A Specification for an MTV generator

Revision: 1.7

Jeffrey R. Lewis

September 27, 1993

1 Introduction

A Message Translation and Validation (MTV) module is a module common in command and control systems. It performs the functions of translation and validation between three main representations of messages: External Representation (EXR), Internal Representation (INR) and User Representation (USR). EXR messages are transmitted to and from systems outside of the command and control system. Such messages tend to be terse and/or encoded for efficient communication between systems. Internal representation is how messages are stored and manipulated by the command and control system of which the MTV component is a part. The user representation is a format intended for viewing and input by the users of the system. Translation must be performed between EXR and INR representations, and between INR and USR representations. Validation must be performed on EXR messages and USR messages; INR messages are presumed to be valid.

An MTV generator is a program that takes as input a specification of an MTV module and produces the code for that module. The following document gives a requirements specification for MTV modules and thus specifies what an MTV generator must support.

2 The MTV representations

2.1 The External Representation

An external message consists of "bits on a wire"—simply a string of bits. Thus is has no explicit structure; however, it does have implicit structure described by its message format. According to the format, an external message is subdivided into fields of bits or bytes. Fields may be fixed length or variable length. When variable length, they must be terminated with a delimiter. The contents of a field is an encoding for either symbolic data, numeric data, or text.

For example, the following external message consists of a reporting location (loc), a Julian date and time, and a text message. The reporting location's value is a code for a particular Air Force Base. The date and time is a binary encoding of January 1 at 9:30am. The text is a variable length byte-field, terminated with a period, and is ASCII for "Hi".

$$\overbrace{010}^{\text{loc}}\underbrace{\overbrace{000000001}^{\text{day}}}_{1}\underbrace{\overbrace{01001}^{\text{hour}}}_{9}\underbrace{\overbrace{011110}^{\text{minute}}}_{30}\overbrace{\underbrace{01001000}_{\text{“H”}}\underbrace{01101001}_{\text{“i”}}\underbrace{00101110}_{\text{“.”}}}^{\text{message text}}$$

The specification for an EXR is usually part of the requirements levied upon the system that is employing MTV. Typically, these specifications are informal, textual descriptions of the EXR. An example informal EXR description is presented in figure 1. This is a rather contrived example, pieced together from several actual message formats, but it serves to demonstrate most of the message features that an MTV system is required to handle. Omitted for brevity in this example are the details of the bit layout for the first two fields.

2.2 The Internal Representation

The internal representation is a datatype in the language that the MTV instance is written in. In this case, that will be Ada. However, what the MTV generator will accept for an INR will be a subset of Ada datatypes. In particular, the INR can consist of integers, strings, enumerations, records, variant records, arrays and linked lists. The INR for a given message format is defined by the designer of the system employing MTV.

An INR is assumed to represent a valid message. For INR messages created by MTV, this is assured by the translation and validation process. The application employing MTV is held responsible for the validity of any INR message that it creates.

An INR corresponding to the previous example is presented in figure 2. Notice that the elements of the EXR description correspond roughly to the Ada datatype. There are, however, significant differences. For example, the Detection Confidence and Probability of Detect fields, which are separate in the EXR description are intermingled in a single array in the INR.

2.3 The User Representation

The user representation is a user-readable character string. The intent of the USR is for presenting messages to the user in a readable format and as a representation for user input of messages. A USR should be easily understood by a user familiar with the message format,

Example Message Format				
Field Number	Field Name	Field Size	Range of Values	Amplifying Data
1	Reporting Location	2 bits	0–2	0 = Andrews AFB 1 = Peterson AFB 2 = Wright Patterson AFB
2	Date/Time Group	 9 bits 5 bits 6 bits	 1–366 0–23 0–59	Julian Date & Time Julian Day Hours Minutes
4	Message Text	100 chars (max)		Variable length string
	Text Delimiter	1	.	Period
3	Altitude or Track Confidence	2 chars	01–99 HH MM LL NN	In thousands of feet High confidence Medium confidence Low confidence No confidence
5	Contact Data Group	 4 chars 1 char 2 chars 3 chars	 0001–9999 *a* 00–89 000–179	This group is repeated zero or more times in the body of the message Track Number H = Hostile U = Unknown F = Friendly Latitude Longitude
	Group Delimiter	3 chars	END	
6	Detection Confidence	8 chars	*aaaaaaaa*	H = High M = Medium L = Low N = None
	Field Separator	1 char	/	Slash
7	Probability of Detect	8 chars	*nnnnnnnn*	0-9

