Message Specification Language (MSL): A Domain Specific Design Language for Message Translation and Validation

Revision: 1.9

Lisa Walton and James Hook

August 25, 1994

Contents

1 Overview

MSL (Message Specification Language) is a simple, textual, specification language that enables a user to define the translation and validation functions used in message processing software components. This document builds on the domain analysis and structure identified by Lewis in his specification of the MTV domain [1], which in turn built upon work by the SEI on a "Model Solution" to the MTV problem [2].

The basic components of an MTV solution are translation functions mapping among the *external representation* (EXR), the *internal representation* (INR), and the *user representation* (USR), together with a check function that may be applied to external or user messages. To support the specification of the check functions, and to give a clear semantic delineation to the scope of the possible translations, Lewis introduced a new representation, the *logical representation* (LOG), which generalizes the INR.

These components can also be described by the following MSL semantic domains.

1. Semantic Entities (Logical Declarations)
   - Logical Types (structure of logical expressions)
   - Logical Values (meaning of logical expressions)

2. Semantic Actions (Translation Functions)
   - The relationship between the EXR and LOG representations (explicit)
   - The relationship between the LOG and USR representations (implicit)

3. Validity Checks (Logical Constraints)
   - The set of syntactic constraints on the EXR and USR (implicit)
   - The set of semantic constraints on the EXR and USR (explicit)

From these three components, the system generates an internal representation, together with the necessary translation and validation functions.

In the logical declarations section, the user gives type definitions that implicitly define a set of constructor functions and basic values that can be used in the construction of logical expressions throughout the specification. The data types defined in the logical section must include the logical representation of the message format, which is the basis for the generation of the Internal representation data type.

The translation function definition section explicitly describes the translation from the external format to the logical format (EXR to LOG). The atomic translation functions are either specified by specific external-pattern, logical-value pairs or from a small set of built-in functions. Translation functions are combined with structural combinators derived from the basic message types. These include product, sum, array and list. The definition of local functions is allowed to support reuse and structure the specification.

From these descriptions, three other translation functions are also derived

- The LOG to EXR function
- The LOG to USR function
- The USR to LOG function

The constraint declaration section defines functions from the logical representation to an enumeration type of failure conditions.

2 Logical Declarations

The language for logical descriptions is a simple, lexically-scoped, strongly-typed, first-order declarative language. The primitive types are integers, integer subranges, strings, and strings of bounded length. Type constructors are provided for enumeration types, product types, sum types, arrays of fixed dimension and (homogeneous) lists.

2.1 Logical Types

Types may be declared and bound to *type identifiers*. Type equality is structural and is only defined in a type environment in which all free type identifiers can be resolved. Types are declared with declarations of the form

$$\textbf{type}\ i = t$$

where i is a type identifier and t is a type. The special type declaration that denotes the type of the message as a whole takes the form

$$\textbf{message\_type}\ i = t$$

There should be exactly one of these message_type declarations in each message specification.

The effect of a type declaration is to extend the type environment with the named type and the expression environment and type assignments with all projection and injection functions implicitly defined in t. All types are of one of the following forms.

1. Atomic types which are illustrated by the following table:

integer	Unbounded integer type
integer(k)	Integers between 0 and k, inclusive
integer($l..h$)	Integers between l and h, inclusive.
string	A string of unbounded length
string(k)	A string with maximum length k.

   There is a notion of type containment induced by the subrange types. For example, the type **integer(1..5)** is a subset of **integer**.

2. Product types provide a record-like labeled product. The type

$$\{l_1 : t_1, \ldots, l_n : t_n\}$$

   defines a type and n projection functions, $l_1 \ldots l_n$, each mapping the product type to the associated component type. Note that for the purposes of determining type equality, the order of the typed projection functions inside a product does not matter. For example,

   { day : integer, month : integer, year : integer }

   is equivalent to

   { month : integer, day : integer, year : integer }

3. Sum types provide a variant-record structure. The type

$$[l_1 : t_1, \ldots, l_n : t_n]$$

defines a type and n injection functions, $l_1 \ldots l_n$, each mapping the associated component type into the sum type. As with the product type constructor, the order of the labeled injection functions within the sum type constructor is arbitrary.

[true : one, false : one]

When the component type is one, as above, it may be omitted from the declaration. The boolean example above could also be written as the enumeration type

$$[\mathit{true}, \mathit{false}]$$

4. Arrays are declared by a type followed by a dimension written inside brackets.

type IntArray = integer [8]

5. Lists are indicated by a type followed by a postfix $*$.

type IntList = integer *

2.2 Logical Values

The logical expression language is used to define logical constraints and to express simple transformations on actions. Logical expressions may appear as part of an action or constraint (see next sections), or may be bound to an identifier as follows.

