The reverseXSL Transformer

Message DEFinition Files

V2.0 May09 **web release**

V2.1 Apr12 Now as Open Source

Foreword

This document describes the DEF file format used by the reverseXSL Parser component. It describes how to define the syntax and structure of any character-based message, and explains how the transformation to XML can be controlled.

Table of Contents

1. What You Should Know 3

2. Design Principles 4

3. Operations 7

3.1 Transformer Overview 7

4. DEF File Structure 9

4.1 General Layout 9

4.2 What is a Segment (noted SEG)? 10

4.3 What is a Group (noted GRP)? 11

4.4 What is a Data Element (noted D)? 12

4.5 What is a Named Condition (noted COND)? 13

4.6 What is a Mark (noted MARK)? 15

4.7 What is the Effect of min/max & Accept Loop Counts? 16

4.8 How is Data actually Separated from the Message's Syntax? 18

4.9 Delimiter or Data? Using a Release Char 22

4.10 Using Reserved XML tags: NOTAG, RAW and SKIP 23

4.11 Using the Reserved NULL Value 25

4.12 Exception Handling 25

5. DEF File Line Formats 27

5.1 SEG (segment) Definitions 27

5.2 D (data element) Definitions 30

5.3 GRP (group) Definitions 32

5.4 MSG & END Definitions 33

5.5 COND Definitions 33

5.6 MARK Definitions 34

5.7 Comments 35

6. Namespaces (SET BASENAMESPACE) 36

7. Advanced Questions 38

8. Sample Program Code 48

9. Command Line Tools 50

10. Philosophical Considerations 51

11. Known Issues and Limitations 55

12. APPENDIX: Regular Expressions 59

updates available from:
www.reverseXSL.com

Copyright
Art of e.biz
(except for the APPENDIX)

Formally…

ReverseXSL.com is a trademark of Art of e.Biz, an independent software editor with registered offices in Belgium.

The present document is not contractual and does not bind Art of e.Biz to meet the functions and features as exactly described in the present document. Although care as been taken to reflect to the best extent possible the design and features of the product described herewith, Art of e.Biz reserves the right to modify the enclosed descriptions without notice.

Art of e.Biz disclaims any explicit or implied warranties or fitness for purpose. The company and the author(s) of the present document will not be liable for any loss of profit or any other commercial damage.

Product names, logos, brands, and other trademarks cited, featured, or referred to within the present document and all associated materials, do remain the property of the respective trademark holders.

These trademark holders are not affiliated with ReverseXSL.com or Art of e.Biz, neither the related products, nor the website. They do not sponsor or endorse our materials.

No part of this document can be copied or reproduced or transmitted in any form and by any means without the permission of the respective authors or its copyright owners except within the scope of the reverseXSL Open-source Licensing Agreement which is Apache 2.0 .

www.reverseXSL.com

1. What You Should Know

*The present document is **reference material**. Tutorials, samples, trial exercises as well as overviews are exclusively published at the web site.*

We strive to provide you with the best documentation available. There's hardly anything more frustrating than starting to lose time by the lack of proper information.

This manual describes only <u>one</u> piece of meta-data within the reverseXSL Transformation software. We advise you to start your journey to the documentation from the web site www.reverseXSL.com > <u>documentation</u>, if not already the case.