Note: if no Contact Data is available, each element in the Detection Confidence field must indicate None

Figure 1: External message format example

```
type Reporting_Location_Type is (Andrews_AFB,
                                 Peterson_AFB,
                                 Wright_Patterson_AFB);

subtype Julian_Day_Type is Integer range 1 .. 366;
subtype Hour_Type is Integer range 0 .. 23;
subtype Minute_Type is Integer range 0 .. 59;
type Julian_Date_Time_Record_Type is record
  Julian_Day: Julian_Day_Type;
  Hour: Hour_Type;
  Minute: Minute_Type;
end record;

type Message_Text_Type is array (1 .. 100) of Bytes;

type Track_Info_Discr_Type is (Altitude_Discr, Confidence_Discr);
subtype Track_Altitude_Type is Integer range 1000 .. 99000;
type Confidence_Type is (High, Middle, Low, None);
type Track_Info_Type (Discr: Track_Info_Discr_Type) is record
  case Discr is
  when Altitude_Discr =>
    Altitude: Track_Altitude_Type;
  when Confidence_Discr =>
    Confidence: Confidence_Type;
  end case;
end record;
```

Figure 2: Internal message format example

```
type Contact_Data_Type;
type Contact_Data_Link_Type is access Contact_Data_Type;
subtype Track_Number_Type is Integer range 1 .. 9999;
type Force_Code_Type is (Hostile, Unknown, Friendly);
subtype Contact_Latitude_Type is Integer range 0 .. 89
subtype Contact_Longitude_Type is Integer range 0 .. 179
type Contact_Data_Type is record
  Track_Number: Track_Number_Type;
  Force_Code: Force_Code_Type;
  Latitude: Contact_Latitude_Type;
  Longitude: Contact_Longitude_Type;
  Next: Contact_Data_Link_Type;
end record;

subtype Barrier_Probability_Type is Integer range 0 .. 9
type Barrier_Segment_Type is record
  Detection_Confidence: Confidence_Type;
  Probability_of_Detect: Barrier_Probability_Type;
end record;
type Barrier_Array_Type is array (1 .. 8) of Barrier_Segment_Type;

type Example_Message_Type is record
  Reporting_Location: Reporting_Location_Type;
  Reporting_Time: Julian_Date_Time_Record_Type;
  Reporting_Message_Text: Message_Text_Type;
  Reporting_Track_Info: Track_Info_Type;
  Reporting_Contact_Data: Contact_Data_Link_Type;
  Reporting_Barrier_Data: Barrier_Array_Type;
end record;
```

Figure 3: Internal message format example (cont'd)

although it may still be too terse for a non-familiar user. For example, a USR message for the external message from section 2.1 might be:

```
Peterson AFB 001/09:30 - "Hi"
```

The design of a USR is left up to the designer of the system employing MTV. There appears to be no established formal or informal method for describing USRs. This is presumably because the design of a USR is straightforward, given the structure already established in the INR.

2.4 The Logical Representation

In order to specify the contents of messages independent of their format, a fourth representation is introduced: the *logical representation.* The logical representation is a mathematical representation that precisely delineates what messages can be described by MTV. It is also the representation in which message constraints are specified. The logical representation does not, however, describe the physical representation of a message.

Like the INR, the logical representation is a structured representation—unlike the EXR and USR which are just strings of bits and characters. In fact, a given INR will likely be very similar in structure to its logical representation. The logical representation, however, is not tied down to the details of a particular programming language, and enables the specification of a message format and constraints independent of the programming language being used to implement it.