1. **val** $f = e$ where e is a logical expression (see below).
2. **fun** $f\ (l_1 : t_1, \ldots, l_n : t_n) = e$ where e is a logical expression (see below).

Function declarations must give the name of the function symbol being defined and the types of all the formal parameters. Within declarations, values and functions may be declared locally with a let binding. For example,

$$\begin{array}{lll} \textbf{fun}\ f\ (l_1 : t_1, \ldots, l_n : t_n) = & \textbf{let} & \textbf{val}\ x = e_1 \\ & & \textbf{fun}\ f'\ (x_1, \ldots, x_n) = e_2 \\ & \textbf{in} & e_3 \end{array}$$

The result type of the declaration is inferred from the type of the expression defining the body. All logical expressions must take one of the following forms.

1. Atomic expressions include expression identifiers (often called simply identifiers or variables), the "unit" value `()`, and string and integer constants. The types of identifiers must be specified in the environment. The type of `()` is `one`, of a string constant of length n is `string(`n`)`, and of an integer constant k is `integer(`$k \cdots k$`)`. The sub-typing relationship allows these constants to then match all types of which they are members.

Special Integer and String Values Base 10 integer constants may be expressed in the standard fashion, and in other bases using the notation

base#number#

For example, the integer 5 may be represented either as **5** or in base 2 as **2#101#**.

There are several special control characters available in msl represented by

carriage return	"\r"
newline	"\n"
tab	"\t"

Note that there should be no embedded blanks in any string value, since blanks are used as field delimiters in the inferred USR2LOG translation function.

Operators on Logical Expressions Expressions may be modified by unary operators or combined using binary operators. For example, $100 * 16$. The following is a list of valid expression operators

+	Binary function on integers
-	Binary function on integers
*	Binary function on integers
div	Binary function on integers
mod	Binary function on integers
~	Unary function on integers (for negative integer values)
and	Binary function on boolean expressions
or	Binary function on boolean expressions
not	Unary function on boolean expressions
dim	Unary function from arrays to integers (returns array length)
..	Unary function from pairs of integers to integer lists (returns bounds on index references, for example, 0..10)

2. Logical values of product type may be constructed by a tuple of labeled values inside product braces.

$$\{l_1 = e_1, \cdots, l_n = e_n\}$$

Two logical products are type equivalent as long as they each form a permutation of the same sequence.

3. Sum values are produced by a case statement. The case statement takes an expression in a subtype and a list of "arms". Each arm contains a label, an identifier, and an expression. The label must correspond with the label of the injection tag in the case statement. The identifier is bound to the injected value when the expression is evaluated. All arms must have compatible types, and the list of arms must be exhaustive.

$$\textbf{case } e \quad \textbf{of} \quad l_1(i_1) \,.\, e_1 \cdots l_n(i_n) \,.\, e_n \quad \textbf{end}$$

4. Logical arrays are constructed by a list of values inside angle brackets. Array elements must be of compatible type.

$$\langle e_1, \ldots, e_n \rangle$$

5. Logical lists are constructed by a list of compatibly typed values inside brackets.

$$[|e_1, \ldots, e_n|]$$

The special value [| |] denotes the empty list.

6. Bounded quantification of a logical list or array expression has the form

exists x **in** e_1 . e_2	where	x is an identifier, and e_1 a logical list
forall x **in** e_1 . e_2		e_2 is any boolean expression with x occurring freely in it.

For example,

exists i **in** 0..dim(A)-1 . A[i] <A[i+1]

where i is an index ranging over the elements of the list $[|0, 1, \ldots, n-1|]$, n is the length of the array A, and $0 \leq i \leq n-1$

7. Application of a function to a list of arguments is achieved by writing an identifier followed by the parenthesized list of actual arguments. All functions must be total.

$$f\ (x_1, \ldots, x_n)$$

3 Translation Functions

The only explicitly defined translation function is from the EXR to LOG representation. This function must declare an invertible map between the external representation and the logical representation. The function declaration may be structured by declaring (reusable) component functions.

As a reader of bit strings, the EXR to LOG translation function must take the appropriate action when either a read success or failure occurs. A component function which successfully reads a sequence of bits translates them and passes on the remainder of the unread sequence. In the case of a read fail (in which the component function did not find a bit sequence of the appropriate format) one of two actions occur.

