Formal specification in VDM-SL of the Secure EDIFACT reference implementation

Gavin Kelly

Information Systems Engineering
Centre for Mechanical and Optical Metrology
National Physical Laboratory
Queens Road
Teddington
Middlesex
United Kingdom
TW11 0LW

ABSTRACT
Strict Conformance Testing (SCT) is a methodology for testing products against security standards. SCT has been applied to Secure EDIFACT, and a test suite has been developed for products conforming to the Secure EDIFACT Guidelines. This report contains the reference implementation of a Secure EDIFACT translator, which was used to validate the SCT test suite.

ISSN 1361–407X

National Physical Laboratory
Teddington, Middlesex, United Kingdom, TW11 0LW

Approved on behalf of the Managing Director
by Dr D. Rayner, Head of Information Systems Engineering Branch

Contents

1 Introduction

This document describes the development of the reference implementation of a Secure EDIFACT translator (see [1] for the reasons why such an implementation is needed). Briefly, a Secure EDIFACT translator is either an originator or receiver of secured EDIFACT interchanges. An originator accepts a request from the user to secure a given interchange by means of a specific service (for example, message content integrity). It then processes the interchange, and sends the interchange onto a network, with the necessary security added on. A receiver takes an interchange from the network, checks that any security that had been added is correct, and passes the original (i.e. before security was added) interchange on to its rightful recipient.

Section 2 gives an overview of the specification of the Secure EDIFACT translator that requires no understanding of the formal language VDM-SL. Section 3 gives a summary of issues and problems relating to the standard, and the rest of the document presents the specification of the secure EDIFACT translator, which relies on some understanding of both VDM-SL (see [2] for a description of VDM-SL, [3] for the International Standard for VDM) and Secure EDIFACT (see [4] for EDIFACT; [5] and [6] for Secure EDIFACT) in order to be fully comprehensible. There is an appendix that deals with how the specification was interfaced to the test suite.

2 Overview of the Specification

The reference implementation of a Secure EDIFACT translator was envisaged as being primarily of use in providing formal validation of the Secure EDIFACT Test Suite, with additional benefits of identifying ambiguities and mistakes in the Secure EDIFACT Guidelines, and of providing a rigorous foundation upon which future real implementations could be based. With the availability of tools [7] for converting VDM-SL formal specifications into C++ code, it was decided to develop the reference implementation in the formal specification language VDM-SL, and generate from this an executable C++ implementation. This has led to the formal specification having the appearance, in many places, of a high level programming language. Also, some of the more elegant features of VDM-SL, such as sequence modification, had to be avoided, as the C++ generator was known not to be able to translate them.

The remainder of this section describes the various aspects involved in the development of the specification.

2.1 Modelling EDIFACT Interchanges

EDIFACT Interchanges are, conceptually, hierarchical data structures, being made up of functional groups of messages. These messages are sequences of data segments, which are themselves sequences of data elements. These data elements may be further broken down into component values - which are strings of characters.

These data structures are conceptual in the sense that a transmitted Interchange consists of a single stream of characters, with punctuation indicating the various data structures above. It was decided that the VDM-SL specification should model the conceptual structure of an EDIFACT Interchange, and this was a straightforward task that raised only one issue: it is possible to require values to be of fixed length, yet there are also recommendations in the syntax implementation guidelines for EDIFACT (Section 9.3; [8]) that values representing

numbers should not have leading zeros. There needs to be clarification on how to represent numbers correctly.

2.2 Modelling UNSM and Segment Structure

An EDIFACT Message consists of a series of data segments (the body) surrounded by two special segments: a Header Segment and a Trailer Segment. UNSMs (United Nations Standard Messages) form a special class of message in that the body of data segments conforms to a standard format. This format allows segments, and groups of segments to be repeated, nested, or omitted. A recursive VDM-SL type is all that is required to model the format as stated in the EDIFACT syntax implementation guidelines (9.5 [8]). The issues raised as a result of specifying the format are:

- There is no explicit indication of the limit on the number of levels in the hierarchy of segment groups, but there are allowed (according to [8]) only nine numbers in the tag, giving an implicit maximum depth. This may lead to implementors not knowing how to implement tags correctly; and could lead to mutually incompatible translators
- The diagram in the EDIFACT implementation guidelines (9.5.4 [8]), illustrating segment nesting, does not correspond to the text.
- The diagrams in section 9.1 of ISO 9735 [4], again illustrating how segments may be nested, do not conform to the UNSM format, and it is not obvious that they conform to any sensible interpretation of nesting. Both this and the previous item may lead to confusion in what constitutes a correctly nested message, and this in turn increases the possibility that implementors will make mistakes.

Each segment in a UNSM has a prescribed form, in that each of its elements is predefined as being either composite or simple; and each simple element (or component of a composite element) has restrictions on length, alphanumeric format and its mandatory/ conditional nature. Modelling this in VDM-SL brought up no issues.

2.3 Dealing with Incoming EDIFACT Interchanges

We shall give a brief overview of the steps the Translator takes to deal with an incoming string representing an EDIFACT interchange (that may or may not have secured messages within it). Then we shall describe the steps in more detail, along with any issues.

- The UNA string, if present, is examined, and the relevant separators are updated if necessary.
- The string representing the incoming Interchange is broken up into a linear sequence of segments (called a segment stream).
- The (linear) segment stream is formatted into the hierarchical model of an Interchange.
- The service segments (e.g. headers and trailers) are semantically checked.
- For each UNH,UNT pair delineating a message, any Security envelope is separated off from the body of the message.

- Each message (now without security segments) is checked to see if it matches the UNSM it purports to be.
- Each separated Security Envelope is checked
- The valid hierarchical interchange is returned, along with the list of separated security envelopes

2.3.1 UNA String

There are a set of special characters (separators), stored in a list. Sending a UNA string replaces the standard list of special characters with the characters sent in the UNA string. This would be trivial if it were not for the fact that a translator must cope with both comma and point as decimal separators. This necessitates representing a decimal separator as two characters. However, if a UNA string is sent, this overrides the above. There is an issue in how to signal to the translator that it is to revert to the standard 'comma and point' system, as the user can define the separator using only one character, i.e. only one of comma or point. Another issue is that VDM-SL does not support the question mark, which is the standard EDIFACT escape character. This means that a dummy character has to be placed in the specification, which can then be altered to a question mark in the generated C++.

2.3.2 Segmentation

This is a simple matter of scanning along the EDIFACT string, skipping over escape characters, until a segment separator is found, at which point the string is split to give a segment string. This is then split further into its constituent element strings by a similar scanning process, looking for data element separators. Finally, a data element string is split into its component values by looking for component separators. Proceeding in this manner, a sequence of segments, correctly split into data elements, is produced. No issues arose as a result of writing the specification for this.

2.3.3 Hierarchy

This is done by pairing off segments denoting the headers of Interchange, Functional groups, and Messages with the segments representing the relevant trailers, checking that these structures are correctly nested, and that there are no extraneous segments. There were no issues raised in the writing of the specification of this.

2.3.4 Service Segments

There are various constraints on elements in the service segments: that header-trailer pairs shall have the same reference numbers; and that the numbers of segments/ messages/ functional groups in the Interchange shall be the same as the counters in the relevant trailers. It is also checked that the syntax type and version number correspond to either the value stated in ISO 9735 [4] or the security implementation guidelines [5]. Again, no issues arose.

2.3.5 Separating Security

Security Segments may be in two places within a message: between the header and the body; between the body and the trailer. It is therefore sufficient to remove any segments with tags that identify them as security segments from immediately after each UNH message header; and likewise from immediately prior to each UNT message trailer. As each security segment is separated off, it is checked structurally for its conformance to the security guidelines [5]. No issues arose from specifying this.

2.3.6 Checking Messages

An external database is queried with the UNSM name. If the database doesn't contain the relevant UNSM, a warning is added to the log file. If it does contain the UNSM, however, it returns to the specification the template of the UNSM. The UNSM is then compared to its template by locating each segment, in turn, within the template, ensuring that no mandatory segments have been omitted, and that no segment (or group of segments) is repeated more than its allowed number of times.

Another external database is then queried with the (tag)name of each segment in the message. If the segment is contained in the database, its template is returned to the specification, and the segment contents are compared against this template. This involves checking that: mandatory elements and components of elements are present; there are not too many elements or components; that values have valid length and characters; that numeric elements are valid numbers. The only issue to arise was how to treat fixed length number values (see section 2.1). It was decided to allow fixed length numbers to have fewer digits than is strictly allowed - this meant that examples of EDIFACT messages given in the security guidelines [5] would not cause the translator to flag an error, which they would do if fixed length numbers were treated strictly.

2.4 Security Checking

Firstly, it is checked that the structure of the security envelope conforms to the branching diagram in the security guidelines [5], in the same manner that UNSMs are structurally checked (see previous section). The individual security segments, and their constituent elements, had been structurally checked when they where being separated off from the UNSM.

It is then checked that the security headers link correctly to the security trailers. It is considered an error if the two sets of link numbers are not equal, or if there are repeated link numbers within the headers. There is a warning flagged if the header and trailer links indicate that the envelopes are not nested within each other. This issue is not raised in the security guidelines [5], but it is thought that overlapping envelopes ought to be avoided as there might otherwise be ambiguity, for example, in the order messages should be signed.

The next stage in checking the security is to collect all the relevant security parameters. A decision was made as to which parameters would need to be checked using a separate security module. The most obvious candidates are the Validation Values ([5] 0560). Along with these, it was thought that time stamps ([5] S501) and message sequence numbers ([5] 0516) should also be checked using the security module. This is done by passing the values to an implicit VDM-SL boolean-valued function (see Annex A2) that shall interface with the security module

and return a value that indicates whether or not the values it was passed were valid.

Many of the data elements in the security segments are constrained to take values from a finite set. Each such element, as listed in the security guidelines [5], is checked that it has a value in the relevant set. The issue raised in the specification of this is to do with conditional data elements. Many of the conditional elements may be safely omitted, but some seem to require a default value if the element is omitted. An example of this is Scope of Security Application (0541). It is permissible to omit this, but no clear default value is given to indicate on what the security module is to calculate signatures, etc.

If none of the above steps produced an error, then the EDIFACT string is deemed to be correct. A copy of the Interchange, in the hierarchical form, is returned, along with the sequence of security envelopes from the secured messages within the interchange, to allow further external processing.

2.5 Securing Outgoing Interchanges

There were alternate approaches to securing outgoing messages. The first was to give the translator an EDIFACT interchange to secure, and a prompt telling the translator which service is required (Message Sequence Integrity, Non-Repudiation of Origin, etc.). The other approach was to give the translator the EDIFACT interchange, as before, but provide skeleton security envelopes, instead of the service prompt. It was decided that the former approach, although being analogous to the way a real translator would behave, was less generic than the latter. As there are many non-equivalent ways, within secure EDIFACT, of offering the same security service, it was felt inadvisable to limit the reference implementation to one particular protocol.

2.5.1 Preprocessing Outgoing Interchanges

Given a hierarchical EDIFACT interchange and a sequence of collections of segments representing a skeleton security envelope, the EDIFACT interchange is flattened into a linear series of data segments. In turn, each message within this linear series is isolated. If there was no corresponding skeleton security envelope for this message, then the next message is proceeded to. However, if there was a security envelope to be added, then this is done as follows.

2.5.2 Securing Messages

The envelope to be put around the message is structurally checked, to make sure that it conforms to the branching diagram of the security guidelines [5]. If it passes, then, for each occurrence of Security Reference Number ([5] 0516); Date and Time ([5] S501); Validation Result ([5] S508) in the skeleton envelope, a request is put out to the security module for data. This data is then incorporated into the skeleton envelope to make a full security envelope. Each segment of this is then checked with respect to the makeup of its elements.

As the final step in securing the message, the security header segments are inserted immediately after the message header, and the security trailer segments are inserted immediately prior to the message trailer. This means that the previous 'number of segments in message' counter will be invalid - it is recalculated, and inserted into the element. An issue is raised about whether adding a lot of security to a long message may take the segment count to an invalid

number (i.e. a number higher than that which can be stored in the UNT data element 0074). A security envelope may add up to 126 segments to a message.

2.5.3 Postprocessing Outgoing Interchanges

The linear sequence of segments constituting the secured interchange must now be converted to an EDIFACT string. Values are converted by adding escape characters before any characters that need them. Component elements are then concatenated, separated by component separators. Then strings representing segments are generated by stringing elements together, separated by element separators, and the tagname prepended. This list of segment strings is then concatenated, to give the EDIFACT Interchange string. This is the final output for outgoing messages.

2.6 Testing the Implementation

Since the VDM-SL specification had been written with the knowledge that it had to be executable, the C++ generation proceeded reasonably smoothly, except for problems with the VDM-SL tool [7] not being compatible with the latest version of the 'gcc' C++ compiler.

A simple interface to the generated code was written, to allow input of an incoming EDIFACT string. The examples of secured EDIFACT messages in the security guidelines [5] were sent to the translator, and a number of errors in these were spotted, as a result of the translator refusing to accept them. One example had inadvertently omitted a component separator ([5] 5.1.3 USH, the second occurrence of S500). Also, in the examples, there is a constant misnaming of the USC segment as CERT. The third error is an example of a 'too short' fixed length number: Security Result Link (0534) is fixed at two digits, but consistently given as a single digit in the examples.

This simple test also highlighted a few minor errors in the specification, mainly in the in-built templates representing the service segments.

The next test was to alter some of the constrained data elements. Changing elements so that they were invalid (e.g. altering Security Function Coded (0501) in the security header so that it was no longer in the set $\{1, 2, 3\}$) did produce the required errors. Various header-trailer pairs were mis-linked, and counters of segments, messages and functional groups were altered so that they did not equal the true number of segments, etc. All these produced the required errors.

The final test in the incoming direction was performed by setting up a virtual security module, which responded that the security details it was sent to verify were false. This fed back to produce an error message in the translator.

For tests in the outgoing direction, these relied mainly on the output from the 'incoming' tests. The security envelopes were stored, and sent back modified as skeleton envelopes. Invalid modifications consisted of removing mandatory segments from the skeleton; repeating segments to beyond their limit; altering security links so that they did not match; and having the security module send invalid (too long) data elements in response to requests for data. All these caused the translator to report error messages.

3 Summary of Issues

During the development of the Reference Implementation, a number of issues regarding the standards were highlighted. These were:

- ISO 9735 [4] places an implicit maximum depth to which segment groups can be nested, but this is not made explicit anywhere.
- There are errors in the EDIFACT guidelines' [8] diagram illustrating segment group nesting.
- Diagrams in ISO 9735 [4] representing messages do not correspond to UNSMs as defined in the EDIFACT guidelines [8].
- There is no specified way of reverting to the standard decimal separator notation, once the user has altered it via a UNA string.
- There is a conflict over fixed length numbers, and the prohibition of leading zeros.
- There is no prescription on how security envelopes may overlap, which might cause ambiguities.
- Defaults should be specified for some conditional data elements, as there can be ambiguity if they are omitted.
- Adding security to a message might mean it has an invalid number of segments in a message.
- There are errors in the examples of secured messages given in the security guidelines [5].