A logical message is composed of primitive fields and composite fields. A primitive field in a logical message is one of:

string (*maxlen*) A fixed or variable length character string (optionally bounded in length by *maxlen*)

integer (*range*) An unsigned integer, optionally bounded by *range*

name An instance of a previously defined logical representation.

Composite fields define the structure of a message. They combine fields into a new field with a given structure. A composite field is one of:

{ $label_1$: $field_1$, …, $label_n$: $field_n$ } A labelled n-ary product of *field*s. This is like a record in Ada.

[$label_1$: $field_1$, …, $label_n$: $field_n$] A labelled n-ary sum of *field*s. This is like a variant record in Ada where the discriminant is implicit and is an enumeration of $label_1$ … $label_n$. The *field*s are optional—a sum may just be a union of symbolic constants, corresponding to an enumeration in Ada.

field[*n*] An array of *field*s of length *n*.

$field^*$ A sequence of *field*s (a list). Essentially an array of arbitrary length. This has no immediate counterpart in Ada, although a sequence is easily implemented using a linked list.

A logical representation for the example message is in figure 4.

```
message Julian_Date_Time = {
     Julian_Day: integer (1-366),
     Hour: integer (0-23),
     Minute: integer (0-59)
}

message Confidence = [ High, Medium, Low, None ]

message ContactData = {
     TrackNumber: integer (1-9999),
     ForceCode: [ Hostile, Unknown, Friendly ],
     Position: { Latitude: integer,
                 Longitude: integer }
}

message Barrier_Segment = {
     Detection_Confidence: Confidence,
     Probability_of_Detect: integer (0-99),
}

message Example = {
     Reporting_Location: [ Andrews_AFB,
                           Peterson_AFB,
                           Wright_Patterson_AFB ],
     Reporting_Time: Julian_Date_Time,
     Reporting_Message_Text: string (100)
     Reporting_Altitude_or_Track_Confidence: [ Altitude: integer,
                                               Track_Confidence: Confidence ],
     Reporting_Contact_Data: ContactData*
     Reporting_Barrier_Data: Barrier_Segment[8]
}
```

Figure 4: Logical Representation Example

3 Message Translation

For each of the three primary MTV representations (EXR, INR and USR), a translation must be defined to and from the logical representation. In describing these translations, however, it will become evident that they perform the dual role of serving as formal specifications for the representation being targeted (EXR, INR and USR). Indeed, the specification for an EXR (or USR) to Logical translation can easily be seen as a specification for a parser of a particular external (or user) message format.

To describe the possible range of translations required of an MTV system, a set of primitive translation functions and *combinators* that build on them is presented for each of the EXR $\leftrightarrow$ Logical and USR $\leftrightarrow$ Logical translations. A *combinator* is a function that takes other translation functions as arguments. The translation for a message is built up by specifying the translations of the primitive fields, then combining those translations using combinators.

Unlike the requirements for EXR and USR translations, the requirements for INR translations are very simple. Since the two representations have similar structure, all that is required is to be able to specify the following mappings: integer $\leftrightarrow$ Integer, string $\leftrightarrow$ String, product $\leftrightarrow$ record, sum $\leftrightarrow$ variant record, array $\leftrightarrow$ array and sequence $\leftrightarrow$ linked list.

In the following, a *pattern* refers to something with the expressive power of regular expressions. For convenience, a single quoted string indicates a bitstring, and a double quoted string indicates a character string.

3.1 EXR to Logical

The translation from EXR to Logical is a mapping from bitstrings to logical representations. Each of the following functions and combinators takes the indicated arguments and a bitstring to be translated. Since fields can be variable length, each function or combinator returns both the appropriate logical representation, and the number of bits consumed in translation.