1. If the component function is trying to read something of sum type, it must backtrack to the beginning of its read and use the next component in the sum translation function to re-read the bit sequence.

2. If function is not a sum constructor, then the failure of any component function causes the entire function to fail. Note that if every component of a sum constructor fails then it fails as well.

Read failures will propagate upward through enclosing constructors until either a sum translation function is encountered (in which case it can backtrack and re-read) or the top level function is reached. If the top level EXR to LOG function fails, then the message is actually invalid, and an appropriate error message must be generated.

It is assumed that all LOG representations are valid, and hence there is no corresponding notion of failure in the inverse function (LOG to EXR).

3.1 External to Logical Translation Functions

The EXR to LOG translation functions (semantic actions) are defined by function declarations of the form

EXRaction $i : t = a$

where i is a function identifier, t is a type or previously defined type identifier, and a is an action. There should be exactly one special message action in each message specification of the form

EXRmessage_action $i : t = a$

All actions, are primitive or composite functions from bits to logical types, and fall into one of the following categories.

1. Atomic built-in primitive bit string reader/writer functions

Bin2Int i	Binary to integer translation function on binary strings of length i
Int i	Translates i bytes of a binary string into an integer
Asc2Int i	Translates i bytes into an ASCII string, then to an integer
Asc i	Translate i bytes of a binary string into an ASCII string
VAsc s	Translates a variable length binary string ending in s into an ASCII string

All of the above primitives which read integers may also be modified by special scaling or offset functions over logical expressions of the form

Scale f f^{-1}

where f denotes a function from integers to integers and f^{-1} is the inverse of f. For example,

fun Pennies_to_Dollars (n:integer(100..99999)) = n **div** 100
fun Dollars_to_Pennies (n:integer(1..999)) = n * 100

EXRaction Money : integer = Asc2Int 2 (Scale Dollars_to_Pennies Pennies_to_Dollars)

Note that the type bounds are required on the functions because fractions are not legal logical values.

Because the EXR may contain delimiters and list separators between meaningful strings of bits or contain meaningless bits within a meaningful byte of information, two special primitives are required.

Skip i	Reads i bits and returns nothing
Delim s	Translates the bit representation of the string s and returns nothing

These primitives may not stand alone, and must be composed with other actions using one of the three special action composition operators **@ * ::**. For example,

(Bin2Int 6) @ (Skip 2)

The action composition operators can only be used to combine actions when at least one of the actions involved returns a Unit type. In addition, * and :: may only be used to combine a list action with a unit action (see below).

If any of the primitives fail to find the correct bit format a read failure occurs. If the primitive is a component of a structural combinator, then that combinator must deal appropriately with the failure of one of its component functions (use one of the two failure conventions described in the previous section).

2. Composition with user defined atomic external-pattern logical-value pairs.

$$a \mid v \Rightarrow e$$

where a is an EXRaction, v is a value of type string, integer, or unit and e is any logical expression. These user defined primitives act as "guarded reads" which fail if a pattern is not matched. For example,

EXRaction High_Confidence : [High, Med, Low, None] = Asc 1 | "H" ⇒ High

This declares an action which reads a byte and translates it to a character. If the character is an "H", it returns the unit value High. If not, the action fails. As with the built-in primitives, any enclosing combinator must also deal with the failure of one of its components.

A composite action that is part of a composite sum action may optionally omit its result value if that value is of unit type. See the structural sum combinator below for further details.

3. Structural combinators (product, sum, array, and list) over action declarations.

As with the primitives, any structural combinator can have a read fail. If the combinator is itself a component of another combinator, then that combinator must follow one of the failure conventions described in the previous section.

(a) Product action constructors define a sequence of actions.

$$\{l_1 : a_1, \ldots, l_n : a_n\}$$

For example,

```
EXRaction   Julian_Date : {day:integer, hour:integer, minute:integer} =
          {   year = (Bin2Int 9)@(Skip 2), day = (Bin2Int 6),
              month = (Skip 2)@(Bin2Int 5)
          }   @(Delim "End")
```

If any of the components of the product action constructor has a read failure, the product also has a read failure.

(b) Sum action constructors define a selection, similar to a case statement, of the form

$$[l_1 : a_1, \ldots, l_n : a_n]$$

Of the sequence of actions, only the leftmost successful action is returned from the sum. In the special case that an element of a sum is an action that reads a unit value, the label of the sum element will be preserved as the logical field value.

