Document Generation using ASIS Tools

Steven V. Hovater[1]

Rational Software
Lexington, Massachusetts, USA
svh@rational.com

Abstract. The Ada Semantic Interface Specification offers a unique capability to Ada development projects: the ability to construct tools that navigate through the semantic network formed by one's compiled code. In particular, producing accurate documentation based upon existing source code is a possibility; if one can identify the composite types that act as messages between different parts of one's system, then it is entirely feasible to generate an interface description document based upon the actual content of the code, thus automating what has been historically, a tedious process. ...

1 The Problem Space

One way to insure consistency between software artifacts (e.g. documentation and as-built interfaces in the software) is to use the software itself as the basis for the documentation. The phrase *self-documenting code* has been in widespread use for decades, sometimes taken seriously, other times in levity.

The Ada Semantic Interface Specification (ASIS) enables the creation of tools that can extract hard-to-research information from (compiled) Ada source code. This paper considers one such tool that uses the semantic information accessible from ASIS calls.

MIL-STD 498 (and later, ANSI J-016) proscribes in its data item description (DID) for the Interface Design Description (IDD), that the data elements that form the interface be documented to include:

1. name/identifier
2. data type (alphanumeric, integer, etc.)
3. size and format
4. units of measurement
5. range or enumeration of possible values
6. accuracy (how correct)
7. precision (number of significant digits)

Projects developing to MIL-STD 498 will first produce an Interface Requirements Specification (IRS), defining the information (from a high level) being exchanged between the interfaces. The software design then implements these requirements. The IDD is the documentation for the interfaces as realized in the software design; production of an IDD is generally a manual process involving considerable labor.

2 Proof of Concept

2.1 Theory

The Ada Semantic Interface Specification (ASIS)[1] is a programmatic interface which enables tools to be written that can navigate the compiler-generated semantic network (DIANA[2]) representative of the Ada source code.

The Ada programming language provides the record type as a means of aggregating disparately-typed components within a single structure. There are no requirements in the Ada Language Reference Manual that compiler vendors ensure their product provides identical memory layout of components within record types between compiler versions.

The Ada language also provides the record representation clause, a means for the programmer to specify the memory layout for objects declared of the record type. Thus programmers can ensure the components within their records conform to a predictable, defined memory layout. The representation clause can also ensure data portability between different hardware architectures.

Our goal was to create an application that would combine these concepts to generate Interface Design Description documentation. For record types or record components that require additional documentation beyond what is obtainable via programmatic examination of the compiler-generated semantic information, the application will extract (optional) annotations associated with those components or record types.

2.2 Practice

The initial work began by examining the information produced by running the Ada Analyzer[3] report "Display Expanded Type Structure". There we observed that the Ada Analyzer could navigate through the type structure, and collect information such as the base type, size, and sub-components.

The next step began with the creation of a very crude stand-alone prototype in Ada83 using the ASIS interface provided by Rational's Apex product.

The prototype took a record type declaration and decomposed it into the record component names. Our next iteration extended the prototype to also display the record component type names. Per our requirements, the application would be recursive in the sense that when we encountered a record type in the component list, the application would display the component record type component's components, and so forth, so that all record types were fully decomposed into scalar (e.g. non-record) types. Furthermore, each scalar type would be reduced (if appropriate) to its base type and constraints.

From a documentation viewpoint, the top-level record should be at a section level, and any record types found during the navigation of the top-level record

[1] Recently approved as an ISO standard, ISO/IEC 15291:1999.
[2] Descriptive Intermediate Attributed Notation for Ada.
[3] Ada Analyzer is a trademark of LittleTree Consulting.

component should be represented in subsections following the description of the top-level record component. This represented a design challenge - the most straightforward coding of the ASIS record decomposition routine would produce a nested table of record components, not a series of individual tables related by parent-child relationships.

The solution to this challenge was to create an intermediate memory-based structure which had entries corresponding to the top-level components. For components whose type was also a record type, a link for that component would be created to a new instance of the intermediate structure, the child record type components recorded, and ultimately, execution would return to the top level record component navigation. Thus, the recursive ASIS program could be written, but the results would be stored in the appropriate area in the intermediate representation, allowing complete tables to be constructed, with the nesting represented by the links.

With proof that we could indeed extract the information we needed, we then considered whether we should build a stand-alone tool, or create a new customization to the Ada Analyzer. We opted for the quickest route - that of using the framework already provided by LittleTree's product, which would allow us to use the wealth of utilities that support the Ada Analyzer.

The prototype was then frozen, and for our next iteration, we began implementation of the tool as an Ada Analyzer extension.

