;

  for Test_Rec1_T use record
    S2 at 0 range 0 .. 39;
    S1 at 0 range 40 .. 119;
  end record;
  for Test_Rec1_T'Size use 128;
end Test1;
```

The type generator produces the following C header file:

```
typedef struct
  {
   char           S2[5];
   char           S1[10];
  } TEST_REC1_T;
```

2.3 Examples

2.3.1 Example of a Simple Record

Below is a simple type which has been run through the type generator. In this example, the type is a record which contains a string and an enumeration.

Original type: Note that the fields in the record are declared in a different package (Rel) from the type itself (Rel_Msg). The type to be generated is Del_Rel_T. It does not have a rep-spec.

```
package Rel is
  subtype Name_T is String (1 .. 8);
  type Kind_T is (None, Os, Firm, Appl);
end Rel;

with Rel;
package Rel_Msg is
  type Delete_Rel_T is record
    Kind : Rel.Kind_T;
    Name : Rel.Name_T;
  end record;
end Rel_Msg;
```

Generated Ada source code: All types the type being generated are declared in one package. Any '.' found in type names have been converted to '_'. For example, the enumeration type in the original structure is Rel.Kind_T. In the newly created Ada, the type name is Rel_Kind_T. This was done for simplicity of the type generator tool (only generates a single Ada file in a single

package spec). Also, the newly created type has a rep-spec for the field locations and the type size to match the original type. If the generated type contained arrays, they would also have a rep-spec for the array's element size.

```
package Test2 is
  type Rel_Kind_T is
           (None, Os, Firm, Appl);
  subtype Rel_Name_T is String (1 .. 8);

  type Rel_Msg_Delete_Rel_T is record
    Kind : Rel_Kind_T;
    Name : Rel_Name_T;
  end record;

  for Rel_Msg_Delete_Rel_T use record
    Kind at 0 range 0 .. 7;
    Name at 0 range 8 .. 71;
  end record;
  for Rel_Msg_Delete_Rel_T'Size use 72;
end Test2;
```

Generated C source code: As with the Ada, all types referenced are contained in a single header file. The new C type would have record fields ordered to match the bit layout of the original type. Also, pad fields would be inserted if necessary.

```
  typedef enum e_REL_KIND_T
    {NONE, OS, FIRM, APPL
    }
    REL_KIND_T;

  typedef struct
    {unsigned  char    KIND;
     char              NAME[8];
    }
    REL_MSG_DELETE_REL_T;
```

2.3.2 More Complicated Example

The next example is a bit more complicated, containing a record with more fields, an array of records, and a rep-spec that results in unused space. The generated C code will contain pad fields to fill this gap.

Original type:

```
package Hw is
  subtype Hw_Type_T is String (1 .. 5);
  Hw_Stat_Unknown : constant Integer := 0;
  Hw_Stat_Down    : constant Integer := 4;
  subtype Hw_Status_T is Integer range
            Hw_Stat_Unknown .. Hw_Stat_Down;

  type Hw_Entry_T is record
    Hw_Type    : Hw_Type_T;
    Hw_Id      : Integer;
    Hw_Status  : Hw_Status_T;
    Time       : Long_Float;
    Cpu_Avg    : Integer;
    Cpu_Max    : Integer;
    Io_Mb_In   : Integer;
    Io_Mb_Out  : Integer;
    Mem_Reads  : Integer;
    Mem_Writes : Integer;
    Mem_Other  : Integer;
  end record;

  for Hw_Entry_T use record
    Time       at 0  range 0 .. 63;
    Hw_Id      at 8  range 0 .. 31;
    Hw_Status  at 12 range 0 .. 31;
    Cpu_Avg    at 16 range 0 .. 31;
    Cpu_Max    at 20 range 0 .. 31;
    Io_Mb_In   at 24 range 0 .. 31;
    Io_Mb_Out  at 28 range 0 .. 31;
    Mem_Reads  at 32 range 0 .. 31;
    Mem_Writes at 36 range 0 .. 31;
    Mem_Other  at 40 range 0 .. 31;
    -- gap           at 44 range 0 .. 55;
    Hw_Type at 51 range 0 .. 39;
  end record;

  type Hw_Table_T is array (1 .. 20)
                                of Hw_Entry_T;
end Hw;
```

Generated Ada source code: Due to naming convention, the names in this type end up being odd (Hw_Hw_Table_T).

