CHAPTER 5
EXTERNAL REPRESENTATION OF DIANA

This chapter describes how a DIANA tree may be represented in ASCII for communication between different computing systems. The presentation is informal; for a detailed discussion of the issues involved, see Chapter 4 of the IDL Reference Manual [9]. Although any conforming implementation of DIANA is required to be able to map to and/or from this *external* representation of DIANA, other *internal* representations are permitted. Indeed, we expect these latter (non-conforming) representations to be the preferred means of communication between tools on a single computing system. The standard external form is defined to assist debugging and to allow communication between computing systems, not as the typical communication between tools.

The design of this external representation was guided by three principles:

- There must be a relatively straightforward way of deducing the external representation from the DIANA specification of Chapter 2.
- The external representation must not unduly constrain the implementation options outlined in Chapter 6.
- It must be possible to map between the external representation and a variety of internal representations in a reasonably efficient manner.

We expect that each installation that wishes to communicate with others via an ASCII representation of DIANA will create a reader/writer utility to map between the external representation and whatever internal representations are in use at the installation.

The external representation is described in Figure 5-1 on page 146. It is the usual sort of recursive construction. Note that square brackets [...] surround the attributes of a node and angle brackets <...> surround items of a sequence.

We illustrate the external representation using the IDL example from Section 1.4.1, repeated here as Figure 5-2 on page 147. From this example, nodes **plus**, **leaf**, and **tree** might be represented externally as follows:

```
plus                -- a node with no attributes

leaf [ name "A"; src representation_of_source_position ] -- leaf for A

tree [ left leaf [name "A"]; right leaf [name "B"]; op plus] -- A + B
```

DEFINITION OF EXTERNAL DIANA

node represented by the name of its type, followed by `[', followed by the representation of its attributes (separated by semicolons), followed by `]'. If there are no attributes, the `[ ]' may be omitted.

attribute represented by the name of the attribute, followed by the representation of the value of the attribute.

comment start with double hyphen ('--'); terminate with the end of the line.

REPRESENTATION OF BASIC TYPES

`Boolean` represented by the tokens `TRUE` and `FALSE`.

`Integer` represented by a sequence of digits with an optional sign. The value is interpreted as being a decimal integer.

`Rational` represented as a decimal or based number (in the ADA sense and using the ADA syntax), or as the quotient of two unsigned integers, decimal numbers, or based numbers.

`String` represented as the sequence of ASCII characters representing the value of the string, surrounded by double quotes. Any quotes within the string must be doubled. The nonprinting ASCII characters are represented as in ADA.

Sequence represented by a sequence of representations for individual values of the sequence, separated by spaces, and surrounded by angle brackets (`<...>').

Private types are provided by the structure definition. For our purposes, the external representations of the private types used in DIANA are provided in a refinement of the DIANA abstract structure.

Spaces are not significant except to separate tokens.

Case distinctions between identifiers (such as node names) are significant, as in IDL.

Figure 5-1: External DIANA Form

Note that no representation is shown for the value of the attribute *src*, which is the private type Source_Position; this point is addressed further below. Note also that, because these examples show external DIANA which is expected to be ASCII text, the usual typographic conventions for node names and attributes are not followed in them.

```
Structure ExpressionTree Root EXP Is

      -- First we define a private type.

      Type Source_Position;


      -- Next we define the notion of an expression, EXP.

      EXP  ::=  leaf | tree ;


      -- Next we define the nodes and their attributes.

      tree  =>  op: OPERATOR, left: EXP, right: EXP ;
      tree  =>  src: Source_Position ;
      leaf  =>  name: String ;
      leaf  =>  src: Source_Position ;


      -- Finally we define the  notion of an OPERATOR as the
      -- union of a collection of nodes; the null => productions
      -- are needed to define the node types since
      -- node type names are never implicitly defined.

      OPERATOR  ::=  plus | minus | times | divide ;

      plus => ;   minus => ;   times => ; divide => ;

End
```

Figure 5-2: Example *ExpressionTree* of IDL Notation

The external representation also provides a means for sharing attribute values between nodes. This fact does not necessarily imply that the corresponding internal representation is shared; for some attributes, the sharing in the external representation can be viewed simply as a technique for compressing space.