The following are the translation functions for primitive fields:

ascii(*length*): bits → string × integer Convert a fixed length string of bits, *length* bytes long, into a character string using an ASCII interpretation. The returned length is length of the input string *in bits*.

vascii(*delim*): bits → string × integer Convert a variable length string of bits into a character string using an ASCII interpretation. The string is delimited with the string specified in the *delim* argument. The returned length is the length of the ASCII encoded string plus the length of the delimiter, *in bits*.

binary(*length*): bits → integer × integer Convert a string of bits of the specified length into an integer using the binary conversion appropriate for the architecture of implementation. The returned length is the same as the input length.

asc2int(`length`): bits → integer × integer Convert a string of bits, interpreted as an ASCII string of digits *length* bytes long, into an integer. The returned length is length of the input string *in bits*.

In addition, both `binary` and `asc2int` may be modified by a combination of the the simple arithmetic operators $+$, $-$, $\times$ and $\div$ to specify offsets and scaling.

The following are the translation combinators for composite fields:

$\{spec_1 \ldots spec_m\}$**: bits** $\rightarrow \{label_1 : field_1,\ \ldots,\ label_n : field_n\} \times$ **integer**

Convert a bitstring into a logical product structure. The *spec*s are either a translation specification for a field

$$label_i : translation$$

or a delimiter spec

$$delim$$

or simply a number of bits to skip

$$(skip)$$

There must be exactly one translation spec for each element of the logical structure, although they need not be in order. The length returned is the total length consumed by the entire record.

$[spec_1 \ldots spec_n]$**: bits** $\rightarrow [\ label_1 : field_1,\ \ldots,\ label_n : field_n\ ] \times$ **integer**

Convert a bitstring into a logical sum structure. Each of the *spec*s is of the form:

$$pattern \Rightarrow label_i : translation$$

where *pattern* is a discrimination pattern that determines whether the bitstring represents a value from the $label_i$ element of the sum. There must be exactly one *spec* for each element of the logical sum, although they need not be in order. Each discrimination pattern is applied to the bitstring in order until one matches, then the associated *translation* is applied. For the case of a sum element where the *field* is omitted, the *translation* is similarly omitted, and the number of bits consumed is the length of the matching pattern. The length returned is the length consumed by the applied translation.

translation [*length*]: **bits** $\rightarrow$ *field*[*length*] $\times$ **integer** Convert a bitstring into a logical array of length *length*. The field type *field* is the same as the field type of *translation*. The length returned is the total length consumed by all the translations.

seq(*translation*, *end-pattern*, *delim***)**: **bits** $\rightarrow$ *field** $\times$ **integer** Convert a bitstring into a logical sequence. The *end-pattern* is a pattern to match on the bitstring to determine when the end of the sequence has been reached. The *delim* is an inter-element delimiter. The specified translation is repeated and the results made into a logical sequence until *end-pattern* matches. The field type *field* is the same as the field type of *translation*. The length returned is the total length consumed by all the translations (including delimiters).

In addition, a record translation spec may be modified by a distribution operator. A distribution operator is used to convert a record whose fields are each an array of the same length into an array of records, each field of which is now a singleton. In short, it converts a record of arrays into an array of records. This is indicated by prefixing the record spec with **array_distr**. It is an error to specify a distribution to a record all of whose fields aren't an array of the same length. Similarly, an array of records may be factored into a record of arrays using **array_factor**.

A translation from EXR to Logical for the Example message is presented in figure 5.

3.2 Logical to EXR

The operation necessary for Logical to EXR are then a straightforward variation on the EXR to Logical operations. Note that it is not necessary here to return the additional length information that was necessary in the EXR $\rightarrow$ Logical case for keeping track of how much of the bitstring was consumed by a translation.

ascii(*length*): string $\rightarrow$ bits Convert a logical string of the specified length into an ASCII character bitstring.

vascii(*delim*): string $\rightarrow$ bits Convert a variable length logical string into a delimited ASCII character bitstring.

binary(*length*): integer $\rightarrow$ bits Convert a logical integer into a bitstring of the specified length using the conversion appropriate for the architecture of implementation.

int2asc(`length`): integer $\rightarrow$ bits Convert a logical integer into an ASCII character bitstring of the specified length.