4 The Specification

The following sections contain the formal specification of a Secure EDIFACT translator, written in the formal language VDM-SL. It specifies all the data structures used in EDIFACT as detailed in the EDIFACT standard [4], by VDM-SL types. There are also 'template' types, and values of these types will correspond to the various UNSMs and data segment definitions as laid out in the UN directories of UNSMs and data segments.

There are functions specified that will check the conformance of variables to all the ISO 9735 [4] types, and functions that specify various useful constraints that apply to data elements. There are also functions that specify how to extract from an EDIFACT string its individual elements of data.

All the service and security segments from ISO 9735 [4] and the security implementation guidelines [5], respectively, are specified, as are the branching diagrams for security envelopes [5], and AUTACK messages [6].

There are high level specifications of how to deal with incoming secured interchanges, and how to secure outgoing interchanges. At a lower level, the processes of separating and adding security to messages are specified, as are the processes of checking the contents of both security and service segments.

5 Types

5.1 Types modelling ISO 9735

An interchange may or may not have an advice segment. If it does, the advice string must begin with the characters UNA, and be preceded by 6 parameter characters. The one designated as unused in ISO 9735 must be a space character:

types

```
1.0  interchange :: advice-string : [una-string]
 .1                 interchange-body : plain-interchange;

2.0  una-string = char'
     inv u ≜
 .2      len (u) = 9 ∧ u (1,    ,3) = "UNA" ∧ u (8) =
```

Regardless of any advice segment, the interchange must be made up of: a header with tag name UNB; a body which is either a sequence of functional groups, or a sequence of messages; and a trailer with tag name UNZ:

```
3.0  plain-interchange :: interchange-header : data-segment
 .1                       interchange-body : (message*) | (functional-group*)
 .2                       interchange-trailer : data-segment

 .3  inv i ≜
 .4      i.interchange-header.tag.tagname = "UNB" ∧
 .5      i.interchange-trailer.tag.tagname = "UNZ";
```

A functional group consists of: a header with tag name UNG; a body consisting of a sequence of messages; and a trailer with tag name UNE. All messages should be of the same type, which is checked by comparing all the types of message in the group to the type of the first message. The type of a message is given by the first component of the second data element of the UNH segment that begins each message:

```
4.0  functional-group :: functional-header : data-segment
 .1                      functional-body : message+
 .2                      functional-trailer : data-segment

 .3  inv f ≜
 .4      f.functional-header.tag.tagname = "UNG" ∧
 .5      f.functional-trailer.tag.tagname = "UNE" ∧
 .6      let first-unh = f.functional-body (1) (1) in
 .7      ∀ i ∈ inds (f.functional-body) ·
 .8          let each-unh = f.functional-body (i) (1) in
 .9          first-unh.elements (2) (1) = each-unh.elements (2) (1);
```

A message consists of: a header with tag name UNH; a body consisting of a sequence of data segments; and a trailer with tag name UNT:

```
5.0  message = data-segment+

 .1  inv m ≜
 .2      len (m) > 2 ∧
 .3      m (1).tag.tagname = "UNH" ∧
 .4      m (len (m)).tag.tagname = "UNT";
```

A data segment consists of a tag, which indicates the tag code and nesting level, and a sequence of data elements. These data elements are either simple, composite, or omitted:

```
6.0  data-segment :: tag : tagtype
 .1                  elements : data-element*
```

A data element is a sequence composed of Values, any number of which may be nil. A simple data element is a data element of unit length, that is not nil. A composite data element is a series of one or more values, any number of which may be null (see following note for the exception to this), corresponding to omitted composite data elements. An omitted element is a unitary length sequence with the only entry being nil. (Technical notes:- the case of all component elements being omitted must be represented as an omitted data element; also, it must be possible for a composite element to have a sole entry, as this is the case where all but the first component are omitted. Composite couldn't be defined as a restricted data element due to a bug in the VDM-SL tool.) A value is simply a non-null string of characters.

```
7.0  data-element = [value]+;


8.0  simple = data-element

     inv s ≜
 .2      len (s) = 1 ∧ s ≠ [nil ];
```

9.0 $composite = [value]^+$

inv $s \triangleq$
.2 len $(s) \geq 1 \wedge$ elems $(s) \neq \{\text{nil}\}$;

10.0 $omitted = data\text{-}element$

inv $o \triangleq$
.2 $o = [\text{nil}]$;

11.0 $value = \text{char}^+$;

A tag is composed of a name (of three letters, as stipulated in the General Introduction to UNSM Descriptions in UNTDID) and an optional indication of its nesting depth and repetition at each level of nesting.

12.0 $tagtype :: tagname : tagnametype$
$tagvalue : [\mathbb{N}_1{}^+]$;

13.0 $tagnametype = \text{char}$

inv $c \triangleq$
.2 len $(c) = 3$;

Template Types

The template for each data segment definition is given by a sequence of structural elements. These structural elements correspond to the segment format, and indicate the nature of each data item, i.e. whether it is composite or simple:

$segment\text{-}template = element\text{-}template^+$;

$element\text{-}template = composite\text{-}template \quad simple\text{-}template$;

Each composite data element must be specified as being either mandatory or conditional. And each component data element therein must be specified:

16.0 $composite\text{-}template :: MC : \text{M} \mid \text{C}$
.1 $COMP : simple\text{-}template^+$;

| N

It is necessary to have a look-up table that returns a segment's template, given its tag name.

18.0 $template\text{-}map = tagnametype \xrightarrow{m} segment\text{-}template;$

A message template for a UNSM is defined recursively. It is a sequence of message elements, which are either group or segments. A segment template consists of a name, a status flag, and a limit on the number of repetitions. A group template consists of a status flag, a limit on the number of repetitions of the group, a group number. In addition it has a name, which in this case is the name of the trigger segment (which is always mandated to have one repetition for each instance of the group, so the other segment details are unnecessary). Finally, the group template specifies the contents of the group, which is itself a message template.

As an example, the template corresponding to figure 2 (in section 10) is coded as *security_envelope* in the **values** section. (The numbers in the dashed boxes of figure 2 are position indicators, as dealt with in Miscellaneous Types, below. They are not relevant to the coding of the template from the diagram.)

19.0 $UNSM\text{-}template = (UNSM\text{-}segment \mid UNSM\text{-}group)^{+}$

20.0 $UNSM\text{-}group :: name : tagnametype$
.1 $MC : \mathrm{M} \mid \mathrm{C}$
.2 $repetition : \mathbb{N}_1$
.3 $group\text{-}number : \mathbb{N}_1$
.4 $body : UNSM\text{-}template;$

21.0 $UNSM\text{-}segment :: name : tagnametype$
.1 $MC : \mathrm{M} \mid \mathrm{C}$
.2 $repetition : \mathbb{N}_1;$

$message\text{-}map = (\mathsf{char}^{+}) \xrightarrow{m} UNSM\text{-}template;$

5.3 Miscellaneous Types

When converting EDI values into numbers, it is desirable to record what category of number the initial string represented. A VDM-SL number is then represented as its real (*i.e.* rational) expansion, along with the above category flag. The latter may also be set to flag an error, when the string did not represent a true number in any sense. Decimal points need to be represented in a slightly unusual way, as it is necessary to deal with the point and with the comma notation obliviously. For this purpose, we have set up the 'a' and 'b' alternatives.

23.0 $semigroup = \textsc{Rational} \mid \textsc{Integer} \mid \textsc{Natural};$

24.0 $number :: digits : \mathbb{R}$
.1 $conversion : semigroup \mid \textsc{Error};$

25.0 $decimal :: a : \mathsf{char}$
.1 $b : \mathsf{char};$

When matching a data segment against a message template, it is necessary to have a position indicator and repetition counter. The position is a sequence of numbers. The head of the

sequence indicates the position at the current level in the hierarchy, and the subsequent entries indicate the position at the previous levels. An increase in level is only triggered by a group's trigger, and so, for example, level 0 and level 1 segments are both marked by a single number. Repetitions are also counted by a sequence of numbers. Again, the header indicates the number of repeats at the current level, and subsequent numbers indicate repetition at higher levels.

As an example, we again refer to figure 2 (section 10). The numbers in the dashed boxes are the *position_type*'s of the segments. We note that the USA segment may appear in two different contexts: identifying the algorithm used on the message as a whole; identifying the algorithm used in the certificate. These two cases may be distinguished by their *position_type*'s. If we were to parse a message consisting of the following segments, then we would derive the following positions and repetitions:

Tag	USH	USA	USH	USA	USC	USA	USR	UST	USR	UST	USR
Position	1	1:1	1	1:1	2:1	1:2:1	2:2:1	2	1:2	2	1:2
Repetition	1	1:1	2	1:2	1:2	1:1:2	1:1:2	1	1:1	2	1:2

We observe that there are three instances of the USA segment. The first two are related to the message contents, and the third is related to the certificate. The first two may be differentiated between solely on the basis of their repetition counter - the first is instantiated by the first occurrence of the USH, and the second is instantiated by the second occurrence of the USH. The third USA (the one detailing the certificate), is also instantiated by the second USH occurrence, but is differentiated by its position counter having 3 entries - indicating that it is further down the hierarchy. As a technical note, we have allowed values in the *repetition_type* sequence to take the value zero, as later functions will need an initial number of repetitions before they have matched the first occurrence.

$$\textit{position-type} = \mathbb{N}_1{}^{+};$$

27.0 $$\textit{repetition-type} = \mathbb{N}^{+};$$

In constraints, it is necessary to refer to instances of a segment. The segment is identified by a sequence of positive integers, and the above *position_type* suffices. The *instance_type* is the analogue of the *repetition_type*, being a sequence. Each integer value in the sequence gives the relevant repetition at that level of the hierarchy. A null value indicates that any repetition, at that level, is allowed. So the instance [2,nil] of segment [1,1] would match with the second occurrences of USA in the above example. It would match neither the first occurrence, due to instance mismatch ($[1,1] \notin [2,\mathsf{nil}]$), nor the third (on purely positional considerations $[1,1] \neq [1,2,1]$).

A segment reference is the segment's position and instance. Simple and component references are given by supplying an extra one or two numbers, respectively.

28.0 $$\textit{instance-type} = [\mathbb{N}_1]^{*};$$

$$\textit{segment-reference} :: \textit{instance} : \textit{instance-type}$$
$$\textit{position} : \textit{position-type};$$

30.0 *simple-reference* :: *instance* : *instance-type*
.1 *position* : *position-type*
.2 *simple-ref* : $\mathbb{N}_1$;

31.0 *component-reference* :: *instance* : *instance-type*
.1 *position* : *position-type*
.2 *composite-ref* : $\mathbb{N}_1$
.3 *component-ref* : $\mathbb{N}_1$;

32.0 *element-reference* = *simple-reference* *component-reference*;

During the checking of a message against its template, each segment in the message will be given a position and repetition indicator. These are both stored in a list.

33.0 *parsed-posn* = *position-type*$^+$

34.0 *parsed-repn* = *repetition-type*$^+$

6 Functions

6.1 Structural Checks

6.1.1 Checking Segments

There follows a function that checks that a candidate data segment has the structure that it purports to have, as indicated in the tag code. The candidate must not have more elements than the template indicates, and if it has fewer, then the ones that have been omitted by truncation must be identified, in the template, as being conditional.
Each data element in the candidate must either be:

- simple - in which case the template must indicate a simple data element at this slot, and the data element must be of the correct length and content;
- composite - in which case the template must indicate a composite data element at this slot, and the data element, and all its component parts must be correct; or
- omitted - in which case the template should indicate that the presence of this data element is not mandatory.

functions

```
35.0  correct-segment : segment-template × data-element* × decimal → 𝔹
      correct-segment (template, candidate, point) ≜
 .2     (len (candidate) ≤ len (template)) ∧
 .3     (∀ elt-indx ∈ {len (candidate) + 1, ..., len (template)} ·
 .4           template (elt-indx).MC ≠ M) ∧
 .5     (∀ elt-indx ∈ inds (candidate) ·
 .6           ((candidate (elt-indx) ≠ [nil ] ∧
 .7             is-simple-template (template (elt-indx)) ∧
 .8             len (candidate (elt-indx)) = 1 ∧
 .9             correct-simple (template (elt-indx), candidate (elt-indx) (1), point)) ∨
 .10           (candidate (elt-indx) ≠ [nil ] ∧
 .11            is-composite-template (template (elt-indx)) ∧
 .12            len (candidate (elt-indx)) ≥ 1 ∧
                correct-composite (template (elt-indx).COMP, candidate (elt-indx), point)) ∨
 .14           (candidate (elt-indx) = [nil ] ∧
 .15            template (elt-indx).MC ≠ M)));
```

A simple numeric data element is deemed to be correct if it converts to a number, and the length of that number is no longer than its template allows. (Technical Note: there is a confusion as to whether fixed length numbers should be allowed leading zeros. To deal with this, the specification allows fixed length numbers to be shorter than their specified length.) Other simple data elements must simply contain no characters outside the relevant character set, and be of the correct fixed length, or no longer than the variable length, as the case may be.

```
36.0  correct-simple : simple-template × value × decimal → 𝔹
 .1   correct-simple (stemplate, scandidate, point) ≙
 .2     if stemplate.AN = N
 .3     then let result = string-to-number (scandidate, point),
 .4              leng = number-length (scandidate, point) in
 .5          (if result.conversion = ERROR
 .6           then false
 .7           else (leng ≤ stemplate.length ∧ stemplate.FV = F) ∨
 .8                (leng ≤ stemplate.length ∧ stemplate.FV = V))
 .9     else ((len (scandidate) = stemplate.length ∧ stemplate.FV = F) ∨
.10          (len (scandidate) ≤ stemplate.length ∧ stemplate.FV = V)) ∧
.11          ((stemplate.AN = A ∧ elems (scandidate) \ elems (alpha) = {}) ∨
.12           (stemplate.AN = AN ∧
.13            elems (scandidate) \ (elems (alpha) ∪ elems (numer)) = {}));
```

A composite data element is deemed to be correct if it contains no more components than its template allows. Further, if it contains less, then none of the components omitted by truncation may be identified as being mandatory by the template. Further, each component must tally with the template, in that those indicated by the template as being mandatory must be present, and that any that are present must conform in length and content to the template. This is checked by treating the component data elements as simple elements, and checking that as such they would be valid.

```
37.0  correct-composite : simple-template+ × composite × decimal → 𝔹
 .1   correct-composite (ctemplate, ccandidate, point) ≙
 .2     (len (ccandidate) ≤ len (ctemplate)) ∧
 .3     (∀ com-indx ∈ {len (ccandidate) + 1, ..., len (ctemplate)} ·
 .4          ctemplate (com-indx).MC ≠ M) ∧
 .5     (∀ com-indx ∈ inds (ccandidate) ·
 .6          ((ccandidate (com-indx) ≠ nil ∧
 .7            correct-simple (ctemplate (com-indx), ccandidate (com-indx), point)) ∨
 .8           (ccandidate (com-indx) = nil ∧
 .9            ctemplate (com-indx).MC ≠ M)));
```

6.1.2 Checking Messages

The next few functions allow a message in flat file format to be checked against a template.