```
package Test3 is
  type Hw_Hw_Table_T_Index1 is range 1 ..
20;
  subtype Hw_Hw_Type_T is String (1 .. 5);
```

```
  type Hw_Hw_Status_T is range 0 .. 4;

  type Hw_Hw_Entry_T is  record
    Hw_Type    : Hw_Hw_Type_T;
    Hw_Id      : Integer;
    Hw_Status  : Hw_Hw_Status_T;
    Time       : Long_Float;
    Cpu_Avg    : Integer;
    Cpu_Max    : Integer;
    Io_Mb_In   : Integer;
    Io_Mb_Out  : Integer;
    Mem_Reads  : Integer;
    Mem_Writes : Integer;
    Mem_Other  : Integer;
  end record;
  for Hw_Hw_Entry_T use record
    Hw_Type    at 0 range 408 .. 447;
    Hw_Id      at 0 range 64 .. 95;
    Hw_Status  at 0 range 96 .. 127;
    Time       at 0 range 0 .. 63;
    Cpu_Avg    at 0 range 128 .. 159;
    Cpu_Max    at 0 range 160 .. 191;
    Io_Mb_In   at 0 range 192 .. 223;
    Io_Mb_Out  at 0 range 224 .. 255;
    Mem_Reads  at 0 range 256 .. 287;
    Mem_Writes at 0 range 288 .. 319;
    Mem_Other  at 0 range 320 .. 351;
  end record;
for Hw_Hw_Entry_T'Size use 448;


  type Hw_Hw_Table_T is array
    (Hw_Hw_Table_T_Index1) of Hw_Hw_Entry_T;
  for Hw_Hw_Table_T'Component_Size use 448;
  for Hw_Hw_Table_T'Size use 8_960;
end Test3;
```

Generated C source code: Pad fields contain the following as part of the field's name "Typegen_Added_Pad_Field".

```
typedef struct {
  double            TIME;
  signed    int     HW_ID;
  unsigned  int     HW_STATUS;
  signed    int     CPU_AVG;
  signed    int     CPU_MAX;
  signed    int     IO_MB_IN;
  signed    int     IO_MB_OUT;
  signed    int     MEM_READS;
  signed    int     MEM_WRITES;
  signed    int     MEM_OTHER;
  unsigned  char  Typegen_Added_Pad_Field0;
  unsigned  char  Typegen_Added_Pad_Field1;
  unsigned  char  Typegen_Added_Pad_Field2;
  unsigned  char  Typegen_Added_Pad_Field3;
  unsigned  char  Typegen_Added_Pad_Field4;
  unsigned  char  Typegen_Added_Pad_Field5;
  unsigned  char  Typegen_Added_Pad_Field6;
  char           HW_TYPE[5];
  } HW_HW_ENTRY_T;


typedef HW_HW_ENTRY_T  HW_HW_TABLE_T[20];
```

2.3.3 Original Type Defined in C

In the following example, the original type is defined in C. Again, the generated C code includes pad fields; compiler alignment is not assumed.

Original type:

```
#include "/usr/include/time.h"

typedef unsigned char  Acid[8];
typedef unsigned char  Tail_no[10];
typedef struct
  {unsigned  char    version;
  signed    int     pcno;
  signed    short   orig;
  signed    short   dest;
  signed    int     suborig;
  signed    int     subdest;
  signed    short   sysorig;
  struct   timespec trtm;
  struct   timespec tmstp;
  Acid              acid;
  Tail_no           Tail;
 }
 msghd;
```

Generated Ada source code: Character arrays in C type are created as arrays in Ada, not strings. This is an example of why we decided to allow the generated code to be modified. Modification allows for more natural definitions in each language. If the character array were changed to a string in the Ada type, the type matching program treats an array of characters as a compatible with a string of the same size.

```
package Test4 is
  subtype Unsigned_Char is Character;
  subtype Int           is Integer;
  type Short is range -32_768 .. 32_767;
  subtype Long is Integer;

  type Timespec is record
    Tv_Sec  : Int;
    Tv_Nsec : Long;
  end record;
  for Timespec use record
    Tv_Sec  at 0 range 0 .. 31;
    Tv_Nsec at 0 range 32 .. 63;
  end record;
  for Timespec'Size use 64;

  type Acid_Index1    is range 0 .. 7;
  type Tail_No_Index1 is range 0 .. 9;

  type Acid is array (Acid_Index1)
                      of Unsigned_Char;
  for Acid'Component_Size use 8;
  for Acid'Size use 64;

  type Tail_No is array
         (Tail_No_Index1) of Unsigned_Char;
  for Tail_No'Component_Size use 8;
  for Tail_No'Size use 80;

  type Msghd is record
    Version : Unsigned_Char;
    Pcno    : Int;
    Orig    : Short;
    Dest    : Short;
    Suborig : Int;
    Subdest : Int;
    Sysorig : Short;
    Trtm    : Timespec;
    Tmstp   : Timespec;
    Acid_F  : Acid;
    Tail    : Tail_No;
  end record;
  for Msghd use  record
    Version at 0 range 0 .. 7;
    Pcno    at 0 range 32 .. 63;
    Orig    at 0 range 64 .. 79;
    Dest    at 0 range 80 .. 95;
    Suborig at 0 range 96 .. 127;
    Subdest at 0 range 128 .. 159;
    Sysorig at 0 range 160 .. 175;
    Trtm    at 0 range 192 .. 255;
    Tmstp   at 0 range 256 .. 319;
    Acid_F  at 0 range 320 .. 383;
    Tail    at 0 range 384 .. 463;
  end record;
  for Msghd'Size use 480;
end Test4;
```

Generated C source code: Type referenced from /usr/include/time.h is included in this header file.