$\{spec_1 \ldots spec_m\}$: $\{label_1 : field_1, \ldots, label_n : field_n\}$ $\rightarrow$ bits Convert a logical product structure into a bitstring. The *spec*s are either a translation specification for a field

$$label_i : translation$$

```
Loc = [ '00' ⇒ Andrews_AFB,
        '10' ⇒ Peterson_AFB,
        '11' ⇒ Wright_Patterson_AFB ]

Date = { Julian_Day: binary(9), Hour: binary(5), Minute: binary(6) }

Confidence = [ "HH" ⇒ High,
               "MM" ⇒ Medium,
               "LL" ⇒ Low,
               "NN" ⇒ None ]

ContactData = { asc2int(4),
                [ "H" ⇒ Hostile,
                  "U" ⇒ Unknown,
                  "F" ⇒ Friendly ],
                asc2int(2),
                asc2int(3) }

Barrier_Segments = { Detection_Confidence: Confidence[8],
                     "/",
                     Probability_of_Detect: asc2int(1)[8] }

Example = { Reporting_Location: Loc,
            Reporting_Time: Date,
            Reporting_Message_Text: vascii(".") }
            Reporting_Altitude_or_Track_Confidence:
                [ [0-9] ⇒ Altitude: asc2int(2) * 1000,
                  . ⇒ Track_Confidence: Confidence ]
            Reporting_Contact_Data: seq(ContactData, "END", ""),
            "END",
            Reporting_Barrier_Data: array_distr Barrier_Segments }
```

Figure 5: EXR → Logical translation example

or a delimiter spec

$$delim$$

or simply a number of bits to skip

$$(skip)$$

There must be exactly one translation spec for each element of the logical structure, although they need not be in order.

$[spec_1 \ldots spec_n]$**:** $[\ label_1 : field_1,\ \ldots,\ label_n : field_n\ ] \rightarrow$ **bits** Convert a logical sum structure into a bitstring. Each of the *spec*s is of the form:

$$label_i : (tag, translation)$$

The *tag* is a bitstring to identify this element of the sum, if necessary. There must be exactly one *spec* for each element of the logical sum, although they need not be in order. For the case of a sum element where the *field* is omitted, the *translation* is similarly omitted.

translation $[length]$**:** $field[length] \rightarrow$ **bits** Convert a logical array of the specified length into a bitstring. The field type *field* is the same as the field type of *translation*.

seq($translation$, $delim$**):** $field^* \rightarrow$ **bits** Convert a logical sequence into a bitstring. The *delim* is an inter-element delimiter. The field type *field* is the same as the field type of *translation*.

In addition, both `binary` and `int2asc` may be modified by a combination of the the simple arithmetic operators $+$, $-$, $\times$ and $\div$ to specify offsets and scaling. Further, distributions and factors, as described in the previous section, may also be applied to these translations.

3.3 USR to Logical

The translation for USR to Logical is very similar to the one for EXR to Logical. The most significant difference is that everything in the USR is characters, not bits.

ascii(*length*): chars $\rightarrow$ string $\times$ integer Convert a fixed length string of characters, *length* long, into a string using an ASCII interpretation. The returned length is length of the input string.

vascii(*delim*): chars $\rightarrow$ string $\times$ integer Convert a variable length string of characters into a string using an ASCII interpretation. The string is delimited with the string specified in the *delim* argument. The returned length is the length of the string plus the length of the delimiter.

asc2int(`length`): chars $\rightarrow$ integer $\times$ integer Convert an ASCII string of digits *length* bytes long, into an integer. The returned length is length of the input string.

$\{spec_1 \ldots spec_m\}$: chars $\rightarrow$ $\{label_1 : field_1, \ldots, label_n : field_n\}$ $\times$ integer Convert a character string into a logical product structure. The *spec*s are the same as for EXR to Logical, except that the skip is in chars, not bits. There must be exactly one translation spec for each element of the logical structure, although they need not be in order. The length returned is the total length consumed by the entire record.