3 Production

3.1 Structure

Since at the top-level we are concerned with only record structures, the application was designed to accept only record structures at the initial point of the ASIS navigation code. There are two flavors of record types that we considered (in ASIS terms):

1. A_Record_Type_Definition
2. A_Derived_With_Record_Type_Definition

Furthermore, there was the requirement that the tool be able to handle discriminated record types. For components of discriminated record types, the constraint upon the discriminated record type (required by Ada for the declaration of a component with a discriminated record type) is used as the default value for the discriminated record decomposition. However, top-level discriminated record types offered a challenge.

Since the intention of the tool is that it run in a batch mode, the most practical way to handle top-level discriminated record types was to impose the restriction that any top-level discriminated record types have default values for their discriminants. Hence, if the tool encounters a discriminated record type at the top level, it assumes the default values for the discriminated record type. Top-level record types with no default discriminants are reported to the user as an error condition.

Another restriction placed on tool usage is that if the users wanted to get a bitmap of the structure table for a record type, the type must have a representation specification.

The Ada language has a rich support for representation clauses, with the interesting complications for our implementation efforts:

1. Not all components of a record need be rep spec'd to have valid code.
2. The order of the declarations in a representation clause may differ from the order of the declaration of the components in the record clause.
3. Ada does not require that the entries in the representation clause be listed in address/bit order.

Our way of working through this, is to first compare the number of record components (including any discriminants) to the number of the entries in the representation clause. If the two counts are different, we flag the type as "not-bitmap-able". If, however, the two counts are identical, we take the next step, which is to sort the representation clause entries into address/bit order. Finally, we sort the internal representation table of record components into the same order as the (reordered) representation clause entries.

For discriminated record types, we must first determine the list of components that are selected by the (specific or default) discriminator values, then collect the intersection of the selected component names with their corresponding entries in the representation clause, and finally sort the surviving components names into (address/bit) representation clause order. As in the non-discriminated case, if there isn't a one-to-one mapping between the (surviving) component names and the (surviving) entries in the representation clause, then the type is flagged as "not-bitmap-able".

3.2 Arrays

An additional requirement is to handle arrays. Ada allows arrays to be components of a record type, as long as there are constraints upon the array component. (Granted, other pathological cases could be constructed, such as pointers to arrays, which we ignored for the first implementation.) For arrays, we must determine several properties:

1. the number of elements (derived from the constraints)
2. the type of the elements in the array.

If the component is an array of arrays, the tool continues to decompose the arrays until a base element type is discovered. Then, the dimensions of the array, as obtained during the array navigation, as well as the fundamental properties of the base element type, are reported. If the base element type is a record type, the tool creates a subsection and decomposes the record type. It should be understood that each time the tool encounters a record type, the subsection created has identical structure as when the record type is at the top-level.

Again, it's easy to construct pathological cases of non-homogeneous arrays that we haven't considered in this first implementation. Since the goal of the tool is to produce documentation for the customer's code base, our focus is to provide coverage of the constructs represented in that code, rather than develop a general solution.

3.3 Segmented Type Names

Types whose name consist of several segments (e.g. *somepackage.subpackage.typename*) offer a challenge. While it is possible to eventually navigate to the definition of a segmented type name, it is far easier to rely upon a navigation interface offered by the Ada Analyzer, **Ada_Program.Definition**, that cuts short the effort needed for navigation to the type declaration.

3.4 Record Handling

In all cases, we have a record declaration to start with. In Ada, a record declaration can contain both component and pragma entries. We ignore any possible pragma entries, as they are not represented in the customer code base. For all practical purposes, we consider a record declaration to be composed of a number of component declarations. Each component declaration follows the format (with the initial value, optionally):

```
name : type_mark := initial_value ;
```

We begin by obtaining the list of the record components from the record declaration. The list is then operated upon by the Process_Component procedure, once per component (after resolution of derived types via **Asis.Type_Definitions.Type_Structure**, and resolution of private types by **Asis.Type_Definitions.Completed_Ground_Type**.)

The essence of Process_Component is a case statement, with a branch for each of the basic type kinds:

- A_Subtype_Definition
- A_Record_Type_Definition
- A_Derived_With_Record_Type_Definition
- An_Enumeration_Type_Definition
- A_Float_Type_Definition
- A_Decimal_Fixed_Type_Definition
- A_Fixed_Type_Definition
- An_Integer_Type_Definition
- A_Modular_Type_Definition
- An_Array_Type_Definition

Each component of the record is examined. We extract the component name, its dimensionality, the name of its base type, its range (derived from constraints

upon the component subtype), size, accuracy, and any description, note or unit annotations.

Each type encountered during component decomposition may require special handling to extract this data. For example, the method used for gathering the size information from a (scalar) enumeration type differs from the method used for an array of enumeration types.

All this information about the component is then passed to the Add_Element_Table_Entry routine, which populates the in-memory intermediate representation.