However, any attribute value which is *inherently shared internally*[1] must be represented externally in shared form. All of the tree-valued attributes of DIANA fall in this category.

In order for an attribute value to be shared in the external representation, one occurrence of the value must be labeled and all other occurrences must refer to that label. Any attribute value may be labeled, including node-valued attributes. The labeled *occurrence* of the value is represented in a normal way, except that it is preceded by a label identifier and a colon (':'). Each label *reference* consists of the label identifier followed by a caret ('^'), rather than the usual representation of the attribute value. A label identifier is a sequence of letters, digits, and isolated underscores starting with a letter; case distinctions among the letters are significant. For example, the tree for A+A could be represented in any one of the following four ways (among others):

```
tree [ left leaf [ name "A"]; op plus; right leaf [ name "A" ] ]

tree [ left leaf [ name y: "A" ]; op plus; right leaf [ name y^ ] ]

tree [ left  x:leaf [ name "A"]; op plus; right x^ ]

tree [ left x^; op plus; right x:leaf [ name "A" ] ]
```

Additionally, a node-valued attribute can be written free standing without being nested within some other node. For example, a fifth representation for the preceding example is

```
tree [ left x^; op plus; right x^ ]
x: leaf [ name "A" ]
```

Note that in these examples we have consistently avoided giving a representation for the source position attributes. Recall that source position is a private type whose representation must be supplied as part of the structure definition or a refinement of the structure. One way to represent the source position is to use the representation defined in the example refinement in Section 1.4.3 on page 28, repeated here for convenience in Figure 5-3 on page 149. Using this external form, a source position might be represented using the node structure:

```
leaf [ name "A";
       src source_position
         [ file "<user>test.ada" ;  line 3 ;  char 15 ]
     ]
```

---

[1] The phrase 'inherently shared internally' is intentionally loose. We believe that the phrase captures the essence of the situations where it is convenient to use sharing in the external representation. For a complete discussion of this issue, see the IDL Reference Manual.

```
Structure AnotherTree Renames ExpressionTree Is

     -- first the internal representation of Source_Position

     For Source_Position Use Source_Package;

     -- next the external representation of Source_Position
     -- is given by a new node type, source_external_rep

     For Source_Position Use External source_external_rep;

     -- finally, we define the node type source_external_rep

     source_external_rep =>  file   : String,
                             line   : Integer,
                             char   : Integer;

End
```

Figure 5-3: Example *AnotherTree* of IDL

Alternatively, a specification could define the source position to be represented externally as a string:

```
leaf [ name "A"; src "<user>test.ada/15/3"]
```

Each of these particular external representations in some sense contains the same information in that either one could be mapped to the same internal representation by the reader utility. Each installation must establish conventions for communicating between the reader/writer utility and its user-supplied packages to allow such user-supplied types to be mapped to and from the external form. Of course, other representations for the source position attribute are possible, many containing quite different information. A more complete treatment of the external representation of private types may be found in the IDL Reference Manual.

The refinement of the DIANA structure defines the external representation for four private types, *symbol_rep*, *number_rep*, *operator*, and *value*. Types *symbol_rep*, and *number_rep* are represented as strings externally, and *operator* is represented by an enumeration type.

The type *symbol_rep* is a string that contains the source representation of identifiers. The type *symbol_rep* also represents character literals, which are distinguished from other identifiers by surrounding the character with single quote

marks, as in ADA. An implementation must decide how to treat upper and lower case characters: it can normalize the representations of identifiers to use the basic character set, all lower case letters changed to upper case, or it can preserve the case used in the source, so that source can be reconstructed accurately.

The type *number_rep* is a string that has the source representation of numeric literals. An implementation may choose to normalize *numeric_literals* by removing underscores.

The type *operator* is represented by an enumeration type. In the refined DIANA specification a minimum enumeration set is given; it may be expanded by an implementation to include any other built-in subprograms.

The type *value* is represented as an integer or rational type if a value has been computed, or with a distinguishing node for the cases where the value has not yet been computed. A representation for ADA strings and arrays is also provided: a sequence of values.