A message is deemed to be correct if the leading segment is found to have a match in the template (by calling the segment locator hunt_for), and the tail is in itself a correct message, taking into account any previous repetitions and changes in position. The checking function also records the position and repetition of each segment, for use in the constraints checking. If the segment is the last, then it is checked that there are no remaining mandatory segments in the template. Then, any error message is returned, along with the list of positions and repetitions of located segments.

The function will initially be called with the following parameters: the required template; the whole sequence of segments that constitute the message; [] - the parser will add repetitions

of segments as it finds them; [] - the parser will likewise add positions; [1] - being the first valid place in the template from which to look for the first segment; [0] - there have been no previous instances of the segment at template position [1].

```
38.0  correct-message : UNSM-template × data-segment+ × parsed-posn × parsed-repn ×
position-type × repetition-type → char* × parsed-posn × parsed-repn
 .1   correct-message (mtemplate, mcandidate, posn, repn, start-from, how-many) ≙
 .2     let mk-(next-posn, next-repn, here, new-many, pass) =
 .3             hunt-for ((hd (mcandidate)).tag.tagname, mtemplate,
 .4                       start-from, how-many),
 .5         new-posn = posn ↷ [here],
 .6         new-repn = repn ↷ [new-many] in
 .7     if pass ≠ []
 .8     then mk-(pass, new-posn, new-repn)
 .9     else if len (mcandidate) = 1
 .10        then mk-(remaining-mand (mtemplate, next-posn, next-repn), new-posn,
 .11                 new-repn)
 .12        else correct-message (mtemplate, tl (mcandidate), new-posn,
 .13                              new-repn, next-posn, next-repn);
```

If the segment locator hunt_for finds a match at the current position, then it first checks that this doesn't take the number of repetitions of this segment over the allowed number. If it is valid, and the segment is a trigger for a group, the parser opens the triggered group, by increasing the length of the position marker, and increments the relevant repetition counter. If the match is against a non-trigger segment, then it simply updates the repetition counter.

If no match is found at the current location, then an error is returned if the segment is mandatory and there have been no previous instances of the segment. If there is no missing mandatory segment, then the position marker is moved along the current level in the hierarchy, or if it cannot do this, up to the previous level of the hierarchy. A failure to be able to do either of these means that the end of the template has been reached without finding a match for the segment, and so an error is returned.

The locator routine should initially be called with the following parameters: the tag name of the segment that is to be located; the whole template in which the segment is to be located; the position from which the hunt is to start; the number of repetitions that have been previously recorded at that position.

The locator routine will return, on a valid match: the next starting position; the number of repetitions so far recorded at that position; the position pointer to be added to the list; the repetition counter to be added to the list. If the match was with a non-trigger, the first pair will be identical to the second pair. If the match was with a trigger, then the new repetition will be prepended with a 0, and the new position with a 1.

```
39.0  hunt-for : tagnametype × UNSM-template × position-type × repetition-type → position-type ×
repetition-type × position-type × repetition-type × char*
 .1   hunt-for (candidate, template, posn, repn) ≙
 .2     let current = pin-point (template, posn) (1) in
 .3     if candidate = current.name
 .4     then (if hd (repn) = current.repetition
 .5           then mk-(posn, repn, posn, repn, "Too many " ↷ candidate)
 .6           else (if is-UNSM-group (current)
 .7                 then mk-([1] ↷ posn, [0] ↷ [(1 + hd (repn))] ↷ tl (repn), posn,
 .8                          [(1 + hd (repn))] ↷ tl (repn), [])
 .9                 else mk-(posn, [(1 + hd (repn))] ↷ tl (repn), posn,
.10                          [(1 + hd (repn))] ↷ tl (repn), [])))
.11    else (if current.MC = M ∧ hd (repn) = 0
.12          then mk-(posn, repn, posn, repn, "Missing mandatory " ↷ current.name)
.13          else (if len (posn) > 1
.14                then (if hd (posn) < len (pin-point (template, tl (posn)) (1).body)
.15                      then hunt-for (candidate, template, [(hd (posn) + 1)] ↷ tl (posn),
.16                                     [0] ↷ tl (repn))
.17                      else hunt-for (candidate, template, tl (posn), tl (repn)))
.18                else if hd (posn) < len (template)
.19                     then hunt-for (candidate, template, [hd (posn) + 1], [0])
.20                     else mk-(posn, repn, posn, repn,
.21                              "Can't find a place for " ↷ candidate)));
```

To search a template for remaining mandatory segments once all the candidate segments have been located, we reproduce the part of hunt_for that moves through the template. If it manages to reach the end of the template, moving left to right and upwards, without hitting a mandatory segment, then it returns an empty string, else it returns an error message.

```
40.0  remaining-mand : UNSM-template × position-type × repetition-type → char
 .1   remaining-mand (template, posn, repn) ≙
 .2     let current = pin-point (template, posn) (1) in
 .3     if current.MC = M ∧ hd (repn) = 0
 .4     then "Missing mandatory " ↷ current.name
 .5     else (if len (posn) > 1
 .6           then (if hd (posn) < len (pin-point (template, tl (posn)) (1).body)
 .7                 then remaining-mand (template, [(hd (posn) + 1)] ↷ tl (posn),
 .8                                      [0] ↷ tl (repn))
 .9                 else remaining-mand (template, tl (posn), tl (repn)))
.10           else if hd (posn) < len (template)
.11                then remaining-mand (template, [hd (posn) + 1], [0])
.12                else []);
```

To find the contents of a template, given a position, it is necessary to repeatedly prune the template until the position indicator is of unit length, and then return the relevant branch of the template. It is necessary to return the branch in a sequence, as the very ends of the branch (*e.g.* the USA segment at [1,2,1] in figure 2 section 10) are UNSM_segments rather

than sequences.

```
    pin-point : UNSM-template × position-type → UNSM-template
.1  pin-point (template, posn) ≙
.2    if len (posn) = 1
.3    then [template (posn (1))]
.4    else pin-point (template (posn (len (posn))).body, posn (1, ..., len (posn) − 1));
```

To deal with UNSM structures that are not defined internally, there are two functions, the last of which is implicitly defined. When the C++ code is generated, it will be possible to hand write this function, which will access a database, returning a message structure, given a UNSM name. The first function will call this second function, passing to it the name of the message. If it receives a template back, it checks that the original message conformed to the template. If it received no template, it returns a *Not Found* flag.

```
    external-UNSM : message → NOT_FOUND | PASS | FAIL
.1  external-UNSM (candidate) ≙
.2    let template = DB-UNSM (candidate (1).elements (2) (1)) in
.3    if template = nil
.4    then NOT_FOUND
.5    else let mk-(pass, -, -) = correct-message (template, candidate, [], [], [1], [0]) in
.6         if pass = []
           then PASS
           else FAIL;


43.0 DB-UNSM (message-name : char*) unsm : [UNSM-template]
.1   post true ;


44.0 external-segment : data-segment × decimal → NOT_FOUND | PASS | FAIL
.1   external-segment (candidate, point) ≙
.2     let template = DB-segment (candidate.tag.tagname) in
.3     if template = nil
       then NOT_FOUND
.5     else let pass = correct-segment (template, candidate.elements, point) in
.6          if pass
.7          then PASS
.8          else FAIL;


45.0 DB-segment (segment-name : char*) seg : [segment-template]
     post true ;
```

6.2 Constraint Functions

6.2.1 Number Handling

There follow several functions to deal with constraints. The first of these converts Values to numbers. For a start, nil strings, empty strings, and strings with no decimal digits before a decimal point are all invalid ('.2' should be sent as '0.2' in EDIFACT). A positive rational is passed to the second function which generates an natural number, and a power of ten by which to divide this natural. If the string ended in with a decimal point, then the numerator is returned as '1', and the converter signals an error ('2.' should be written as either '2.0' or '2'). Otherwise, the number will be divided and the sign changed as required.

```
46.0  string-to-number : value × decimal → number
 .1   string-to-number (string, point) ≙
 .2     if len (string) = 0 ∨ hd (string) = point.a ∨ hd (string) = point.b ∨
 .3       (len (string) > 1 ∧ (string (1, ..., 2) = "-" ↷ [point.a] ∨
 .4         string (1, ..., 2) = "-" ↷ [point.b]))
 .5     then mk-number (0, ERROR)
 .6     else (if hd (string) = '-'
 .7           then (if len (string) = 1
 .8                 then mk-number (0, ERROR)
 .9                 else let mk- (numerator, denominator, pass) =
 .10                             positive (tl (string), point, 0, 0) in
 .11                   (if denominator = 1 ∨ ¬ pass
 .12                    then mk-number (0, ERROR)
 .13                    else (if denominator = 0
 .14                          then mk-number (− numerator, INTEGER)
 .15                          else mk-number (− numerator/denominator, RATIONAL))))
 .16          else let mk- (numerator, denominator, pass) = positive (string, point, 0, 0) in
 .17               (if denominator = 1 ∨ ¬ pass
 .18                then mk-number (0, ERROR)
 .19                else (if denominator = 0
 .20                      then mk-number (numerator, NATURAL)
 .21                      else mk-number (numerator/denominator, RATIONAL))));
```

To convert a non-signed string, a character is removed from the front. If it is a digit, then the number is altered correspondingly. A decimal point allows the denominator to start increasing by factors of ten, assuming no previous decimal points. The function should initially be called with: the string containing the EDI value; the decimal point; 0 as the initial numerator seed, 0 as the initial denominator seed. It will return the numerator and denominator and a true flag for a valid number, and a false for an invalid number.

```
47.0  positive : (char*) × decimal × ℕ × ℕ → ℕ × ℕ × 𝔹
      positive (string, point, numerator, denominator) ≜
        if len (string) = 0
        then mk- (numerator, denominator, true)
        else cases hd (string) :
              '0' → positive (tl (string), point, numerator × 10 + 0, denominator × 10),
              '1' → positive (tl (string), point, numerator × 10 + 1, denominator × 10),
              '2' → positive (tl (string), point, numerator × 10 + 2, denominator × 10),
              '3' → positive (tl (string), point, numerator × 10 + 3, denominator × 10),
              '4' → positive (tl (string), point, numerator × 10 + 4, denominator × 10),
              '5' → positive (tl (string), point, numerator × 10 + 5, denominator × 10),
              '6' → positive (tl (string), point, numerator × 10 + 6, denominator × 10),
              '7' → positive (tl (string), point, numerator × 10 + 7, denominator × 10),
              '8' → positive (tl (string), point, numerator × 10 + 8, denominator × 10),
              '9' → positive (tl (string), point, numerator × 10 + 9, denominator × 10),
              (point.a), (point.b) → if denominator = 0
                                     then positive (tl (string), point, numerator, 1)
                                     else mk- (0, 0, false),
              others → mk- (0, 0, false)
            end;
```

The length of a number, in deciding whether it has exceeded its allowed maximum, ignores occurrences of the decimal point and minus sign. The set consisting of these 'rogue' characters is formed. Its cardinality is then subtracted from the length of the string. The function does not return the number of decimal digits in the string, but this number will coincide with the returned value for all valid numbers.

```
48.0  number-length : value × decimal → ℕ
 .1   number-length (string, point) ≜
 .2     len (string) − card (elems (string) ∩ ({'-'} ∪ {point.a} ∪ {point.b}));
```

To convert a natural number into a value of a given length, the least significant digits are added onto the front of a string, one by one, and the number is shifted to the right, until either the number has been exhausted, or the maximal length of the string would be exceeded.

```
49.0  natural-to-string : ℕ × ℤ × char* → value × 𝔹
      natural-to-string (x, length, string) ≜
 .2     if len (string) = length
 .3     then mk- (string, false)
 .4     else (if x < 10
 .5           then mk- ([numer (x + 1)] ⌢ string, true)
 .6           else natural-to-string (x div 10, length, [numer ((x mod 10) + 1)] ⌢ string));
```

To convert a general rational number is somewhat more involved. First, the integral part of the absolute value is turned into a string. If this is to be the final result (i.e. the original number was natural), or the number was to long, the calculation goes no further. (A minus sign is added, if the original number was negative.) If the calculation has survived this far, the

decimal part of the number is made into a string, to a length that fills any remaining space. A flag is set if there was not enough space for the complete decimal expansion. Trailing zeros are removed, and any minus sign is added.

```
50.0  rational-to-string : ℚ × ℕ1 × decimal → value × 𝔹 × 𝔹
 .1   rational-to-string (x, length, point) ≙
 .2     let mk-(integer-part, pass) = natural-to-string (floor (abs (x)), length, []) in
 .3     if ¬ pass ∨ x = floor (abs (x)) ∨ (x > 0 ∧ len (integer-part) = length)
 .4     then mk-(integer-part, pass, x = floor (x))
 .5     else if floor (x) = x ∨ len (integer-part) = length
 .6          then mk-("-" ⌢ integer-part, pass, x = floor (x))
 .7          else let mk-(temp-decimal-part, -) =
 .8                       natural-to-string (floor ((abs (x) − floor (abs (x))) ×
 .9                                                 (10 ↑ (length − len (integer-part)))),
 .10                                                length − len (integer-part), []),
 .11                   rounded = (0 ≠ floor (floor (x) ×
 .12                                         (10 ↑ (length − len (integer-part))))) in
 .13               let decimal-part = remove-zeros (temp-decimal-part),
 .14                   unsign = integer-part ⌢ [point.a] ⌢ decimal-part in
 .15               if rounded
 .16               then (if x = abs (x)
 .17                     then mk-(unsign, true, false)
 .18                     else mk-("-" ⌢ unsign, true, false))
 .19               else (if x = abs (x)
 .20                     then mk-(unsign, true, true)
 .21                     else mk-("-" ⌢ unsign, true, true));

51.0  remove-zeros : value → value
 .1   remove-zeros (decimal-part) ≙
 .2     let length = len (decimal-part) in
 .3     if decimal-part (length) ≠ numer (1) ∨ length = 1
 .4     then decimal-part
 .5     else remove-zeros (decimal-part (1, ..., length − 1));
```

6.2.2 'Set' handling

There follow functions which allow a certain degree of syntactical checking of the UNSM messages.