The present document describes only this piece

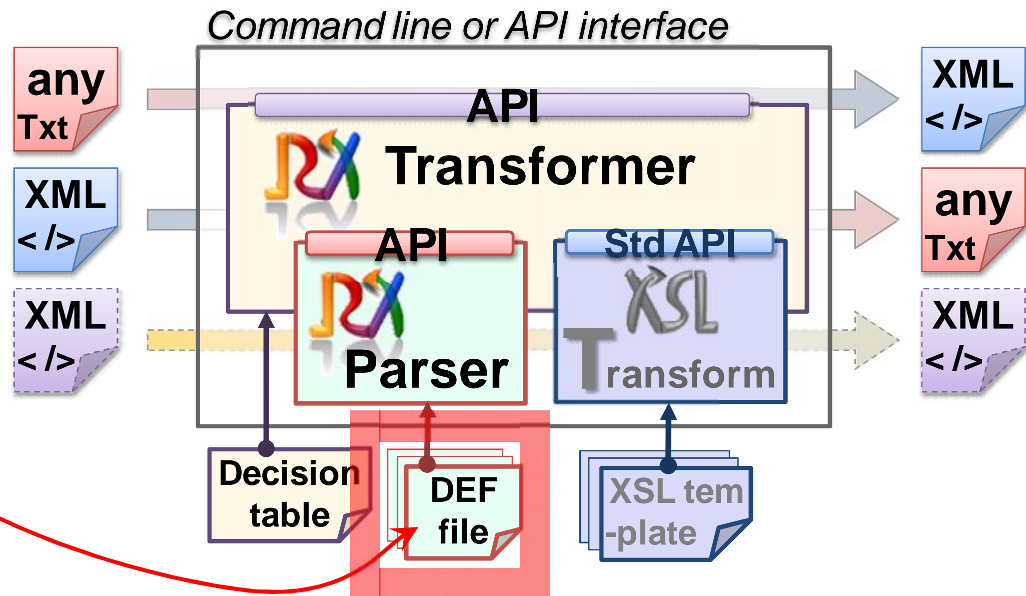

Figure 1. reverseXSL software components

Prerequisites:

Regular expressions are found in numerous computer languages and scripts. We use the implementation made available in Java runtimes since version 1.4, itself inherited from the UNIX egrep and sed shell commands.

- An understanding of **regular expressions** is a must, and in particular the use of **capturing groups** in regular expressions. Don't try to guess their meanings and behaviour, but learn carefully the meaning of the following 11 special characters . * + ? ([\ { ^ \ and | (plus of course the closing)] }). You will save yourself from a lot of frustration, spare much time, and possibly begin to grasp their magical power. We provide a set of links below to help with learning.
- An understanding of the **message formats** that you have to process is also a must, whether these are application-specific, corporate conventions, or EDI standards. It may seem stupidly obvious to raise the point, but again, we'd like to raise the importance of knowing precisely the syntax variations and dependency conditions in the message structures at stake in order to formalize efficiently the parsing. Every printable or control character that can occur in a message has to be either explicitly matched, or explicitly ignored. There is no implicit behaviour because that would imply the possibility of silently missing information!

  We endeavour to document samples from as many formats as possible on the web site.

- Third requirement? You should know about XML and XSLT of course; if you read this documentation and evaluate the reverseXSL software, there's a very high chance that this is already the case, or that it will soon be…

With the above, you are ready to go.

Useful links:

The one-hour Regex tutorial at www.reverseXSL.com.

Dedicated to the quickest start with the reverseXSL Parser.

http://docs.oracle.com/javase/tutorial/essential/regex/

Regular Expressions and the Java Programming Language.

2. Design Principles

This section is fairly formal. As evoked above, this document is a reference manual. Feel free to skip it during a first reading.

As it should be, we formalized requirements before writing the software. Although we could have archived today this section, we decided to preserve and even maintain it, because it is quite efficient in deciding whether this software can meet your own requirements or not.

[1] A DEF file is by design a **one-way** description of an inbound message structure (character-based data, printable or not) and of its transformation into a canonical XML representation. A DEF file cannot be used to map back an XML into its original syntax.

[2] The class of messages that the software can parse shall encompass all character-based data interchange standards and de-facto formats. Precisely, any message which can be interpreted by drilling down into hierarchically nested and sequentially concatenated structures, possibly mixing fixed size and delimited fields in any order. Delimiters can be implicit, alike in a numerical to alphabetical transition for instance. Delimiters at any point can themselves be variable, and belong to a range of possibilities. The correct Interpretation of data can mix multiple techniques: explicit tagging, pattern-based, positional, inter-dependent from other element values or structures, and of course context-sensitive (i.e. depending from the whole sequence of already parsed elements). Formally, this is the class of left-to-right scanning context-sensitive grammars from the theory of formal languages.