The following example describes one of each, first an action declaration which reads in a bit string character and interprets it as a confidence level; next one that reads a bit string and returns either a Confidence level of type one or, if that value not found, an Altitude of type integer.

```
EXRaction          Confidence : [ High, Med, Low, None ] =
                   [  High : Asc 1 | "H",
                      Med : Asc 1 | "M",
                      Low : Asc 1 | "L",
                      None : Asc 1 | "N"
                   ];
EXRmessage_action  Conf_or_Alt : [ Conf : [ High, Med, Low, None ], Alt : integer ] =
                   [  TrackConf : Confidence, Altitude : Asc2Int 5 ];
```

As noted above, in the Confidence sum it would have been equally correct to write, for example, High : Asc 1 | "H" $\Rightarrow$ High.

A sum has a read failure only if all of its component actions fail. As with the product constructor, if a sum is itself a component function then the encompassing function must handle the failure using the appropriate failure convention.

(c) Array action constructors are of the form $\langle a \rangle$ For example,

$$\langle \text{ Asc2Int 1 } \rangle$$

The failure conventions for arrays are identical to those of products.

(d) List action constructors are one of two forms, where a, a_1, and a_2 are semantic actions and s is a logical string value. The first form describes a possibly empty list followed by delimiter s, and the second form is defines a non-empty list which contains separators. Note that a_1 and a_2 must be of the same type.

a * (Delim s) for example (Int 1) * (Delim "END")

a_1 @ (Delim s) :: a_2 for example (Int 1) @ (Delim ",") :: (Int 1)

A list constructor may fail either because a list element was not found, or because a separator (first kind) or a delimiter (second kind) was not found.

3.2 Logical to User Translation Functions

The LOG to USR translation is inferred from the user-specified EXR to LOG translation functions. Because it is assumed that all LOG representations are correct, there is no explicit notion of failure for these functions. However, the inverse function (USR to LOG) can fail. The results of a read fail for either a primitive function or structural action combinator are as for the EXR to LOG functions.

4 Logical Constraints

Validity checks are defined in two ways. First, there are implicit syntactic constraints in the external to logical and user to logical mappings. If a message can be converted into a logical representation with these mappings then it is syntactically valid.

The second checking phase verifies that the message is semantically valid. Logical validity constraints are specified explicitly as tests of logical assertions about semantic entities in the logical expression language (described in Section 2).

Logical constraints have the general form

raise *error* **if** $(m : mtype)\ e$

where *error* is a string, m is a variable, *mtype* is the type of a message, and e is a logical boolean expression (possibly containing previously defined identifiers). For example, the sum actions defined in the previous section could be constrained by the following

fun No_Confidence (conf:[High, Med, Low, None]) = (conf = None);

raise Contact_Error **if** (m : TC_or_Alt)
No_Confidence(TrackConf(m)) **or** (Altitude(m) $\leq$ 1000);

5 Miscellaneous Semantic Issues

1. String values may not have any embedded white spaces.

2. Impact of scaling functions on action types. All type declarations refer to the logical message representation. For example, in the specification the type of EXRmessage_action is integer(1..999) $\rightarrow$ integer(1..9999), despite the type of the action Asc2Int.

   fun fun Dollars_to_Pennies (n:integer(1..999)) = n * 100
   fun Pennies_to_Dollars (n:integer(100..9999)) = n **div** 100

   EXRaction Money : integer = Asc2Int 2 (Scale Dollars_to_Pennies Pennies_to_Dollars)

3. Ambiguous declarations. The type of an arm of a sum should never be a prefix of the type of another arm in the same sum. For example, in the following examples, the first arm will always succeed and the second will never be reached even if a read failure occurs after the sum succeeds

```
EXRaction BogusSum1 : sumtype = [ Arm1 : Int 3, Arm2 : (Int 2) @ (Delim ‘‘.’’)]

EXRaction BogusSum2: sumtype = [ Arm1 : Asc 3, Arm2 : (Asc 2) @ (Delim ‘‘.’’)]
```