To check whether a segment instance is to be constrained, it is necessary to check that its repetition indicator tallies with an instance indicator. Any integer in the instance indicator should correspond to an equal number in the repetition type. A nil in the instance indicator makes no requirement on the repetition. The process terminates when all of the template's (or candidate's - whichever has the fewer) slots have been checked. So 1, 1, 2 will match against all of nil; (nil, nil); (nil, 1); (nil, nil, 2) but not (2); (nil, 2). Segment positions can be compared directly. (Technical Note: It is not recommended that the candidate be shorter than the template. This would only occur in badly written constraints, however.)

```
52.0  instance-match : instance-type × repetition-type → 𝔹
 .1   instance-match (template, candidate) ≙
 .2     let pass = (hd (template) = nil  ∨ hd (template) = hd (candidate)) in
 .3     if len (template) = 1 ∨ len (candidate) = 1 ∨ ¬ pass
 .4     then pass
 .5     else instance-match (tl (template), tl (candidate))

       pre  len (template) ≤ len (candidate) ;
```

To count the number of instances, the sequence of stored repetitions and positions is scanned. If the positions and instances match the stipulated ones, then one is added to the tally. If the end of the message has been reached, then the tally is returned, else the next segment is compared.

```
53.0  count : parsed-repn × parsed-posn × instance-type × position-type × ℕ → ℕ
 .1   count (candidate-repn, candidate-posn, inst, posn, tally) ≙
 .2     let pass =
 .3             hd (candidate-posn) = posn ∧
 .4             instance-match (inst, hd (candidate-repn)) in
 .5     if len (candidate-repn) = 1
 .6     then (if pass
 .7           then tally + 1
              else tally)
 .9     else if pass
 .10         then count (tl (candidate-repn), tl (candidate-posn), inst, posn, tally + 1)
             else count (tl (candidate-repn), tl (candidate-posn), inst, posn, tally);
```

To sum instances of data elements, if a match is found at a segment, and the contents of the data element are convertible to a number, then this number is added to the tally (and returned, if the end of the message is reached). Then, the next segment of the message is tried. Note that it is possible to sum over an element that is not numerical, although zero will be returned as the result. It is for other constraints to check that the elements are numerical, as required.

```
54.0  sum : data-segment* × parsed-repn × parsed-posn × decimal × element-reference × ℝ
→ ℝ
.1    sum (candidate, candidate-repn, candidate-posn, point, elt, total) ≙
.2      let match = hd (candidate-posn) = elt.position ∧
.3                  instance-match (elt.instance, hd (candidate-repn)) in
.4      if match ∧ get-segment-contents (hd (candidate), elt) ≠ nil
.5      then (let nval =
.6                  string-to-number (get-segment-contents (hd (candidate), elt),
.7                                    point).digits in
.8            if len (candidate) = 1
.9            then total + nval
.10           else sum (tl (candidate), tl (candidate-repn), tl (candidate-posn), point, elt,
.11                     total + nval))
.12     else if len (candidate) = 1
.13          then total
.14          else sum (tl (candidate), tl (candidate-repn), tl (candidate-posn), point, elt,
.15                    total);
```

To extract values from a given segment, an element reference must be given. (It should be noticed that only the simple, or composite & component references are needed for this function, and so the other parts of the element reference type may be any value - such as [],[].) The template is not consulted as to whether the relevant data element is composite or simple. It is the responsibility of the programmer to check that the constraints and the template match.

If the reference is to a simple element, and the segment is sufficiently long, then the value at the relevant position within the segment is returned. If the segment was insufficiently long, then a nil is returned, as the element is deemed to have been omitted by truncation. (No reference is made to any template at this point. Any reference to an element outside the scope of the template will return a nil, rather than an error).

If the reference is to a composite element, the relevant component of the relevant element is taken. Again, the above caveat holds, this time with regards to both the length of the composite element, and also the length of the segment.

```
55.0  get-segment-contents : data-segment × element-reference → [value]
      get-segment-contents (candidate, elt) ≙
.2      if is-simple-reference (elt)
.3      then (if len (candidate.elements) ≥ elt.simple-ref ∧
.4              (candidate.elements) (elt.simple-ref) ≠ [nil ]
.5            then (candidate.elements) (elt.simple-ref) (1)
.6            else nil )
.7      else (if len (candidate.elements) ≥ elt.composite-ref ∧
.8              (candidate.elements) (elt.composite-ref) ≠ [nil ] ∧
.9              len (candidate.elements (elt.composite-ref)) ≥ elt.component-ref ∧
.10             (candidate.elements) (elt.composite-ref) (elt.component-ref) ≠ nil
.11           then (candidate.elements) (elt.composite-ref) (elt.component-ref)
.12           else nil );
```

Given a message that has passed through the parsing process, it is then possible to extract values. The function is initially called with an empty tally. As each segment is read through, a match in both instance and position will result in the relevant value being appended to the tally. Otherwise, the tally is kept the same, and the function proceeds to the next segment.

```
56.0  get-message-contents : data-segment* × parsed-repn × parsed-posn × element-reference ×
[value]* → [value]*
 .1   get-message-contents (candidate, candidate-repn, candidate-posn, elt, tally) ≙
 .2     if candidate = []
 .3     then tally
 .4     else let new-tally =
 .5                 if hd (candidate-posn) = elt.position ∧
 .6                   instance-match (elt.instance, hd (candidate-repn))
 .7                 then tally ↷ [get-segment-contents (hd (candidate), elt)]
 .8                 else tally in
 .9        get-message-contents (tl (candidate), tl (candidate-repn),
 .10                              tl (candidate-posn), elt, new-tally);
```

It is desirable to check whether values at a given reference are numbers, and if so, whether they are integers, or natural numbers. It is also possible to ask whether the values are in a given set of rational numbers. Using the following function, it is then possible to check whether the specified instances of a data element are similarly confined.

```
number-set : value × (semigroup | SET) × [ℚ-set] × decimal → 𝔹
number-set (candidate, comparison-type, comparison-set, point) ≙
  let nval = string-to-number (candidate, point) in
  cases comparison-type :
    RATIONAL → ¬ (nval.conversion = ERROR),
    INTEGER → nval.conversion = INTEGER ∨ nval.conversion = NATURAL,
    NATURAL → nval.conversion = NATURAL,
    SET → nval.digits ∈ comparison-set
  end;
```

To check that all the relevant instances of a value are of the correct number type any segments that match the reference are checked as single numbers, as above. If any single one fails, then the function returns a false.

```
58.0  universal-number-set : data-segment* × parsed-repn × parsed-posn × element-reference ×
(semigroup | SET) × [ℚ-set] × decimal → 𝔹
 .1   universal-number-set (candidate, candidate-repn, candidate-posn, elt,
 .2                              comparison-type, comparison-set, point) ≜
 .3    let match = hd (candidate-posn) = elt.position ∧
 .4                  instance-match (elt.instance, hd (candidate-repn)) in
 .5    if len (candidate) = 1
 .6    then (¬ match) ∨ number-set (get-segment-contents (hd (candidate), elt),
 .7                          comparison-type, comparison-set, point)
 .8    elseif (¬ match) ∨ number-set (get-segment-contents (hd (candidate), elt),
 .9                          comparison-type, comparison-set, point)
.10    then universal-number-set (tl (candidate), tl (candidate-repn), tl (candidate-posn),
.11                                  elt, comparison-type, comparison-set, point)
.12    else false;
```

For string values, it is only required that the instances be in an enumerated set.

```
59.0  universal-string-set : data-segment* × parsed-repn × parsed-posn × element-reference ×
([value]-set)
                                → 𝔹
 .2   universal-string-set (candidate, candidate-repn, candidate-posn,
 .3                              elt, comparison-set) ≜
 .4     elems (get-message-contents (candidate, candidate-repn, candidate-posn, elt, [])) ⊆
 .5     comparison-set;
```

6.3 File Format Conversion

6.3.1 EDIFACT String to VDM-SL structure

The following functions are used to remove the syntax from an EDIFACT string, and mould it into a vdm flat-file format.

To parse a string of characters representing a data element (composite, simple or omitted), the characters are read off the front of the string one at a time. If it is not a component separator, or an escape character, then the character is added to the end of the last element in the sequence of values (or to the first element, if there be only one!) A component separator induces the parser to start a new component after the current one. An escape character causes the parser to read the next character (there will always be one, as the previous level of the parser will not allow an escape as the final character), and place the relevant character onto the current component. Invalid characters will flag an error. Omitted component data elements are represented by an empty sequence [], rather than a nil. This will be amended at a higher level of the parser.

```
60.0  scan-data-element : (char*) × char × char × data-element → data-element
      scan-data-element (string, colon, escape, tally) ≙
        let empty-elt = [[]],
            empty-val = [],
            last = len (tally),
            finished = (len (string) = 0) in
        if finished
        then (if tally = empty-elt
              then [nil ]
.9            else if tally (last) = empty-val
.10                then tally (1, ..., last − 1) ↷ [nil ]
.11                else tally)
.12     else cases hd (string) :
.13           (colon) → if tally (last) = empty-val
.14                       then scan-data-element
.15
.16                                 tl (string), colon, escape,
.17                                 tally (1, ..., last − 1) ↷ [nil ] ↷ empty-elt)
                          else scan-data-element (tl (string), colon, escape, tally↷empty-elt),
.19           (escape) →
.20                scan-data-element
.21
.22                     tl (tl (string)), colon, escape,
.23                     tally (1, ..., last − 1) ↷ [tally (last) ↷ [hd (tl (string))]]),
.24           others → scan-data-element
.25                                 (
.26                                  tl (string), colon, escape,
.27                                  tally (1, ..., last − 1) ↷ [tally (last) ↷ [hd (string)]])
.28        end
```

As the segment scanner moves through a character string, it adds normal characters to its buffer. When it hits a data element separator, or the end of its string, it calls the data element scanner, and adds the resulting record onto the end of its list of records. If it was not the end of the segment, then the scanner continues, having erased its buffer. An escape character will be skipped over if it is immediately followed by a data segment separator, but will be preserved in the case of component separators or a further escape, as the data element scanner will deal with these. No other character will be passed down from the message scanner.

```
61.0  scan-segment : char* × char × char × char × data-element* × char* → data-element*
 .1   scan-segment (string, plus, colon, escape, tally, buffer) ≙
 .2     if len (string) = 0
 .3     then tally ↷ [scan-data-element (buffer, colon, escape, [[]])]
 .4     else cases hd (string) :
 .5            (plus) →
 .6                 scan-segment
 .7
 .8                      tl (string), plus, colon, escape,
 .9                      tally ↷ [scan-data-element (buffer, colon, escape, [[]])], []
.10            (escape) → (cases hd (tl (string)) :
.11                               (plus) →
.12                                    scan-segment
.13
.14                                         tl (tl (string)), plus, colon, escape,
.15                                         tally, buffer ↷ [plus]),
.16                               (colon), (escape) →
.17                                    scan-segment
.18
.19                                         tl (tl (string)), plus, colon,
.20                                         escape, tally, buffer ↷ [escape] ↷ [hd (tl (string))])
.21                           end),
.22            others → scan-segment (tl (string), plus, colon, escape, tally,
.23                                   buffer ↷ [hd (string)])
.24          end;
```

To scan a string representing a sequence of data segments, there are three different classes of character. Characters other than escapes and segment terminators are peeled off the front of the string into a buffer, until either an escape or segment terminator is encountered. The latter results in the buffer being scanned as a segment, the (valid) result having its tag converted into the proper format, and adding the resulting data segment to the buffer. The new buffer is returned if the end of the string has been reached. Alternatively, the next data segment is scanned.

An escape followed by a segment separator will result in just the separator being carried through. If the following character is a different separator, or a further escape, then the escape character will be maintained, as it is necessary to the lower scanners. Any other character following an escape is invalid, and will be flagged as such. The invalid case of the string finishing after an escape is caught earlier, as is the case of any other invalid terminator.

```
62.0  scan-stream : char* × char* × data-segment* × char* → data-segment* × [char*]
 .1   scan-stream (string, cntrls, tally, buffer) ≜
 .2     let [colon, plus, -, escape, -, apost] = cntrls in
 .3     if len (string) = 0
 .4     then mk-([], "Message not terminated by legitimate character")
 .5     else cases hd (string) :
 .6           (apost) → let elemnts = scan-segment (buffer, plus, colon, escape, [], []),
 .7                         mk-(result, err) = correct-tag (elemnts (1), nil ) in
 .8                     if err ≠ nil
 .9                     then mk-([], err)
.10                     elseif len (string) = 1
.11                     then mk-(tally ⌢ [mk-data-segment (result, tl (elemnts))], nil )
.12                     else scan-stream (tl (string), cntrls, tally ⌢
.13                                       [mk-data-segment (result, tl (elemnts))],
.14                                       []),
.15           (escape) → cases hd (tl (string)) :
.16                        (apost) →
.17                            scan-stream
.18                                (
.19                                 tl (tl (string)), cntrls, tally, buffer ⌢ [apost]),
.20                        (plus), (colon), (escape) →
.21                            scan-stream
.22                                (
.23                                 tl (tl (string)), cntrls, tally,
.24                                 buffer ⌢ [escape] ⌢ [hd (tl (string))]),
.25                        others → mk-([], "Invalid character " ⌢ [hd (tl (string))] ⌢
.26                                  " after escape")
.27                     end,
.28           others → scan-stream (tl (string), cntrls, tally, buffer ⌢ [hd (string)])
.29       end;
```

To convert the first data element of a segment into a valid vdm tag, on the first instance (signalled by the tally being nil), the tag name is recovered. This is then returned, if it is the only part of the tag element, otherwise it is inserted into the tally, and removed from the element, and the next component is dealt with.