[3] The handling of binary-coded message syntaxes and/or with binary data representations, like based on ASN.1 for instance, falls out of scope.

[4] The reverse transformation from XML to text-based documents shall be performed by XSLT transformations.

www.reverseXSL.com

An XML representation for a DEF file is announced, such as to extend automation facilities like XML schema generation.

An 'include' statement is announced in a forthcoming release.

[5] A DEF file contains enough information to completely validate the syntax of an inbound message.

[6] The DEF file has itself a structure and syntax that privilege the use of a plain text editor (Wordpad, vi, text editors embedded in collaborative development workbenches like Eclipse, StylusStudio, XMLSpy, etc.), with a concern for compact editing. Hence, it was decided that a DEF file is not defined as an XML document itself (alike XSL templates for XSLT). In addition, a DEF file contains numerous regular expressions that would require the use of quoting transcripts (e.g. &) if used in XML, further making the regular expressions more cryptic than necessary and forcing to develop a custom editor.

[7] A DEF file is self-containing, autonomous and linear[1]. This requirement was guided both by efficiency considerations, and operational constraints. There is no provision (other than cut-and-paste editing) to share segment definitions or groups from a library of standard segments for instance and re-use them in many DEF files, or several times within the same DEF file. Such facilities will be offered, if any, by a DEF file-editing environment that in turn, will compile autonomous linear DEF files as described here, from various re-usable objects.

[8] A DEF file matches the definition of one and only one message. Shall multiple messages follow each other within a single 'big' file or within the data portion of a communication protocol data unit, other external functions are required to slice such big file or data into the individual 'messages'. Those functions are not covered by this specification. As a consequence and in the present context, any data characters left after the declared[2] message END will yield an error.

[9] Inter-dependency conditions between data-elements and/or complete segments or groups are supported but only for the validation of existence/non-existence dependencies, possibly linked to some coded element combinations. Other dependencies based on numerical comparisons between two elements, or date comparisons, or the verification of an addition, shall be implemented by the end applications.

[10] The proposed solution does not validate control values like the total count of segments in the message versus a control count. This shall be implemented in header/enveloping processing.

[11] The parsing may continue after the first (or any number) of errors encountered. Exceptions (errors) are either Recorded or Thrown. Whether to record or throw an exception is indicated with every

[1] No section of the file does require navigating to several other places in the file to fully understand its meaning. Every definition is fully specified at the place where it is defined; sequentially, from begin to end.

[2] One may **declare** extra segments to accept garbage data as explained later on. This must be explicit. This rule tells that there's no implicit acceptance of extra characters.

condition in the message definition. The maximum number of acceptable exceptions (after which any additional exception will be thrown) is passed as argument to the parser.

[12] In addition to the above, exceptions can be classified into Fatal and Warning exceptions. Hence one can specify a maximum number of recorded fatal exceptions, or of total recorded exceptions. A limit on the number of recorded warning exceptions alone does not make sense, because a fatal case is at least warning!

Forthcoming 'envelope processing' functionality will do the job in a future release.

[13] There's no mean to deal with legacy EDI escape character sequences in the parser itself (e.g. "?+" in EDIFACT for a literal "+" character). The replacement of conflicting characters (or substrings) in data values shall be performed before calling the parser. For instance, removing character-escape sequences and replacing all separators by control characters (e.g. in EDIFACT using character codes IS1, IS3, IS4 as officially defined), or any other combination of pre/post mappings.

[14] One shall also not forget that the XML character set is quite restrictive. The use of XML escape-sequences (& > < etc.) and then the encoding of data element values that will be mapped into the target XML document is delegated to the Parser itself, at the point of rendering in XML a copy of its internal message element tree. XML encodings are therefore automatically applied as needed.

[15] The parser is using internally the UNICODE character set[3]. All string handlings are based on character counts and not byte counts. The encoding from ASCII (or UTF-8 or else) to UNICODE, or from UNICODE to ASCII (or other) is performed externally to the parser component in proper (see the upper level Transformer and TransformerFactory for relevant facilities).