A complete external representation starts with an indication of the root node of the corresponding structure, followed by a sequence of zero or more representations of nodes. The root indication can be either a label referencing a node elsewhere in the external representation or the root node itself. Since the representation of subnodes can be contained within the representation of the parent node, it is possible for the entire external representation to be given by the root (a compilation node in DIANA). It is permissible, on the other hand, to represent the DIANA tree in a flat form, where node-valued attributes are always represented by labels referencing non-nested representations of the nodes.

Following are two examples, both in flat form. In each case a short ADA fragment is followed by the external form of the DIANA. Note that these examples, like the figures in Chapter 3, are incomplete in that some attributes are omitted for expository convenience.

```
-- From package STANDARD (sort of)
type BOOLEAN is (FALSE, TRUE);
type INTEGER is range MIN_INT .. MAX_INT;

----------

PD0:  type              [ as_id PD1^ ;
                          as_var_s PD2^ ;
                          as_type_spec PD3^ ]
PD1:  type_id           [ lx_symrep "BOOLEAN" ;
                          sm_type_spec PD3^ ]
PD2:  var_s             [ as_list < > ]
PD3:  enum_literal_s    [ as_list < PD4^ PD5^ >
                          sm_size void ]
PD4:  enum_id           [ lx_symrep "FALSE" ;
                          sm_obj_type PD3^ ;
                          sm_rep 0 ;
                          sm_pos 0 ]
PD5:  enum_id           [ lx_symrep "TRUE" ;
                          sm_obj_type PD3^ ;
                          sm_rep 1 ;
                          sm_pos 1 ]
PD6:  type              [ as_id PD8^ ;
                          as_var_s PD7^ ;
                          as_type_spec PD9^ ]
PD7:  var_s             [ as_list < > ]
PD8:  type_id           [ lx_symrep "INTEGER" ;
                          sm_type_spec PD9^ ]
PD9:  integer           [ as_range PD10^ ;
                          sm_type_struct PD9^ ;
                          sm_size void ]
PD10: range             [ as_exp1 PD11^ ;
                          as_exp2 PD12^ ;
                          sm_base_type PD9^ ]
PD11: used_object_id    [ lx_symrep "MIN_INT" ;
                          sm_defn xxx^ ;          -- def for MIN_INT
                          sm_value xxx^ ;         -- def for MIN_INT
                          sm_exp_type PD9^ ]
PD12: used_object_id    [ lx_symrep "MAX_INT" ;
                          sm_defn xxx^ ;          -- def for MAX_INT
                          sm_value xxx^ ;         -- def for MAX_INT
                          sm_exp_type PD9^ ]
```

```
package REPORT is
    function EQUAL ( X, Y : INTEGER ) return BOOLEAN;
end REPORT;
```

```
A01: comp_unit          [ as_pragma_s A02^ ;
                          as_context  A03^ ;
                          as_unit_body A04^ ]
A02: pragma_s           [ as_list < > ]
A03: context            [ as_list < > ]
A04: package_decl       [ as_id A05^ ;
                          as_package_def A06^ ]
A05: package_id         [ lx_symrep "REPORT" ;
                          sm_spec A06^ ;
                          sm_body void ;
                          sm_address void ]
A06: package_spec       [ as_decl_s1 A08^ ;
                          as_decl_s2 A07^ ]
A07: as_decl_s          [ as_list < > ]
A08: as_decl_s          [ as_list <  A09^ > ]
A09: subprogram_decl    [ as_designator A10^ ;
                          as_header A11^ ;
                          as_subprogram_def void ]
A10: function_id        [ lx_symrep "EQUAL" ;
                          sm_spec A11^ ;
                          sm_body void ;
                          sm_location void ]
A11: function           [ as_param_s A12^ ;
                          as_name A18^ ]
A12: param_s            [ as_list < A13^ > ]
A13: in                 [ as_id_s A14^ ;
                          as_name A17^ ;
                          as_exp_void void ]
A14: id_s               [ as_list < A15^ A16^ > ]
A15: in_id              [ lx_symrep "X" ;
                          sm_init_exp void ;
                          sm_obj_type PD9^ ]
A16: in_id              [ lx_symrep "Y" ;
                          sm_init_exp void ;
                          sm_obj_type PD9^ ]
A17: used_name_id       [ lx_symrep "INTEGER" ;
                          sm_defn PD8^ ;
A18: used_name_id       [ lx_symrep "BOOLEAN" ;
                          sm_defn PD1^ ]
```