To deal with the explicit repetition indicators, each subsequent one is checked to be a number, and an error is returned if it is found to be invalid. (Technical Note:- repetitions are naturals, and so the number converter can be oblivious to the true representation of the decimal separator.) Otherwise, the numerical value is added to the tally so far. If further repetition indicators are present, these are dealt with. If not, then the correct tag is returned.

```
63.0  correct-tag : data-element × [tagtype] → [tagtype] × [char*]
      correct-tag (elt, tag-tally) ≜
 .2     if tag-tally = nil
 .3     then (if len (elt) = 1
 .4           then (if elt = [nil ] ∨ (let tg : tagnametype = hd (elt) in
 .5                     len (tg) ≠ 3)
 .6                 then mk- (nil , "Invalid Tag Name")
 .7                 else mk- (mk-tagtype (hd (elt), nil ), nil ))
 .8           else correct-tag (tl (elt), mk-tagtype (hd (elt), [])))
 .9     else let result = string-to-number (hd (elt), mk-decimal ('.', '.')) in
 .10         (if result.conversion ≠ NATURAL ∨ result.digits = 0
 .11          then mk- (nil , "Invalid Repetition Counter in Tag")
 .12          else let newtag = mk-tagtype (tag-tally.tagname,
 .13                                        tag-tally.tagvalue ⌢ [result.digits]) in
 .14               if len (elt) = 1
 .15               then mk- (newtag, [])
 .16               else correct-tag (tl (elt), newtag));
```

6.3.2 VDM-SL structure to EDIFACT string

To parse in the opposite direction, namely from a vdm structure to an EDIFACT string, values are simply copied, adding an escape character whenever needed. Data elements are read in, value by value. If an empty value is encountered, then a separator is added, otherwise the value is converted and added to the string with a separator. This is done until the data element is exhausted, or the only remaining elements are nil.

```
64.0  parse-value : value × char* × char* → char*
 .1   parse-value (val, punc, tally) ≜
 .2     let escape = punc (3),
 .3         new-hd = (if hd (val) ∈ elems (punc)
 .4                   then [escape] ⌢ [hd (val)]
 .5                   else [hd (val)]) in
 .6     if len (val) = 1
 .7     then tally ⌢ new-hd
 .8     else parse-value (tl (val), punc, tally ⌢ new-hd);


65.0  parse-element : data-element × char* × char* → char*
 .1   parse-element (elt, punc, tally) ≜
 .2     let new-tally = (if hd (elt) = nil
 .3                      then tally
 .4                      else tally ⌢ parse-value (hd (elt), punc, [])),
 .5         colon = hd (punc) in
 .6     if len (elt) = 1 ∨ elems (tl (elt)) = {nil }
 .7     then new-tally
 .8     else parse-element (tl (elt), punc, new-tally ⌢ [colon]);
```

A segment is converted into a string in a similar way, using the relevant separators. The correct format for the tag is the prepended.

```
66.0  parse-segment : data-segment × char* × char* → char*
      parse-segment (mk-data-segment (tg, elts), punc, tally) ≜
        let [colon, plus, -, apost] = punc,
            new-tally = (if elems (hd (elts)) = {nil }
                         then tally
                         else tally ↷ parse-element (hd (elts), punc, [])) in
  .6    if len (elts) = 1 ∨ elems (conc (tl (elts))) = {nil }
  .7    then convert-tag (tg, colon, []) ↷ [plus] ↷ new-tally ↷ [apost]
  .8    else parse-segment (mk-data-segment (tg, tl (elts)), punc,
  .9                        new-tally ↷ [plus]);

      convert-tag : tagtype × char × char* → char*
      convert-tag (mk-tagtype (tname, tval), colon, tally) ≜
        if tval = nil
  .3    then tname
  .4    else let seqtval : N* = tval,
  .5             mk- (nstring, -) = natural-to-string (hd (seqtval), − 1, []) in
  .6         if len (seqtval) = 1
  .7         then tname ↷ [colon] ↷ tally ↷ nstring
  .8         else convert-tag (mk-tagtype (tname, tl (seqtval)), colon,
  .9                           tally ↷ nstring ↷ [colon]);
```

It is necessary to be able to change individual elements of a segment. If the segment is already long enough, this is a simple sequence modification. If the length is insufficient, then nil elements are appended, until the correct length is reached.

```
      replace-element : data-segment × data-element × N → data-segment
  .1  replace-element (original, insert, where) ≜
  .2    let elts = original.elements in
  .3    (if len (elts) ≥ where
  .4     then mk-data-segment (original.tag, elts (1, ..., where − 1) ↷ [insert] ↷
  .5                           elts (where + 1, ..., len (elts)))
  .6     else replace-element (mk-data-segment (original.tag, elts ↷ [nil ]), insert, where));
```

To deal with data elements related to the security module, there are two implicitly defined functions. The first takes a text prompt, and a number. These are passed to an external C++ function, which will return the relevant data element. The second function passes a series of values (perhaps with some nil) to the C++ function, along with a text detailing what the values represent. The external function will return a boolean, to verify that the values were correct.

```
69.0  request-details (text : char*, how-many : Z, s-inter : Z) elemens : data-element
  .1  post true ;
```

70.0 *verify* $(vals : [value]^+, text : \mathsf{char}^+, s\text{-}inter : \mathbb{Z})$ $reply : \mathbb{B}$
post *reply*

7 Values

7.1 Miscellaneous Values

There follow the strings listing the valid alphabetic characters, and also the valid decimal digits.

values

71.0 *alpha* : char* = "*ABCDEFGHIJKLMNOPQRSTUVWXYZ* .,-()/ =' :+

72.0 *numer* : char* = "0123456789";

73.0 *dp* : *decimal* = mk-*decimal* ('.', ', ');

7.2 Templates of Service Segments

Below are coded the templates of all of the service segments, as listed in ISO 9735. They are combined together to give a directory of service segments, named service_templates.

74.0 *m-una* = {
.1 "*UNA*" ↦ [mk-*simple-template* (M, AN, F, 1),
.2 mk-*simple-template* (M, AN, F, 1),
.3 mk-*simple-template* (M, AN, F, 1),
.4 mk-*simple-template* (M, AN, F, 1),
.5 mk-*simple-template* (M, AN, F, 1),
.6 mk-*simple-template* (M, AN, F, 1)]};

```
75.0  m-unb = {
                "UNB" ↦ [mk-composite-template (M, [
.2                                      mk-simple-template (M, A, V, 4),
.3                                      mk-simple-template (M, N, F, 1)]),
.4              mk-composite-template (M, [
.5                                      mk-simple-template (M, AN, V, 35),
.6                                      mk-simple-template (C, AN, V, 4),
.7                                      mk-simple-template (C, AN, V, 14)]),
.8              mk-composite-template (M, [
.9                                      mk-simple-template (M, AN, V, 35),
.10                                     mk-simple-template (C, AN, V, 4),
.11                                     mk-simple-template (C, AN, V, 14)]),
.12             mk-composite-template (M, [
.13                                     mk-simple-template (M, N, F, 6),
.14                                     mk-simple-template (M, N, F, 4)]),
.15             mk-simple-template (M, AN, V, 14),
.16             mk-composite-template (C, [
.17                                     mk-simple-template (M, AN, V, 14),
.18                                     mk-simple-template (C, AN, F, 2)]),
.19             mk-simple-template (C, AN, V, 14),
.20             mk-simple-template (C, A, F, 1),
.21             mk-simple-template (C, N, F, 1),
.22             mk-simple-template (C, AN, V, 35),
.23             mk-simple-template (C, N, F, 1)]};
```

```
76.0  m-unz = {"UNZ" ↦ [
.1              mk-simple-template (M, N, V, 6),
.2              mk-simple-template (M, AN, V, 14)]};
```

```
77.0  m-ung = {"UNG" ↦ [
                  mk-simple-template (M, AN, V, 16),
  .2              mk-composite-template (M, [
  .3                                       mk-simple-template (M, AN, V, 35),
  .4                                       mk-simple-template (C, AN, V, 4)]),
  .5              mk-composite-template (M, [
  .6                                       mk-simple-template (M, AN, V, 35),
  .7                                       mk-simple-template (C, AN, V, 4)]),
  .8              mk-composite-template (M, [
  .9                                       mk-simple-template (M, N, F, 6),
  .10                                      mk-simple-template (M, N, F, 6)]),
                  mk-simple-template (M, AN, V, 14),
  .12             mk-simple-template (M, AN, V, 2),
  .13             mk-composite-template (M, [
  .14                                      mk-simple-template (M, AN, V, 3),
  .15                                      mk-simple-template (M, AN, V, 3),
  .16                                      mk-simple-template (M, AN, V, 6)]),
  .17             mk-simple-template (C, AN, V, 14)]};

78.0  m-une = {"UNE" ↦ [
                  mk-simple-template (M, N, V, 6),
  .2              mk-simple-template (M, AN, V, 14)]};

79.0  m-unh = {"UNH" ↦ [
                  mk-simple-template (M, AN, V, 14),
  .2              mk-composite-template (M, [
  .3                                       mk-simple-template (M, AN, V, 6),
  .4                                       mk-simple-template (M, AN, V, 3),
  .5                                       mk-simple-template (M, AN, V, 3),
  .6                                       mk-simple-template (M, AN, V, 2),
  .7                                       mk-simple-template (C, AN, V, 6)]),
  .8              mk-simple-template (C, AN, V, 35),
  .9              mk-composite-template (C, [
  .10                                      mk-simple-template (M, N, V, 2),
  .11                                      mk-simple-template (M, A, F, 1)])]};

80.0  m-unt = {"UNT" ↦ [
                  mk-simple-template (M, N, V, 6),
  .2              mk-simple-template (M, AN, V, 14)]};

81.0  m-txt = {"TXT" ↦ [
  .1              mk-simple-template (C, AN, F, 3),
  .2              mk-simple-template (M, AN, V, 70)]};

82.0  m-uns = {"UNS" ↦ [
  .1              mk-simple-template (M, A, F, 1)]};
```

```
83.0  service-templates = merge {m-una, m-unb, m-unz, m-ung,
 .1                              m-une, m-unh, m-unt};
```

7.3 Templates of Security Segments

Similarly, we code the templates of the security segments, and make a directory, named security_segments.

```
84.0  m-ush = {"USH" ↦ [
 .1             mk-simple-template (M, AN, V, 3),
 .2             mk-simple-template (M, AN, V, 3),
 .3             mk-simple-template (M, N, F, 2),
 .4             mk-simple-template (C, AN, V, 3),
 .5             mk-simple-template (C, AN, V, 3),
 .6             mk-simple-template (C, AN, V, 3),
 .7             mk-simple-template (C, AN, V, 3),
 .8             mk-simple-template (C, AN, V, 3),
 .9             mk-composite-template (C, [
.10                                    mk-simple-template (M, AN, V, 3),
.11                                    mk-simple-template (C, AN, V, 35),
.12                                    mk-simple-template (C, AN, V, 17),
.13                                    mk-simple-template (C, AN, V, 3),
.14                                    mk-simple-template (C, AN, V, 3),
.15                                    mk-simple-template (C, AN, V, 35),
.16                                    mk-simple-template (C, AN, V, 35),
.17                                    mk-simple-template (C, AN, V, 35)]),
.18            mk-composite-template (C, [
.19                                    mk-simple-template (M, AN, V, 3),
.20                                    mk-simple-template (C, AN, V, 35),
.21                                    mk-simple-template (C, AN, V, 17),
.22                                    mk-simple-template (C, AN, V, 3),
.23                                    mk-simple-template (C, AN, V, 3),
.24                                    mk-simple-template (C, AN, V, 35),
.25                                    mk-simple-template (C, AN, V, 35),
.26                                    mk-simple-template (C, AN, V, 35)]),
.27            mk-simple-template (C, AN, V, 35),
.28            mk-composite-template (C, [
.29                                    mk-simple-template (M, AN, V, 3),
.30                                    mk-simple-template (C, N, F, 8),
.31                                    mk-simple-template (C, N, F, 6),
.32                                    mk-simple-template (C, AN, V, 5)])]};
```

```
85.0  m-usa = {"USA" ↦ [
                  mk-composite-template (M, [
 .2                                           mk-simple-template (M, AN, V, 3),
 .3                                           mk-simple-template (C, AN, V, 3),
 .4                                           mk-simple-template (C, AN, V, 3),
 .5                                           mk-simple-template (C, AN, V, 3),
 .6                                           mk-simple-template (C, AN, V, 3)]),
  7               mk-composite-template (C, [
                                              mk-simple-template (C, AN, V, 512),
 .9                                           mk-simple-template (C, AN, V, 3)]),
 .10              mk-composite-template (C, [
 .11                                          mk-simple-template (C, AN, V, 512),
 .12                                          mk-simple-template (C, AN, V, 3)]),
 .13              mk-composite-template (C, [
 .14                                          mk-simple-template (C, AN, V, 512),
 .15                                          mk-simple-template (C, AN, V, 3)]),
 .16              mk-composite-template (C, [
 .17                                          mk-simple-template (C, AN, V, 512),
 .18                                          mk-simple-template (C, AN, V, 3)]),
 .19              mk-composite-template (C, [
 .20                                          mk-simple-template (C, AN, V, 512),
 .21                                          mk-simple-template (C, AN, V, 3)])]};
```

```
86.0  s500 = mk-composite-template (C, [
 .1                                    mk-simple-template (M, AN, V, 3),
 .2                                    mk-simple-template (C, AN, V, 35),
 .3                                    mk-simple-template (C, AN, V, 17),
 .4                                    mk-simple-template (C, AN, V, 3),
                                       mk-simple-template (C, AN, V, 3),
                                       mk-simple-template (C, AN, V, 35),
                                       mk-simple-template (C, AN, V, 35),
                                       mk-simple-template (C, AN, V, 35)]);
```

```
87.0  m-usc = {"USC" ↦ [
                mk-simple-template (C, AN, V, 35),
 .2             s500,
 .3             s500,
 .4             mk-simple-template (C, AN, V, 3),
 .5             mk-simple-template (C, AN, V, 3),
 .6             mk-simple-template (C, AN, V, 3),
 .7             mk-simple-template (C, AN, V, 3),
 .8             mk-simple-template (C, AN, V, 35),
 .9             mk-composite-template (C, [
 .10                                   mk-simple-template (C, AN, V, 4),
 .11                                   mk-simple-template (C, AN, V, 3)]),
 .12            mk-composite-template (C, [
 .13                                   mk-simple-template (C, AN, V, 4),
 .14                                   mk-simple-template (C, AN, V, 3)]),
 .15            mk-composite-template (C, [
 .16                                   mk-simple-template (C, AN, V, 4),
 .17                                   mk-simple-template (C, AN, V, 3)]),
 .18            mk-composite-template (C, [
 .19                                   mk-simple-template (C, AN, V, 4),
 .20                                   mk-simple-template (C, AN, V, 3)]),
 .21            mk-composite-template (C, [
 .22                                   mk-simple-template (M, AN, V, 3),
 .23                                   mk-simple-template (C, N, F, 8),
 .24                                   mk-simple-template (C, N, F, 6),
 .25                                   mk-simple-template (C, AN, V, 5)]),
 .26            mk-composite-template (C, [
 .27                                   mk-simple-template (M, AN, V, 3),
 .28                                   mk-simple-template (C, N, F, 8),
 .29                                   mk-simple-template (C, N, F, 6),
 .30                                   mk-simple-template (C, AN, V, 5)]),
 .31            mk-composite-template (C, [
 .32                                   mk-simple-template (M, AN, V, 3),
 .33                                   mk-simple-template (C, N, F, 8),
 .34                                   mk-simple-template (C, N, F, 6),
 .35                                   mk-simple-template (C, AN, V, 5)])]};

88.0  m-usr = {"USR" ↦ [
 .1             mk-composite-template (M, [
 .2                                   mk-simple-template (M, AN, V, 256),
                                      mk-simple-template (C, AN, V, 256)])]};

89.0  m-ust = {"UST" ↦ [
 .1             mk-simple-template (M, N, F, 2)]};

90.0  security-segments = merge {m-ush, m-usa, m-usc, m-usr, m-ust};
```

Internal Message Templates

To check the structure of the security envelope, we have coded the structure of figure 2 (section 10). Although not strictly a message (not being within a UNH,UNT pair) it nonetheless can be represented as a vdm UNSM.

```
    security-envelope = [
                        mk-UNSM-group
.2
.3                          "USH", C, 9, 1, [
.4                           mk-UNSM-segment ("USA", C, 1),
.5                           mk-UNSM-group
.6
.7                                "USC", C, 2, 2, [
.8                                 mk-UNSM-segment ("USA", C, 3),
.9                                 mk-UNSM-segment ("USR", C, 1)])]),
.10                 mk-UNSM-group
.11
.12                      "UST", C, 9, 3, [
.13                       mk-UNSM-segment ("USR", C, 1)])];
```

The autack message is a UNSM in the strict sense, and is encoded below.

```
92.0  autack = [
 .1          mk-UNSM-segment ("UNH", M, 1),
 .2          mk-UNSM-group
 .3               (
 .4                "USH", M, 99, 1, [
 .5                 mk-UNSM-segment ("USA", C, 1),
 .6                 mk-UNSM-group
 .7                      (
 .8                       "USC", C, 2, 2, [
 .9                        mk-UNSM-segment ("USA", C, 3),
.10                        mk-UNSM-segment ("USR", C, 1)])]),
.11          mk-UNSM-segment ("USB", M, 1),
.12          mk-UNSM-group
.13               (
.14                "USX", M, 9999, 3, [
.15                 mk-UNSM-segment ("USY", M, 1)]),
.16          mk-UNSM-segment ("UNT", M, 1)]
```

8 State

We require various global parameters: 'punctuation' holds all the separators etc. that may be altered by a UNA string; 'decimal_point' holds the value of the decimal point character - which is not simply a single character, as both a comma and a point may be concurrently valid; 'log_file' contains messages that are to be reported back to the user. The error_channel is an indicator that is used by the external error reporting message.

```
93.0  state Translator of
  .1    punctuation : char*
  .2    log-file : (char*)*
  .3    decimal-point : decimal
  .4    error-channel : ℤ
  .5    init s ≙ s = mk-Translator (": + / '", [], dp, 1)
  .6  end
```

9 Operations

Error Handling

Errors are reported via an implicit function that takes a text message (the error message), and an integer indicating the external channel upon which the error is to be reported.