A Example

```
(* Type Declarations *)

type date_type =
     { day : integer(1..7),  month : integer(1..12),  year : integer(1980,1990) };

type president_type = [Reagan,Bush,Clinton,Unknown];

message_type mtyp = { President : president_type,
                      ElectionDate : date_type,
                      Cabinet  : string [12],
                      Donations : integer *  };


(* Logical Function Declarations *)

fun AddC (n : integer(80..99)) = n + 1900;
fun RemC (n : integer(1980..1999)) = n - 1900;

(* Action Declarations *)

EXRaction EDat : date_type =
          { day : Bin2Int 6, month : Bin2Int 4, year : Bin2Int 6};

EXRaction Year : integer = Asc2Int 2 Scale AddC RemC;

EXRaction Pres : president_type  =
          [Reagan: Year | 1984,
           Bush : Year | 1988,
           Clinton : Year | 1992,
           Unknown: Skip 16  ];

EXRmessage_action M : mtyp =
                  { President : Pres,
                    ElectionDate : EDat,
                    Cabinet : < VAsc ``|'' >,
                    Donations : Int 4 * Delim "/" };

(* Constraint Declaration *)

raise NotElectionYr if (m:mtyp) (President(m)=Unknown);
```

B Concrete Syntax

In the grammar below, the following conventions will be used:

(_)$^+$ or _$^+$ indicates that _ may be repeated one or many times
(_)* or _* indicates that _ may be repeated zero or many times.

Anything enclosed by quotes is a literal, and anything in capital letters may be substituted by an appropriate member of the type specified. (For example, INTEGER may be replaced by any value of type integer) Binary numbers are represented as 2#number#.

```
Message_Spec   ::=  ( "type" ID "=" LType )+
                    "message_type" ID "=" LType
                    ( "fun" ID "(" (ID ":" LType)+ ")" "=" LExp )*
                    ( "val" ID "=" LExp )*
                    ( "EXRaction" ID ":" LType "=" EXRact )+
                    "EXRmessage_action" ID ":" LType "=" EXRact
                    ("raise" ID "if" "(" ID ":" LType ")" LExp )*

LType          ::=  ID
                  | "(" LType ")"
                  | "Integer"
                  | "Integer" "(" INTEGER ".." INTEGER ")"
                  | "Integer" "(" INTEGER ")"
                  | "String"
                  | "String" "(" INTEGER ")"
(* Product *)     | "{" Labeled_Type+ "}"
(* Sum *)         | "[ Labeled_Type+ ]
(* Array *)       | LType "[" INTEGER "]"
(* List *)        | LType"*"

Labeled_Type   ::=  ID
                  | ID ":" LType

LAtomVal       ::=  INTEGER
                  | BINARY
                  | STRING
                  | UNIT
```

```
LExp            ::=  ID
                   | LAtomVal
                   | "(" LExp ")"
(* Product *)      | "{" (ID ":" LExp)+ "}"
(* Array *)        | "<" LExp+ ">"
(* ArrayRf *)      | LExp "[" LExp "]"
(* List *)         | "[" LExp+ "]"
(* Case *)         | "case" ID "of" (LExp "." LExp)+ "end"
(* App *)          | LExp "(" LExp* ")"
(* BinOp *)        | LExp BinOp LExp
(* UnOp *)         | UnOp LExp
(* Exists *)       | "exists" ID "in" LExp "." LExp
(* ForAll *)       | "forall" ID "in" LExp "." LExp

BinOp           ::=  = | ⟨ | = ⟨ | ⟩ | =⟩
                   | + | - | * | div | mod | .. | && | or

UnOp            ::=  dim | not |

EXRact          ::=  ID
(* Atomic *)       | "( Bin2Int" INTEGER ")" Scale
(* Atomic *)       | "( Int" INTEGER ")" Scale
(* Atomic *)       | "( Asc2Int" INTEGER ")" Scale
(* Atomic *)       | "( Asc" STRING ")"
(* Atomic *)       | "( VAsc" STRING ")"
(* Product *)      | "{" (ID "=" EXRact)+ "}"
(* Sum *)          | "[" (ID ":" EXRact)+ "]"
(* Array *)        | "<" EXRact ">"
(* List1 *)        | EXRact "*" EXRact
(* List2 *)        | EXRact "::" EXRact
(* Unit *)         | "( Skip" INTEGER ")"
(* Unit *)         | "( Delim" STRING ")"
(* NPMorphism*)    | EXRact LAtomVal vert LExp
(* w/ Unit *)      | EXRact "@" EXRact

Scale           ::=  ( Scale" ID ID ")"
                   | ϵ
```

C Index

References

[1] Jeffrey R. Lewis. A specification for an MTV generator. Technical report, Oregon Graduate Institute, September 1993. 94-003.

[2] Charles Plinta, Kenneth Lee, and Michael Rissman. A model solution for C^3I message translation and validation. Technical report, Software Engineering Institute, Carnegie Mellon University, December 1989. CMU/SEI-89-TR-12 ESD-89-TR-20.