```
typedef struct
  {signed    int     tv_sec;
   signed    int     tv_nsec;
  }
  timespec;

typedef struct
  {unsigned  char    version;
   unsigned  char  Typegen_Added_Pad_Field0;
   unsigned  char  Typegen_Added_Pad_Field1;
   unsigned  char  Typegen_Added_Pad_Field2;
   signed    int     pcno;
   signed    short   orig;
   signed    short   dest;
   signed    int     suborig;
   signed    int     subdest;
   signed    short   sysorig;
   unsigned  char  Typegen_Added_Pad_Field3;
   unsigned  char  Typegen_Added_Pad_Field4;
   timespec   trtm;
```

```
 timespec   tmstp;
 unsigned char   acid[8];
 unsigned char   Tail[10];
}
msghd;
```

3. TYPE MATCHING

As mentioned earlier, the type generator program written is an aide for application developers that want to define types in Ada and C/C++ that are binary compatible. We decided to allow generated types to be modified by developers. An option could have been to have a scheme where types that were to match another in a different language were automatically generated at system build time. This presented complications. As has been shown, the naming convention of fields and types is not as the developer would have originally written. Also, generated types as currently produced are completely defined in a single package or header file and that can lead to duplication of lower level types.

Since we wanted to allow types to be modified by the owning developer, we wanted to automate checks that the types in the different languages that should match one another did.

The developer that maintains Ada and C/C++ types that need to be binary compatible must tool these types into dictionary entries (there are tools in place to do this), putting them in a language independent format. The type matching tool will iterate through the two types ensuring they are binary compatible.

To be binary compatible, the types being compared and all their subcomponents must meet the following conditions. The term "bit location" below refers to the relative bit offsets at which a field within a structure or array is allocated in memory. This includes start byte, start bit, end byte, and end bit.

- For array types and fields, the bit locations, the number of elements, and the element sizes, must match.
- For record types and fields, the bit locations must match.
- For integer fields, the bit locations must match and the types must have overlapping ranges.
- For floating point fields, the bit locations must match and the types must have overlapping ranges.
- For string fields, the bit locations must match or must match the bit location of a character array.
- For enumeration fields, the bit locations must match. An enumeration field in one type matches an integer field in the other type, provided that the bit locations of these fields match. If the fields in both types are enumerations, the number of enumeration values must match. Furthermore, the names of corresponding enumeration literals must either be the same or one must be a sub-string of the other.
- Field names containing "_pad_" will be ignored. This check is not case sensitive. These fields are assumed to be pad fields (for example, fields in a C structure used to consume space in order to match an Ada type whose rep-spec indicates that there are gaps between fields).
- Field names do not need to match. However, the type matching program can optionally check for field names to match.

At system build time, the type matching tool is run against all pairs of types developers have indicated should match one another. Reports are generated when these types do not match and changes are made before the system is handed off to the integration team.

4. OTHER APPLICATIONS

The initial focus of writing type generator and type matching programs was cross-language compatibility. Other possible applications for such a scheme are:

- Systems that transition to new compilers or platforms and go through periods where to old and new co-exist. Data structures passed between "new" and "old" software can be tooled, and type generator can then create binary compatible types. Since it creates types in Ada and C/C++, this scheme can be used to ensure compatibility of two types in the same language built with different compilers or on different platforms.
- Sharing data structures between support and operational software. We have support software data analysis programs that take recorded data structures and characterize system behavior. These programs have traditionally been written independent of the operational software. Interpretation of recorded data structures has been done forcing the data structure to be defined in a common area (the disadvantage of this approach is that the data structures may not be encapsulated as they would have been otherwise) or making queries of dictionary entries (the disadvantage of this approach is that resulting source code is more cumbersome to write than if the type was available to code against). Using the type generator, each area can have their own definition of the same type and build time checks ensure they match one another.

5. CONCLUSION

It is not only possible, but has also been useful to create Ada and C compatible types from dictionaries (containing data structures in a language-independent format). Maintenance effort is reduced with the help of automation. Performance is enhanced by the exchange of binary structures (i.e., by avoiding conversion of the structures to and from a language-independent format, such as XML). Compatibility between Ada and corresponding C/C++ data structures is verified at system build time to catch errors before handoff to test organizations.

6. ACKNOWLEDGMENTS

This work was performed under contract from the Federal Aviation Administration, DTFA01-03-C-00015 with the support and review from Jeff O'Leary, ERAM Product Team Software Lead, Federal Aviation Administration.

7. REFERENCE

[1] Mike Glasgow, Donna Hepner, and Richard Schmidt, Implementing a Table-Driven Types Dictionary Service in Ada, 1992