UNA strings are checked for length, and then inserted into the punctuation state variable. If the decimal point character is left blank, both commas and points are allowed for the decimal separator.

operations

```
94.0  reporterr (mess : char*, errch : Z)
      post false ;

95.0  reporterror : char* ->o ()
 .1   reporterror (mess) ≙
 .2     (reporterr(mess, error-channel) );
```

To get access to non-fatal warnings, and also to find out what the translator has actually done, it is necessary to have a function that returns the contents of the log file.

```
REPORTS : () ->o char**
REPORTS () ≙
  (return log-file );
```

Handling UNA strings

```
GET-UNA : una-string ->o ()
GET-UNA (new-punctuation) ≙
 .2   (if len (new-punctuation) ≠ 9
 .3    then reporterror("Invalid UNA string")
 .4    else skip;
 .5    if new-punctuation (6) = ' '
 .6    then decimal-point := mk-decimal ('.', ',')
 .7    else decimal-point := mk-decimal (new-punctuation (6), new-punctuation (6));
 .8    punctuation := new-punctuation (4,..., 9);
 .9    log-file := log-file⌢["Punctuation changed to"⌢(new-punctuation (4,..., 9))]);
```

Separating Security Segments from Messages

To separate the security headers and trailers, any segment immediately after the UNH that has a security tag is removed, and stored in a list of headers. Anything immediately prior to the UNT segment is removed and stored in a list of trailers. This will result in a secured message, as represented in figure 1 (in section 10), being separated into a sequence of segments

conforming to figure 2 (the security envelope), along with the UNSM body surrounded by the correct UNH-UNT headers/trailers.

```
98.0  SEPARATE-SECURITY : message ⟶° data-segment* × message
  .1  SEPARATE-SECURITY (MESSAGE). ≙
  .2    (dcl security-headers : data-segment* := [],
  .3         security-trailers : data-segment* := [],
  .4         unsecure : message := MESSAGE;
  .5     while unsecure (2).tag.tagname ∈ {"USH", "USA", "USC", "USR"}
  .6     do (if ¬ correct-segment (security-segments (unsecure (2).tag.tagname),
  .7                               unsecure (2).elements, decimal-point)
  .8         then reporterror(" Security Segment " ⌢ unsecure (2).tag.tagname ⌢
  .9                   " is structurally invalid")
 .10         else skip;
 .11         security-headers := security-headers ⌢ [unsecure (2)];
 .12         unsecure := [hd (unsecure)] ⌢ tl (tl (unsecure)));
 .13    while unsecure (len (unsecure) − 1).tag.tagname ∈ {"UST", "USR"}
 .14    do let leng = len (unsecure) in
 .15       (if ¬ correct-segment (security-segments (unsecure (leng − 1).tag.tagname),
 .16                              unsecure (leng − 1).elements, decimal-point)
 .17        then reporterror(" Security Segment " ⌢ unsecure (leng − 1).tag.tagname ⌢
 .18                  " is structurally invalid")
 .19        else skip;
 .20        security-trailers := [unsecure (leng − 1)] ⌢ security-trailers;
 .21        unsecure := unsecure (1, ..., leng − 2) ⌢ [unsecure (leng)]);
 .22    return mk- (security-headers ⌢ security-trailers, unsecure) );
```

9.4 Checking Security Details

To check the security, the envelope is checked to be of the correct message structure. Then, parsing along the list of security segments, each one is checked to be of the correct segment structure. As this is going on, security links are stored, so that they can be compared. Also stored are the details to be sent to the security module: the security reference numbers; the security time stamps; and the validation values for both certificate and security result segments. If there are links in the trailers that aren't in the headers, or vice versa, then an error is flagged. If there is a degeneracy in the set of links, then this also flags an error. The final check on links is to check that the links come in the natural order. If this is violated, then a warning is flagged, as there might be some valid reason for a non-standard ordering.

The stored security details are then sent to the security module, and a security breach is reported if the module reported them to be invalid.

There then follows a list of constraints, where Security Implementation Guidelines have stipulated what values various data elements are permitted to take. Most of these simply check that coded values are in the correct set of integers. At the end, however, security structure version numbers are checked to see if they agree and are valid. Also, it is checked that sufficient security identification details are provided, in accordance with ESIG section 3.1.2

```
99.0  CHECK-SECURITY : (data-segment*) × ℤ →° ()
 .1   CHECK-SECURITY (SECURITY, s-inter) ≙
 .2     (dcl header-links : ℕ* := [],
 .3          srn : [value]* := [],
 .4          sts : [value]* := [],
 .5          trailer-links : ℕ* := [],
 .6          usr-c : [value]* := [],
 .7          usr-t : [value]* := [],
 .8          Any : instance-type := [nil ],
 .9          parse-result : char* × parsed-posn × parsed-repn := correct-message
.10
.11                                   security-envelope, SECURITY, [], [], [1], [0]);
.12     let mk- (pass, sec-posn, sec-repn) = parse-result in
        (if pass ≠ []
         then reporterror("SECURITY ENVELOPE STRUCTURE IS INVALID - "↷
                  pass)
.16      else skip;
.17      for i = 1 to len (SECURITY)
.18      by 1
.19      do (if SECURITY (i).tag.tagname = "USH"
.20          then let mk-number (lnk, -) =
.21                       string-to-number (SECURITY (i).elements (3) (1),
.22                                         decimal-point) in
.23               (header-links := header-links ↷ [lnk];
.24                srn := srn ↷
.25                       [get-segment-contents (SECURITY (i),
.26                                              mk-simple-reference (Any, [1], 11))];
.27                sts := sts ↷
.28                       [get-segment-contents (SECURITY (i),
.29                                              mk-component-reference (Any, [1], 12, 2))];
.30                sts := sts ↷
.31                       [get-segment-contents (SECURITY (i),
.32                                              mk-component-reference (Any, [1], 12, 3))];
.33                sts := sts ↷
.34                       [get-segment-contents (SECURITY (i),
.35                                              mk-component-reference (Any, [1], 12, 4))])
.36          else skip;
.37          if sec-posn (i) = [2, 2, 1]
.38          then (usr-c := usr-c ↷
.39                       [get-segment-contents (SECURITY (i),
.40                                              mk-component-reference (Any, [1], 1, 1))];
.41                usr-c := usr-c ↷
.42                       [get-segment-contents (SECURITY (i),
.43                                              mk-component-reference (Any, [1], 1, 2))])
```

```
.44         else skip;
.45         if sec-posn (i) = [1,2]
.46         then (usr-t := usr-t ↷
.47                         [get-segment-contents (SECURITY (i),
.48                                                 mk-component-reference (Any, [1], 1, 1))];
.49               usr-t := usr-t ↷
.50                         [get-segment-contents (SECURITY (i),
.51                                                 mk-component-reference (Any, [1], 1, 2))])
.52         else skip;
.53         if SECURITY (i).tag.tagname = "UST"
.54         then let mk-number (lnk, -) =
.55                         string-to-number (SECURITY (i).elements (1) (1),
.56                                           decimal-point) in
.57              trailer-links := [lnk] ↷ trailer-links
.58         else skip);
.59      if elems (header-links) ≠ elems (trailer-links)
.60      then reporterror("USH segments do not link to UST segments")
.61      else skip;
.62      if len (header-links) ≠ card (elems (header-links))
.63      then reporterror("Degeneracy amongst header-trailer links")
.64      else skip;
.65      if header-links ≠ trailer-links
.66      then log-file := log-file↷["WARNING: Security envelopes are not well-ordered"]
.67      else skip;
.68      if ¬ verify (srn, "Sequence Numbers", s-inter)
.69      then reporterror("ERROR:SECUITY BREACH: Invalid Message Sequencing")
.70      else skip;
.71      if ¬ verify (sts, "Time stamp", s-inter)
.72      then reporterror("ERROR : SECURITY BREACH : Invalid Time Stamp")
.73      else skip;
.74      if ¬ verify (usr-c, "Certificate result values", s-inter)
.75      then reporterror("ERROR:SECURITY BREACH: Certificate result values invalid")
.76      else skip;
.77      if ¬ verify (usr-t, "Trailer result values", s-inter)
.78      then reporterror("ERROR:SECURITY BREACH: Trailer result values invalid")
.79      else skip;
.80      if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.81                                 mk-simple-reference (Any, [1], 2),
.82                                 {"1", "2", "3", nil })
.83      then reporterror("Request for unknown security function")
.84      else skip;
.85      if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.86                                 mk-simple-reference (Any, [1], 4),
.87                                 {"1", "2", nil })
.88      then reporterror("Invalid Security Scope")
```

```
.89     else skip;
.90     if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.91                                mk-simple-reference (Any, [1], 5),
.92                                {"1", "2", nil })
.93     then reporterror("Invalid Response Request")
.94     else skip;
.95     if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.96                                mk-simple-reference (Any, [1], 6),
.97                                {"1", "2", "3", "4", "5", "999", nil })
.98     then reporterror("Invalid Filter Function")
.99     else skip;
.100    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.101                               mk-simple-reference (Any, [1], 7),
.102                               {"1", "2", "3", "4", "5", "999", nil })
.103    then reporterror("Invalid Character Set Encoding")
.104    else skip;
.105    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.106                               mk-simple-reference (Any, [1], 8),
.107                               {"1", "2", "3", "4", "999", nil })
.108    then reporterror("Invalid Role of Security Provider")
.109    else skip;
.110    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.111                               mk-component-reference (Any, [1], 9, 1),
.112                               {"1", "2", nil })
.113    then reporterror("Invalid Security Party Qualifier")
.114    else skip;
.115    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.116                               mk-component-reference (Any, [1], 10, 1),
.117                               {"1", "2", nil })
.118    then reporterror("Invalid Security Party Qualifier")
.119    else skip;
.120    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.121                               mk-component-reference (Any, [1], 12, 1),
.122                               {"1", nil })
.123    then reporterror("Invalid Timestamp Qualifier")
.124    else skip;
.125    log-file := log-file ↷ ["Coded elements in Headers all OK"];
.126    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.127                               mk-component-reference (Any, [1, 1], 1, 1),
.128                               {"1", "2", nil })
.129    then reporterror("Invalid use of algorithm")
```

```
.130    else skip;
.131    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.132                               mk-component-reference (Any, [1, 1], 1, 2),
.133                               {"0", "1", "2", "3", "4", "5",
.134                                "6", "7", "8", "9", "10",
.135                                "11", "12", "13", "14", "999", nil })
.136    then reporterror("Invalid Cryptographic Mode of operation")
.137    else skip;
.138    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.139                               mk-component-reference (Any, [1, 1], 1, 3),
.140                               {"1", nil })
.141    then reporterror("Invalid Mode of Operation Code List Identifier")
.142    else skip;
.143    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.144                               mk-component-reference (Any, [1, 1], 1, 4),
.145                               {"1", "2", "3", "4", "5",
.146                                "6", "7", "8", "9", "999", nil })
.147    then reporterror("Invalid Algorithm")
.148    else skip;
.149    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.150                               mk-component-reference (Any, [1, 1], 1, 5),
.151                               {"1", nil })
.152    then reporterror("Invalid code list identifier")
.153    else skip;
.154    for i = 2 to 6
.155    by 1
.156    do if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.157                                  mk-component-reference (Any, [1, 1], i, 2),
.158                                  {"1", "2", "3", "4", "5", "6",
.159                                   "7", "8", "9", "10", "11", "999", nil })
.160       then reporterror("Invalid Algorithm Parameter Qualifier")
.161       else skip;
.162    log-file := log-file ↷ ["Coded elements in Group One USA's all OK"];
.163    for i = 2 to 3
.164    by 1
.165    do if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.166                                  mk-component-reference (Any, [2, 1], i, 1),
.167                                  {"3", "4", nil })
.168       then reporterror("Invalid Security Party Qualifier")
.169       else skip;
.170    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.171                               mk-simple-reference (Any, [2, 1], 5),
.172                               {"1", "2", "3", "4", "5", "999", nil })
.173    then reporterror("Invalid Filter Function")
```

```
.174    else skip;
.175    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.176                               mk-simple-reference (Any, [2, 1], 6),
.177                               {"1", "2", "3", "4", "5", "999", nil })
.178    then reporterror("Invalid Character Set Encoding")
.179    else skip;
.180    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.181                               mk-simple-reference (Any, [2, 1], 7),
.182                               {"1", "2", "3", "5",
.183                                "6", nil })
.184    then reporterror("Invalid Character Set Repetoire")
.185    else skip;
.186    for i = 9 to 12
.187    by 1
.188    do if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.189                                  mk-component-reference (Any, [2, 1], i, 2),
.190                                  {"1", "2", "3", "4", nil })
.191       then reporterror("Invalid Separator of Signature Qualifier")
.192       else skip;
.193    for i = 13 to 15
.194    by 1
.195    do if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.196                                  mk-component-reference (Any, [2, 1], i, 1),
.197                                  {"2", "3", "4", nil })
.198       then reporterror("Invalid Date and Time Qualifier")
.199       else skip;
.200    log-file := log-file ↷ ["Coded elements in USC all OK"];
.201    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.202                               mk-component-reference (Any, [1, 2, 1], 1, 1),
.203                               {"3", "4", "5", "6", "7", nil })
.204    then reporterror("Invalid use of algorithm")
.205    else skip;
.206    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.207                               mk-component-reference (Any, [1, 2, 1], 1, 2),
.208                               {"0", "9", "10", "11", "12",
.209                                "15", "16", "999", nil })
.210    then reporterror("Invalid Cryptographic Mode of operation")
.211    else skip;
.212    if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.213                               mk-component-reference (Any, [1, 2, 1], 1, 3),
.214                               {"1", nil })
.215    then reporterror("Invalid Mode of Operation Code List Identifier")
```

```
.216  else skip;
.217  if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.218                             mk-component-reference (Any, [1,2,1], 1, 4),
.219                             {"1", "5", "6", "7", "8",
.220                              "9", "10", "11", "12", "999", nil })
.221  then reporterror("Invalid Algorithm")
.222  else skip;
.223  if ¬ universal-string-set (SECURITY, sec-repn, sec-posn,
.224                             mk-component-reference (Any, [1,2,1], 1, 5),
.225                             {"1", nil })
.226  then reporterror("Invalid code list identifier")
.227  else skip;
.228  for i = 2 to 6
.229  by 1
.230  do if ¬ universal-string-set (SECURITY, sec-repn,
.231                                sec-posn,
.232                                mk-component-reference (Any, [1,2,1], i, 2),
.233                                {"12", "13", "14", "15", "16", "17",
.234                                 "18", "19", "20", "21", "22", "23",
.235                                 "24", "999", nil })
.236     then reporterror("Invalid Algorithm Parameter Qualifier")
.237     else skip;
.238  log-file := log-file ↷ ["Coded elements in Group Two USA's all OK"];
.239  if card (elems (get-message-contents (SECURITY, sec-repn, sec-posn,
.240                                mk-simple-reference ([1], [1], 1), [])))  >
.241    1
.242  then reporterror("Multiple Security Structure Version Numbers Used")
.243  else skip;
.244  if elems (get-message-contents (SECURITY, sec-repn, sec-posn,
.245                                mk-simple-reference ([1], [1], 1), [])) ≠
.246    {"94W"}
.247  then reporterror("Invalid Security Structure Version Numbers Used")
.248  else skip;
.249  for i = 1 to len (SECURITY)
.250  by 1
.251  do (if sec-posn (i) = [1,1] ∧
.252        SECURITY (i).elements (1) (1) = "2" ∧
.253        universal-string-set (SECURITY, sec-repn, sec-posn,
.254                              mk-simple-reference (tl (sec-repn (i)), [1], 9),
.255                              {nil })
.256     then reporterror("Insufficient number of Security Identification Details")
.257     else skip)));
```