3.5 Annotations

We mentioned previously the desire to take advantage of the many utilities provided by the Ada Analyzer. One such set of utilties is the annotation-gathering routines. It is quite easy to gather specifically formatted annotations "attached" to record components (in this case, represented by *An_Element*:

```
Annotation_Analysis.Collect (
       From_Decl_Or_Statement => An_Element,
       Config => Annotation_Configuration,
       Annots => Element_Annotations);
```

The Ada Analyzer code permits user configuration files to direct whether the annotations are expected to precede or follow the declaration. Rather than permit this potentially confusing configuration issue, we hard-coded the application to expect annotations immediately before the declarations.

The project requires additional information to be provided for the creation of the IDD that can not be extracted by the ASIS interface. For these cases, we require the coders to insert annotations. The tool looks for the following annotations:

```
-- @ DESCRIPTION: This is an example description. These
--   annotations can span multiple lines.
-- @ NOTES: This is an example of a note.
-- @ UNITS: kilometers
-- @ ACCURACY: 1 meter
```

Record declarations (e.g. the top level record) can have a special annotation, **MSG_DESCRIPTION**, whose text populates the paragraph following the section heading.

3.6 Producing Output

Once the navigation and collection activity completes, the next step is to construct the element and structure tables, starting with the top-level record. Having created the intermediate representation, this step was straightforward. Records that are encountered during the decomposition of the top-level record follow are

decomposed and represented in nested subsections, each with the appropriate header, description (from the **MSG_DESCRIPTION** annotation), element and structure tables.

An element table is a table that lists (in the order indicated by the representation clause) each component name, its base type (integer, modular, float, record, enum, array), its size, range, and any annotation-specified notes, description, and units.

A structure table is a pictorial representation of the memory layout of the record structure, with the bits along the horizontal axis, and word along the vertical axis.

The output format was chosen to be Rich Text Format, so that the tool-produced documentation would be editable by Microsoft Word. (We briefly explored HTML as an output mechanism, and found it to be expressive enough. However, the RTF was required by our customer.)

As an example of the input and output produced by the tool, consider the following record type *Message_Type* and its representation clause contained within the following package specification:

```
package Iddet_Test_Pkg is
  Word : constant := 4; -- storage unit is byte, 4 bytes per word
  type Mode_Type is (Fix, Dec, Exp);
  type Mode_Mask_Type is array (Mode_Type) of Boolean;
  type My_Float is new Float digits 5;
  subtype Sst is Integer range 1 .. 23;

  -- @MSG_DESCRIPTION: this is the Vrecord_Type annotation
  type Vrecord_Type (Mode : Mode_Type) is
    record
      -- @DESCRIPTION: This component is a non-variant
      --  Natural component.
      -- @UNITS: Kilograms
      -- @NOTES:  These are notes associated with Trange.
      Trange : Natural;
      case Mode is
        when Fix =>
          -- @DESCRIPTION: This component is a variant
          --  Boolean component.
          -- @UNITS: N/A
          -- @NOTES:  These are notes associated with Vboo.
          Vboo : Boolean;
        when Dec =>
          -- @DESCRIPTION: This component is a variant
          --  My_Float component.
          -- @UNITS: Kilograms
          -- @NOTES:  These are notes associated with Vflo.
          Vflo : My_Float;
```

```
      when Exp =>
        -- @DESCRIPTION: This component is a variant
        --  Sst component.
        -- @UNITS: Degrees Centigrade
        -- @NOTES:  These are notes associated with Sst.
        Vint : Sst;
    end case;
  end record;
for Vrecord_Type use
  record at mod 8;
    Mode at 0 * Word range 0 .. 7;
    Trange at 1 * Word range 0 .. 31;
    Vint at 2 * Word range 0 .. 31;
    Vflo at 2 * Word range 0 .. 31;
    Vboo at 2 * Word range 0 .. 1;
  end record;


subtype Mart is Integer range 1 .. 2;
type Fromt is array (Mart'First .. Mart'Last) of
    Vrecord_Type (Mode => Dec);

-- @MSG_DESCRIPTION: this is the Message_Type annotation
type Message_Type is
  record
    -- @DESCRIPTION: This component is an integer component.
    -- @UNITS: Meters
    -- @NOTES:  These are notes associated with X.
    X : Sst;
    -- @DESCRIPTION:This component is a Boolean component.
    -- @UNITS: N/A
    -- @NOTES:  These notes are associated with Y.
    Y : Boolean;
    -- @DESCRIPTION:This component is a nested
    --  variant component.
    -- @UNITS: N/A
    -- @NOTES:  These notes are associated with Z.
    Z : Fromt;
    -- @DESCRIPTION:This component is an array component.
    -- @UNITS: N/A
    -- @NOTES:  These notes are associated with Mode_Mask.
    Mode_Mask : Mode_Mask_Type;
  end record;
for Message_Type use
  record at mod 8;
```

```
      X at 0 * Word range 0 .. 31;
      Y at 1 * Word range 0 .. 0;
      Z at 2 * Word range 0 .. 143;
      -- bits at 1..8 are not used
      Mode_Mask at 7 * Word range 9 .. 11;
    end record;
end Iddet_Test_Pkg;
```

This example shows a top-level record **Message_Type** with both scalar (**X**, **Y**) and composite (**Z**, **Mode_Mask**) elements. Observe the **Z** component is an array of discriminated records.