$[spec_1 \ldots spec_n]$**:** **chars** $\rightarrow$ [$label_1 : field_1$, ..., $label_n : field_n$] $\times$ **integer** Convert a character string into a logical sum structure. Each of the *spec*s is of the form:

$$pattern \Rightarrow label_i : translation$$

where *pattern* is a discrimination pattern that determines whether the bitstring represents a value from the $label_i$ element of the sum. There must be exactly one *spec* for each element of the logical sum, although they need not be in order. Each discrimination pattern is applied to the bitstring in order until one matches, then the associated *translation* is applied. For the case of a sum element where the *field* is omitted, the *translation* is similarly omitted. The length returned is the length consumed by the applied translation.

translation [*length*]**:** **chars** $\rightarrow$ *field*[*length*] $\times$ **integer** Convert a character string into a logical array of length *length*. The field type *field* is the same as the field type of *translation*. The length returned is the total length consumed by all the translations.

seq(*translation*, *end-pattern*, *delim***):** **chars** $\rightarrow$ $field^*$ $\times$ **integer** Convert a character string into a logical sequence. The *end-pattern* is a pattern to match on the character string to determine when the end of the sequence has been reached. The *delim* is an inter-element delimiter. The specified translation is repeated and the results made into a logical sequence until *end-pattern* matches. The field type *field* is the same as the field type of *translation*. The length returned is the total length consumed by all the translations (including delimiters).

3.4 Logical to USR

The translation for Logical to USR is very similar to the one for Logical to EXR.

ascii(*length***): string → chars** Convert a logical string of the specified length into an ASCII character string.

vascii(*delim***): string → chars** Convert a variable length logical string into a delimited ASCII character string.

int2asc(`length`): integer → chars Convert a logical integer into an ASCII character string of the specified length.

$\{spec_1 \ldots spec_m\}$**:** $\{label_1 : field_1, \ldots, label_n : field_n\}$ **→ chars** The *spec*s are the same as for Logical to EXR except that the skip is in chars, not bits. There must be exactly one translation spec for each element of the logical structure, although they need not be in order.

$[spec_1 \ldots spec_n]$**:** $[\ label_1 : field_1, \ldots, label_n : field_n\ ]$ **→ chars** Convert a logical sum structure into a character string. Each of the *spec*s is of the form:

$$label_i : (tag, translation)$$

The *tag* is a character string to identify this element of the sum, if necessary. There must be exactly one *spec* for each element of the logical sum, although they need not be in order. For the case of a sum element where the *field* is omitted, the *translation* is similarly omitted.

translation [*length*]: *field*[*length*] → **chars** Convert a logical array of the specified length into a character string. The field type *field* is the same as the field type of *translation*.

seq(*translation*, *delim***):** *field** → **chars** Convert a logical sequence into a character string. The *delim* is an inter-element delimiter. The field type *field* is the same as the field type of *translation*.

4 Message Validity

Messages have associated with them a notion of validity. There are two aspects to this:

- intra-field validity. The data in a field is in the range of valid data for that field.
- inter-field validity. The data in interdependent fields conform to specified constraints.

Intra- and inter-field validity are specified by constraints in the logical representation. Intra-field validity is specified in the declarations of each logical field. The valid range of data for an intra-field constraint is either implicit (e.g. delimited by the possible elements of a sum), or explicit (e.g. the bounds on an integer field). Inter-field validity is specified as propositions on the logical specification that must be satisfied.

The details of an invalid message are given by its *validity indicator*, a value that gives diagnostic information about what's wrong with a message. Each message format has a unique validity indicator tailored to the specifics of that format.

Derived from analysis of the specification of possible translations, the various intra-field constraint indicators are as follows:

MessageTruncated This indicates that at some point in the translation, it was determined that the bitstring was too short for the specified translation.

MessageTooLong This indicates that at the end of the translation for a message, the total number of bits consumed was not equal to the length of the message.

UnterminatedString This indicates that the delimiter for a variable length string was not found.

StringTooLong This indicates that a variable length string exceeded its optional maximum length.

InvalidDigit This indicates that in an **asc2int** translation, the field contained an ASCII character that was not a digit 0-9.