9.5 Checking Service Segments

Valid Interchanges must have a syntax identifier of UNOA, or alternatively RTOA for secured interchanges. While scanning along the segment stream that constitutes the interchange, notes are kept of the header positions (both message and functional group), as are the tally of messages and functional groups. When a message header is met, it is checked that its Message Version Number and Message Type agree with those determined in the previous functional group header.

Instances of functional group trailers trigger a checking of the number of messages within that trailer, and also a comparison with functional group reference in the corresponding header.

Likewise, instances of message trailers trigger a checking of the number of segments in the message, and a comparison of message reference numbers. Finally, the Interchange trailer is checked to have the same reference number as the Interchange header. And the counter in the trailer is checked to agree with the number of functional groups (or number of messages, if there were no functional groups) in the interchange.

```
100.0  CHECK-SERVICE : data-segment* ⟶° ()
  .1   CHECK-SERVICE (segment-stream) ≙
  .2     (dcl unh-marker : ℕ,
             ung-marker : ℕ,
             message-counter : ℕ := 0,
             group-counter : ℕ := 0;
  .6     if segment-stream (1).elements (1) (2) ≠ "2"
  .7     then reporterror("Invalid Syntax Version Number")
  .8     else skip;
  .9     for i = 2 to len (segment-stream)
  .10    by 1
  .11    do (if segment-stream (i).tag.tagname = "UNH"
  .12        then (unh-marker := i;
  .13              message-counter := message-counter + 1;
  .14              if group-counter ≠ 0 ∧
  .15                segment-stream (ung-marker).elements (7) (1) ≠
  .16                segment-stream (i).elements (2) (2)
  .17              then reporterror("UNG/UNH Message Version Number MISMATCH")
  .18              else skip;
  .19              if group-counter ≠ 0 ∧
  .20                segment-stream (ung-marker).elements (1) (1) ≠
  .21                segment-stream (i).elements (2) (1)
  .22              then reporterror("UNG/UNH Message Type MISMATCH")
  .23              else skip)
  .24        else skip;
  .25        if segment-stream (i).tag.tagname = "UNG"
  .26        then (ung-marker := i;
  .27              group-counter := group-counter + 1)
```

```
.28         else skip;
.29         if segment-stream (i).tag.tagname = "UNE"
.30         then (if string-to-number (segment-stream (i).elements (1) (1), decimal-point) ≠
.31                 mk-number (message-counter, NATURAL)
.32               then reporterror("UNE Message count MISMATCH")
.33               else skip;
.34               if segment-stream (i).elements (2) (1) ≠
.35                 segment-stream (ung-marker).elements (5) (1)
.36               then reporterror("UNG/UNE Reference Number MISMATCH")
.37               else message-counter := 0)
.38         else skip;
.39         if segment-stream (i).tag.tagname = "UNT"
.40         then (if string-to-number (segment-stream (i).elements (1) (1), decimal-point) ≠
.41                 mk-number (i + 1 − unh-marker, NATURAL)
.42               then reporterror("UNT Segment count MISMATCH")
.43               else skip;
.44               if segment-stream (i).elements (2) (1) ≠
.45                 segment-stream (unh-marker).elements (1) (1)
.46               then reporterror("UNH/UNT Reference Number MISMATCH")
.47               else skip)
.48       else skip);
.49   let unz-count =
.50         string-to-number (segment-stream (len (segment-stream)).elements (1) (1),
.51                           decimal-point) in
.52   if group-counter = 0
.53   then (if unz-count ≠ mk-number (message-counter, NATURAL)
.54         then reporterror("UNZ Message count MISMATCH")
.55         else skip)
.56   else (if unz-count ≠ mk-number (group-counter, NATURAL)
.57         then reporterror("UNZ Group count MISMATCH")
.58         else skip);
.59   if segment-stream (len (segment-stream)).elements (2) (1) ≠
.60     segment-stream (1).elements (5) (1)
.61   then reporterror("UNB/UNZ Interchange Reference MISMATCH")
.62   else log-file := log-file ↷ ["Service Links OK"]);
```

9.6 Handling Incoming Messages

To deal with an incoming EDIFACT string, the whole string is converted to a sequence of data segments, after any UNA string has been dealt with. The presence of interchange headers and trailers is checked, and the interchange is then checked to see which it contains: either functional groups; or messages. These are then parsed into the correct grouping, and any inability to do this, due to being given an invalid string, is flagged as an error. The resulting structure is then given the record type of EDIFACT_VDM, the chosen 'flat-file' structure. The sequence of data segments is then scanned. Anything not inside a UNH..UNT pair is deemed to be a service segment, and is checked as such. Anything not outside a UNH..UNT pair is

deemed to be a message conforming to figure 1 (section 10). The security envelope is stripped off, leaving a UNSM message, and a security envelope conforming to figure 2.

```
101.0  INCOMING : char* × ℤ × ℤ ⟶° plain-interchange × data-segment*+
       INCOMING (interchange-string, s-inter, errch) ≙
  .2     (dcl segment-stream : data-segment*,
  .3          i : ℕ,
  .4          unh-marker : ℕ,
  .5          ung-marker : ℕ,
  .6          functional-flag : 𝔹 := false,
  .7          message-list : message* := [],
  .8          EDIFACT-STRING : char* := interchange-string,
  .9          SECURITY-TALLY : data-segment** := [],
  .10         fn-grp-list : functional-group* := [],
  .11         EDIFACT-VDM : plain-interchange;
  .12    error-channel := errch;
  .13    if len (EDIFACT-STRING) ≤ 3
  .14    then reporterror("Edifact string too short")
  .15    else skip;
  .16    if EDIFACT-STRING (1,...,3) = "UNA"
  .17    then (GET-UNA(EDIFACT-STRING (1,...,9)) ;
  .18          EDIFACT-STRING := interchange-string (10,...,len (interchange-string)))
  .19    else skip;
  .20    let mk- (interchange-segments, error-string) =
  .21          scan-stream (interchange-string, punctuation, [], []) in
  .22    if error-string ≠ nil
  .23    then reporterror(error-string)
  .24    else segment-stream := interchange-segments;
  .25    if segment-stream (1).tag.tagname ≠ "UNB" ∨
  .26      segment-stream (len (segment-stream)).tag.tagname ≠ "UNZ"
  .27    then reporterror("Interchange is not contained within UNB UNZ delimiters")
  .28    else skip;
  .29    if segment-stream (2).tag.tagname = "UNG"
  .30    then functional-flag := true
  .31    else skip;
  .32    if segment-stream (2).tag.tagname ≠ "UNH" ∧ ¬ functional-flag
  .33    then reporterror("Incorrect Message/Functional Group Header")
  .34    else skip;
  .35    if ¬ functional-flag
  .36    then (unh-marker := 2;
  .37          if segment-stream (len (segment-stream) − 1).tag.tagname ≠ "UNT"
  .38          then reporterror("Missing UNT Segment")
  .39          else skip;
               for i = 2 to len (segment-stream) − 1
               by 1
```

```
.42         do (if segment-stream(i).tag.tagname = "UNT"
.43             then (message-list := message-list^[segment-stream(unh-marker, ..., i)]
.44                   if i ≠ len (segment-stream) - 1 ∧
.45                     segment-stream(i + 1).tag.tagname ≠ "UNH"
.46                   then reporterror("Invalid Segment Following UNT")
.47                   else skip;
.48                   unh-marker := i + 1)
.49             else skip);
.50         EDIFACT-VDM := mk-plain-interchange
.51
.52                                    segment-stream(1), message-list,
.53                                    segment-stream(len (segment-stream)));
.54         log-file := log-file ^ ["Converted to flat file OK"])
.55     else (unh-marker := 3;
.56         ung-marker := 2;
.57         if segment-stream(3).tag.tagname ≠ "UNH"
.58         then reporterror("Functional Group contains Invalid Message")
.59         else skip;
.60         if segment-stream(len (segment-stream) - 1).tag.tagname ≠ "UNE"
.61         then reporterror("Missing UNE Segment")
.62         else skip;
.63         for i = 2 to len (segment-stream) - 1
.64         by 1
.65         do (if segment-stream(i).tag.tagname = "UNT"
.66             then (message-list := message-list^[segment-stream(unh-marker, ..., i)];
.67                   if i ≠ len (segment-stream) - 2 ∧
.68                     segment-stream(i + 1).tag.tagname ≠ "UNH" ∧
.69                     segment-stream(i + 1).tag.tagname ≠ "UNE"
.70                   then reporterror("Invalid Segment Following UNT")
.71                   else skip;
.72                   unh-marker := i + 1)
.73             else skip;
.74             if segment-stream(i).tag.tagname = "UNE"
.75             then (fn-grp-list := fn-grp-list ^
.76                                  [mk-functional-group
.77                                      (
.78                                       segment-stream(ung-marker), message-list,
.79                                       segment-stream(i))];
.80                   if i ≠ len (segment-stream) - 1 ∧
.81                     segment-stream(i + 1).tag.tagname ≠ "UNG"
.82                   then reporterror("Invalid Segment Following UNE")
.83                   else skip;
.84                   if i ≠ len (segment-stream) - 1 ∧
.85                     segment-stream(i + 2).tag.tagname ≠ "UNH"
.86                   then reporterror("Invalid Segment Following UNG")
```

```
.87                 else skip;
.88                 fn-grp-list := fn-grp-list ^
.89                                 [mk-functional-group
.90
.91                                     segment-stream (ung-marker), message-list,
.92                                     segment-stream (i))];
.93                 message-list := [];
.94                 unh-marker := i + 2;
.95                 ung-marker := i + 1)
.96             else skip);
.97         EDIFACT-VDM := mk-plain-interchange
.98                                 (
.99                                  segment-stream (1), fn-grp-list,
.100                                 segment-stream (len (segment-stream)));
.101        log-file := log-file ^ ["Converted to flat file OK"]);
.102    CHECK-SERVICE(segment-stream) ;
.103    i := 1;
.104    while i ≤ len (segment-stream)
.105    do (if ¬ correct-segment (service-templates (segment-stream (i).tag.tagname),
.106                              segment-stream (i).elements, decimal-point)
.107        then reporterror("INVALID SEGMENT "^segment-stream (i).tag.tagname)
.108        else log-file := log-file^["Service Segment "^segment-stream (i).tag.tagname^
.109                          " OK"];
.110        if segment-stream (i).tag.tagname ≠ "UNH"
.111        then i := i + 1
.112        else (unh-marker := i;
.113              while segment-stream (i).tag.tagname ≠ "UNT"
.114              do i := i + 1;
.115              let mk-(SECURITY, UNSECURED-MESSAGE) =
.116                    SEPARATE-SECURITY (segment-stream (unh-marker, ..., i)) in
.117              (if SECURITY ≠ []
.118               then (CHECK-SECURITY (SECURITY, s-inter) ;
.119                     SECURITY-TALLY := SECURITY-TALLY ^ [SECURITY])
.120               else (log-file := log-file ^ ["Unsecured message"];
.121                     SECURITY-TALLY := SECURITY-TALLY ^ [nil ]);
.122                cases external-UNSM (UNSECURED-MESSAGE):
.123                  NOT_FOUND → log-file := log-file ^
.124                                  ["Couldn't find message in database"],
.125                  FAIL → reporterror("Message failed against external database") ,
.126                  PASS → log-file := log-file ^ ["Message matched against database"]
.127               end;
.128               for j = 2 to len (UNSECURED-MESSAGE) − 2
.129               by 1
.130               do cases external-segment (UNSECURED-MESSAGE (j),
.131                                          decimal-point):
.132                    NOT_FOUND → log-file := log-file ^
.133                                          ["Couldn't find segment in database"],
```

```
.134                FAIL →
.135                    reporterror("Segment failed against  external database"),
.136                others → skip
.137            end)));
.138    return mk-(EDIFACT-VDM, SECURITY-TALLY) );
```