The tool produces RTF-formatted tables as in Figures 1 - 4.

1.1.1 IDDET_TEST_PKG.MESSAGE_TYPE

this is the Message_Type annotation

0 0	0 1	0 2	0 3	0 4	0 5	0 6	0 7	0 8	0 9	1 0	1 1	1 2	1 3	1 4	1 5	1 6	1 7	1 8	1 9	2 0	2 1	2 2	2 3	2 4	2 5	2 6	2 7	2 8	2 9	3 0	3 1	
X																																Word 0
*	Spare																															Word 1
[0.. 31] + Z (2 elements of Vrecord_Type)																																Word 2
[32.. 63] + Z (2 elements of Vrecord_Type)																																Word 3
[64.. 95] + Z (2 elements of Vrecord_Type)																																Word 4
[96.. 127] + Z (2 elements of Vrecord_Type)																																Word 5
[128.. 143] + Z (2 elements of Vrecord_Type)																				Spare												Word 6
Spare										Mode_Mask (3 elements of Enum)										Spare												Word 7

Word	Bit	Name
1	0	Y

Fig. 1. This is the structure table for the top-level record type (Message_Type). The smaller sub-table following the structure table is an index into the structure table to describe elements whose typographic layout size prohibits in-table display

1.1.1.1 Iddet_Test_Pkg.Message_Type.Z (one element) (MODE => 1)

this is the Vrecord_Type annotation

0 0	0 1	0 2	0 3	0 4	0 5	0 6	0 7	0 8	0 9	1 0	1 1	1 2	1 3	1 4	1 5	1 6	1 7	1 8	1 9	2 0	2 1	2 2	2 3	2 4	2 5	2 6	2 7	2 8	2 9	3 0	3 1	
Mode								Spare																								Word 0
Trange																																Word 1
Vflo																																Word 2

Fig. 2. Structure table for the nested discriminated record. Note the heading indicates the discriminant value

In the element table (Figure 3), the range for the X component has been extracted via ASIS, as well as the enumeration values for the Y component. Since the type for X is a subtype, the ASIS code has extracted the base type of the component, and the ranges (e.g. constraints) appropriate to the subtype.

1.1.2 IDDET_TEST_PKG.MESSAGE_TYPE

Name	Description	Type	Range	Acc	Units	Notes
X	This component is an integer component.	Integer	1.. 23		Meters	These are notes associated with X.
Y	This component is a Boolean component.	Enum	False=0 True=1		N/A	These notes are associated with Y.
+ Z	This component is a nested variant component.	Vrecord_Type			N/A	These notes are associated with Z.
+ Mode_Mask	This component is an array component.	Enum			N/A	These notes are associated with Mode_Mask.

Fig. 3. This is the element table for the top-level record. The + decoration on the component name indicates a composite structure

1.1.2.1 Iddet_Test_Pkg.Message_Type.Z (one element) (MODE => 1)

Name	Description	Type	Range	Acc	Units	Notes
Mode		Enum	Fix=0 Dec=1 Exp=2			
Trange	This component is a non-variant Natural component.	Integer	0.. 2147483647		Kilograms	These are notes associated with Trange.
Vflo	This component is a variant My_Float component.	Float	5		Kilograms	These are notes associated with Vflo.

Fig. 4. This is the element table (one element) in the array of discriminated records

Note further that annotations associated with the records and their components are incorporated into the element table.

4 Transition

4.1 Looking Ahead

Hundreds of pages of interface description documents have been produced using this tool. When the source code changes now, instead of gathering an army of technical documentation specialists, the tool can be rerun across the new source code, and the most tedious and painstaking portion of the production of the interface documents automated.

The tool functionality is currently frozen. However, looking ahead, several enhancements could be envisioned. One, the output routines could be rewritten to output XML, with hyperlinks between the components. This would allow a web-enabled document with easy traversal between the various record relationships.

Another enhancement that can easily be envisioned, is to convert the present ASIS code (based upon the Rational ASIS 1.1 implementation) to ASIS 2.0, and take advantage of Ada95's features.