OutOfRange This indicates that an integer was out of its specified range.

MissingDelimiter This indicates that a delimiter specified between two fields in a record, or between two elements of a sequence, did not match the contents of the message at that point.

DiscriminationFailure This indicates that no discrimination pattern for a union matched.

UnterminatedSequence This indicates that the end of a sequence was not found.

An inter-field constraint is given a name for reference in the validity indicator. It is a predicate, defined as follows:

- simple comparisons on integer or string terms: $=$, $\neq$, $<$, $>$, $\leq$ and $\geq$ (for strings, lexicographic ordering is used)

- complex constraints using connectives **and**, **or**, **not** and **implies**
- quantifiers $\forall$ and $\exists$ for expressing constraints over arrays and sequences

An integer or string term is:

- a logical field name
- a simple arithmetic expression on (integer) terms using $+$, $-$, $\times$ and $\div$
- a **length** function for sequences and variable length strings

For example the following expresses the constraint mentioned in the note at the bottom of the EXR description:

length(Reporting_Contact_Data) = 0 implies

$\forall x$ in Reporting_Barrier_Data, x.Detection_Confidence = None

5 Test generation

The MTV generator must be able to generate its own tests. In particular, for each message format, a set of test data is to be supplied by the designer of the format, who also supplies the appropriate validity indicator. The generated test will translate and validate all the test data, indicating any inconsistencies between the specified validity indicator and the actual validity indicator.

6 The MTV artifact

The artifact produced by the MTV generator will be a module in the target language with a datatype for the INR, a declaration for its validity indicator and the following function interface:

- a *value* function from EXR to INR
- an *image* function INR to EXR
- a *check* function that indicates the validity of a given EXR, returning a validity indicator that describes the nature of the problem
- a *value* function from USR to INR
- an *image* function INR to USR
- a *check* function that indicates the validity of a given USR

The *value* and *image* functions are derived from the specified translations to and from the logical representation. In particular:

$$value(x : \mathrm{EXR}) : \mathrm{INR} = LogicaltoINR(EXRtoLogical(x))$$

$$image(x : \mathrm{INR}) : \mathrm{EXR} = LogicaltoEXR(INRtoLogical(x))$$

$$value(x : \mathrm{USR}) : \mathrm{INR} = LogicaltoINR(USRtoLogical(x))$$

$$image(x : \mathrm{INR}) : \mathrm{USR} = LogicaltoUSR(INRtoLogical(x))$$

This module will be realized by an Ada package whose PDL is similar to the one given in figure 6.

```
with MTV_Common;
package MTV_instance is
  -- INTERNAL REPRESENTATION
    type INR is ...

  -- EXTERNAL REPRESENTATION
    subtype EXR is Bitstring

    -- Functions for converting between EXRs and INRs
    function Image(Value_In : in INR) return EXR;
    function Value(Image_In : in EXR) return INR;

    -- Function for checking the validity of an EXR
    type EXR_Validity_Indicator is ...;
    function Check(Image_In : in EXR) return EXR_Validity_Indicator;

  -- USER REPRESENTATION
    subtype USR is VString

    -- Functions for converting between USRs and INRs
    function Image(Value_In : in INR) return USR;
    function Value(Image_In : in USR) return INR;

    -- Function for checking the validity of a USR
    type USR_Validity_Indicator is ...;
    function Check(Image_In : in USR) return USR_Validity_Indicator;

  -- Constraint raised by Image and Value functions
    Constraint_Error : exception;
end MTV_Instance
```

Figure 6: Ada PDL for an MTV instance

A requirement of the *value* functions is that they raise an exception when the input message violates constraints. In Ada, the `Constraint_Error` exception will be the one raised. The user may then consult the validity indicator returned by the *check* function for details of which constraint was violated.

Finally, it is required that each of the *value*/*image* pairs form a retraction pair, i.e. that the following two conditions always hold:

$value(image(x)) = x$

$image(value(x)) = x$ whenever $check(x)$ indicates a valid message