9.7 Handling Outgoing Messages

To send a secured interchange, a plain interchange is fed in along with a sequence of optional security envelopes. If there are any security envelopes to be added, the syntax identifier in the interchange header is changed to RTOA, to signal conformance to ESIG. The interchange structure is flattened out by streaming the contents of the functional groups, and the message contents, sequentially. If there were any security envelopes to be added, they are added one at a time in the relevant messages delimited by UNH..UNT pairs. Note that the queue of security envelopes to be added is strictly one dimensional, whereas the sequence of messages may be two dimensional, in that they may be gathered in functional groups. So, for instance, if an interchange consisted of two functional groups, the first having one message, and the second having two, the third security envelope would be attached to the second message of the second functional group.

```
102.0  OUTGOING : plain-interchange × [data-segment+]* × ℤ →o char*
       OUTGOING (flat-file, security-envelopes, s-inter) ≜
  .2     (dcl segment-stream : data-segment*,
  .3          output-tally : char* := [],
  .4          unh-marker : ℕ := 1,
  .5          unt-marker : ℕ;
  .6     segment-stream := [flat-file.interchange-header];
  .7     if security-envelopes ≠ [] ∧ elems (security-envelopes) ≠ {nil }
  .8     then let unb = segment-stream (1) in
  .9          segment-stream := [mk-data-segment (unb.tag,
  .10                                              [["RTOA"] ↷
  .11                                               [tl (hd (unb.elements))]] ↷
  .12                                              tl ([unb.elements]))]
  .13    else skip;
  .14    if ¬ is-functional-group (flat-file.interchange-body (1))
  .15    then segment-stream := segment-stream ↷ conc (flat-file.interchange-body)
  .16    else for i = 1 to len (flat-file.interchange-body)
  .17         by 1
  .18         do let ffb = flat-file.interchange-body (i) in
  .19            (segment-stream := segment-stream ↷ [ffb.functional-header];
  .20             segment-stream := segment-stream ↷ conc (ffb.functional-body);
  .21             segment-stream := segment-stream ↷ [ffb.functional-trailer]);
  .22    segment-stream := segment-stream ↷ [flat-file.interchange-trailer];
  .23    if security-envelopes ≠ [] ∧ elems (security-envelopes) ≠ {nil }
  .24    then for i = 1 to len (security-envelopes)
  .25         by 1
```

```
.26         do (while segment-stream(unh-marker).tag.tagname ≠ "UNH" ∧
.27                   unh-marker < len (segment-stream)
.28             do unh-marker := unh-marker + 1;
.29             unt-marker := unh-marker;
.30             if unh-marker = len (segment-stream)
.31             then reporterror("More envelopes than messages!")
                else while segment-stream (unt-marker).tag.tagname ≠ "UNT"
                     do unt-marker := unt-marker + 1;
.34             if hd (security-envelopes) ≠ nil
.35             then let insert = ADD-SECURITY (segment-stream (unh-marker,
.36                                                          ..., unt-marker),
.37                                                 security-envelopes (i), s-inter) in
.38                  segment-stream := segment-stream (1,..., unh-marker−1)⌢insert⌢
.39                                    segment-stream (unt-marker + 1,
.40                                                    ..., len (segment-stream))
.41             else skip)
.42     else log-file := log-file ⌢ ["Outgoing interchange has no security added"];
.43     let parse-chars = punctuation (1,..., 2)⌢[punctuation (4)]⌢[punctuation (6)] in
.44     for i = 1 to len (segment-stream)
.45     by 1
.46     do output-tally := output-tally⌢parse-segment (segment-stream (i), parse-chars, []);
.47     return output-tally );
```

9.8 Adding Security to a Message

To add a security envelope to a UNSM message, it is first necessary to check that the envelope conforms to the template given in figure 2 (section 10). It is then necessary for the translator to gather the security details it requires, namely: security reference number; time stamp; security results pertaining to group two; and security results pertaining to the trailer. These details are fed into the relevant slots in the template, and once a segment has all the details it requires, it is checked to see if its elements are valid.

If the security segment is a component of group one, then it is inserted into the message stream immediately prior to the UNSM contents. If it is part of security the trailer group, it is inserted immediately prior to the UNSM trailer. When all segments corresponding to that message have been added in, the UNT register that counts the number of segments within a message is updated, and the new message returned.

```
103.0  ADD-SECURITY : message × data-segment+ × ℤ ⟶° message
  .1   ADD-SECURITY (unsecure, envelope, s-inter) ≜
  .2     (dcl parse-result : char* × parsed-posn × parsed-repn := correct-message (security-envelope,
  .3                                                  envelope, [], [], [1], [0]),
           new-message : data-segment* := unsecure,
  .5       new-secure-seg : data-segment;
         let mk-(pass, sec-posn, -) = parse-result in
         (if pass ≠ []
          then reporterror("Added Security Envelope structure is invalid - " ⌢ pass)
```

```
.9      else skip;
.10     for i = 1 to len envelope
.11     by 1
.12     do (new-secure-seg := envelope (i);
.13         if sec-posn (i) = [1]
.14         then let ref = request-details ("SECURITY REFERENCE", 1, s-inter
.15                  new-segment = (if elems (ref) = {nil }
.16                                 then envelope (i)
.17                                 else replace-element (envelope (i), ref, 11)),
.18                  ref2 = request-details ("DATE AND TIME", 3, s-inter),
.19                  newer-segment = (if elems (ref2) = {nil }
.20                                   then new-segment
.21                                   else replace-element (new-segment,
.22                                                         ["1"] ↷ ref2, 12)) in
.23              new-secure-seg := newer-segment
.24         else skip;
.25         if sec-posn (i) = [2, 2, 1]
.26         then let ref = request-details ("GROUP TWO RESULT", 2, s-inter),
.27                  new-segment = (if elems (ref) = {nil }
.28                                 then envelope (i)
.29                                 else replace-element (envelope (i), ref, 1)) in
.30              new-secure-seg := new-segment
.31         else skip;
.32         if sec-posn (i) = [1, 2]
.33         then let ref = request-details ("TRAILER RESULT", 2, s-inter),
.34                  new-segment = (if elems (ref) = {nil }
.35                                 then envelope (i)
.36                                 else replace-element (envelope (i), ref, 1)) in
.37              new-secure-seg := new-segment
.38         else skip;
.39         if ¬ correct-segment (security-segments (envelope (i).tag.tagname),
.40                               envelope (i).elements, decimal-point)
.41         then reporterror(" Security Segment " ↷ envelope (i).tag.tagname ↷
.42                  " is structurally invalid")
.43         else skip;
.44         let leng = len (new-message) in
.45         if sec-posn (i) (len (sec-posn (i))) = 1
.46         then new-message := new-message (1, ..., i) ↷ [new-secure-seg] ↷
.47                             new-message (i + 1, ..., leng)
.48         else new-message := new-message (1, ..., leng − 1) ↷ [new-secure-seg] ↷
.49                             [new-message (leng)]);
.50     let unt-seg = new-message (len (new-message)),
.51         seg-cnt = unt-seg.elements (1) (1),
.52         mk-number (dgts, -) = string-to-number (seg-cnt, decimal-point),
.53         mk- (new-cnt, pass) = natural-to-string (dgts + len (envelope), 6, []),
```

```
.54         new-unt = replace-element (unt-seg, [new-cnt], 1) in
.55     if ¬ pass
.56     then reporterror("Segment count taken above limit")
.57     else new-message := new-message (1, ..., len (new-message) − 1) ↷ [new-unt];
.58     return new-message ))
```

10 Diagrams

Figure 1

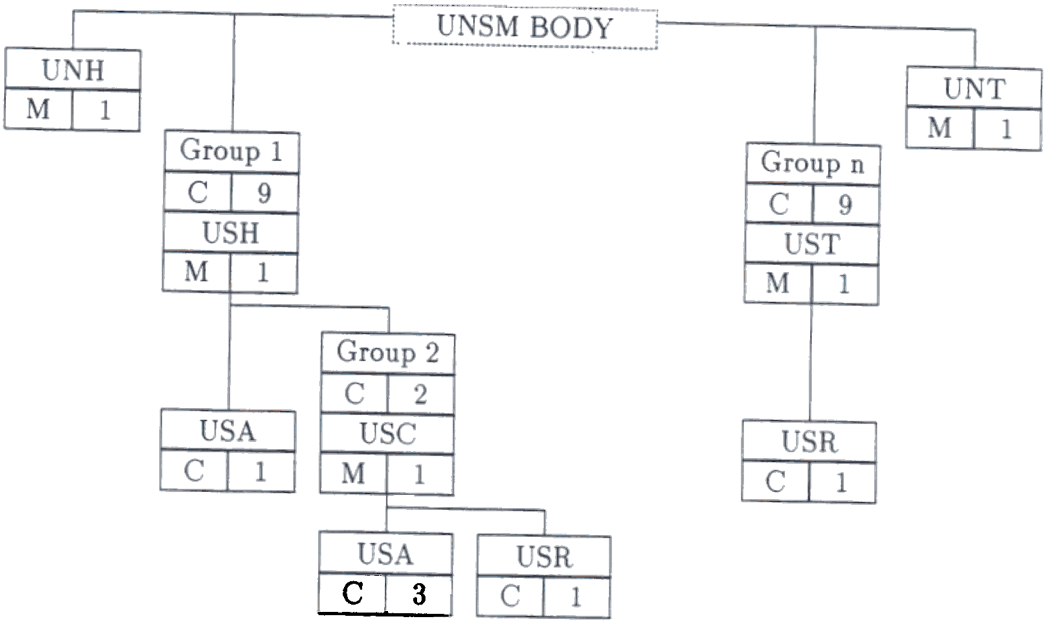
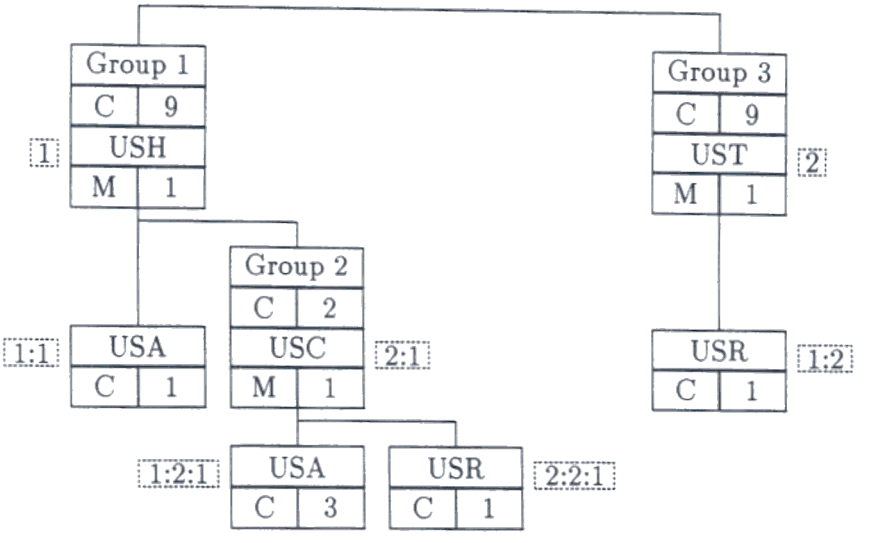


Figure 2.



Index

Acknowledgments

This programme of research was jointly sponsored by the Communications and Information Industries Directorate of the Department of Trade and Industry, and by the Defence Research Agency on behalf of the Ministry of Defence.

The reference implementation was developed using VDM-SL tools from IFAD [7]. The reference implementation was developed under a task led by G Parkin of NPL, and his contribution to the development of the specification is gratefully acknowledged.

References

[1] S Hehir. Validation of the EDIFACT Test Suite - Considerations. Technical Report DSG/D/359, National Physical Laboratory, December 1995.

[2] John Dawes. *The VDM-SL Reference Guide.* Pitman, 1991.

[3] P. G. Larsen, B. S. Hansen, H. Brunn N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language, December 1996.

[4] ISO. *ISO 9735 EDIFACT Application level syntax rules*, 1988(E).

[5] UN/EDIFACT Security JWG. *EDIFACT Security Implementation Guidelines.*, December 1993.

[6] UN/EDIFACT Security JWG. *MIG Handbook - UN/EDIFACT Message AUTACK*, February 1994.

[7] IFAD The VDM-SL Tool Group. *User Manual for the IFAD VDM-SL Toolbox*, May 1996. IFAD; Forskerparken 10; DK-5230 Odense M; Denmark.

[8] UN/EDIFACT. *UNTDID Part 4 Syntax Implementation Guidelines.*

A Interfacing the Specification

This section describes the programs auxiliary to the VDM-SL specification which allow its execution. There are two programs directly related to the VDM-SL: a main one to call the various high-level operations in VDM-SL; and a secondary one that has explicit C++ routines that carry out various functions that were only implicitly defined in the VDM-SL.

There is another program that handles the connections to the network over which the tests are run.

A.1 The Main Auxiliary Program

Main expects three integer parameters that are the network socket numbers to the upper, lower, and security channels. If data comes in on the lower interface, the program treats this as an incoming EDIFACT string, and passes the received data on to VDM-SL operation INCOMING, along with the socket numbers on which the VDM-SL must communicate with the security module and the output module (in this case, the upper interface).

If data comes in on the upper interface, the program expects a request for some form of security to be added to an interchange. For the purposes of the test suite, it is only necessary for the translator to reply whether or not it supports the requested service. The program is easily fine-tuned, in that it is a simple adjustment to make the translator respond to a request for a service with an "OK", rather than an "ERROR". This is clearly not what is required of a real secure translator. It would be a simple matter, however, to alter the interface so that when a security service is requested, it calls the OUTGOING VDM-SL operation with parameters that feed in appropriate security envelope skeletons.

If the data comes from the security module, this is signalled as an error. Security related information should only be sent from the security module as the result of a request from the translator for data.

A.2 Implementation of Implicit VDM-SL Functions

There were several functions that needed to interact externally. To write these, they were coded by implicit VDM-SL functions. These would be skipped over by the C++ generator, and could be written by hand in C++.

The error handling function receives a string containing the text of the error message, and a socket number. It then simply writes the error message on the channel given by the socket number, and exits execution.

There are two functions interfacing the security module: *verify* and *request_details*. *verify* takes a sequence of values; a text string; and a socket number (of the security module channel). It then sends the text string to the security module, which informs the security module of what the following data is (Time Stamp, Signature, etc.), followed by the relevant data elements. When it has finished sending the data, it reads the verdict from the security module. It then returns a boolean version of this verdict (fail if the security module verdict began with an 'n') to the VDM-SL. In a real implementation *request_details* would function in a different way: it would send details of how the relevant security value had been calculated originally, in order that the security module could repeat this calculation. The implementation would then await

the result of this calculation, and compare it to the value that was contained in the message in order to reach a verdict on the validity of the message's security. To specify this behaviour would involve additional coding in: extracting the portion of the EDIFACT string upon which the Validation Result ([5] S508) had calculated; and specifying a protocol for passing, to the security module, details of which cryptographic algorithms, parameters, *etc.* had been used.

request_details takes a text string; number of values required; and the security channel socket number. It sends out the text string to the security module, which gives the security module a description of the data it requires to be sent back (Time Stamp, Signature, etc.). It then reads in the required number of values, converting the incoming strings to VDM-SL types, building up a Sequence of them, empty sequences being converted to the null type. This sequence is then returned to the VDM-SL.

Finally, there are two functions to access the databases of templates. There should be two databases: one for UNSMs, and one for data segments. These databases have not been built; and so, regardless of the name sent to them, they will return a null value, which is the value that the Translator expects back when the segment (or UNSM) was not found in the relevant database. If databases were to be built, these functions would be altered so that if the segment tagname (UNSM name) were to be found in the database, the VDM-SL segment template (UNSM template) would be returned.

A.3 Network Program

The translator interacts with the test suite by the TCP/IP protocol. The *network* program listens to the three EDI channels (Upper, Lower, and Secure) and opens connections when it hears that the three channels are ready. When it has confirmed the connection, and received the socket numbers for the three channels, it executes the *Main* program (above) as a child process, with the socket numbers as parameters. When the *Main* program terminates, the *network* program loops back to obtain a new set of connections, before running the *Main* program again (for the next test).