[16] The target XML document is always a 'simple' XML: by definition, generated documents contain a strict hierarchy of elements. Any element can bear attributes. Each element contains only ordered sequences of sub-elements, or (exclusively) textual data. There are no mixes of character data and child-elements within the same parent element (as in <E1>some data<E2>more data</E2></E1>), although fully allowed by XML standards. Moreover, no use is made of processing instructions, references, or inclusions.

[17] All XML document elements are qualified with a namespace. They are generated by reference to a single target *base* namespace, possibly extended into various derived namespaces throughout the document, but all sharing a common root or base which is set per message. On the other hand, attribute names are never qualified.

[18] The software preserves the ability to generate an XML document with no namespace at all.

[3] The low part of Row 0 of the UNICODE BMP is the ASCII character set, i.e. all 7-bit character codes.

3. Operations

3.1 Transformer Overview

The Parser is only one component within the reverseXSL Transformer (see the figure on page 2 and below). The Transformer handles a message in three steps:

1. A first step, using pattern recognition, matches a transformation profile to an arbitrary input message. A transformation profile contains zero, one, or both of the next two steps, with additional optional message handling parameters.
2. The Parsing step is triggered according to the selected profile. It decodes any structured character-data message and produces an XML document.
3. The XSL Transformation step is triggered according to the selected profile. It converts XML documents into other XML documents, else flat file structures.

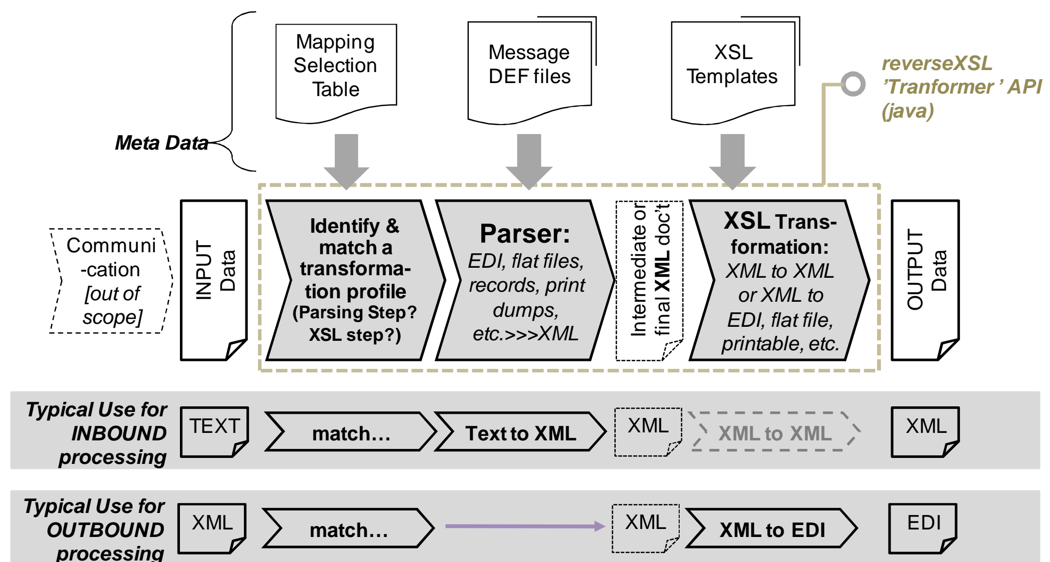

Figure 2. The flow of data

Although all internal software operations and the parsing step in particular are based on characters, the Transformer actually takes a byte stream as input and produces another byte stream as output. Input and output character sets that govern the conversion from bytes to chars and vice-versa are specified at the level of the TransformerFactory.

As far as the Parser is concerned, every API function handles only characters.

Parser operations

The present document focuses exclusively on the Parsing step.

www.reverseXSL.com

The Parser can be used as a standalone component, although it is most useful when combined to XSLT within the Transformer.

In the present version, the Parser handles the complete message in memory. This may pose a problem with very-very large messages, although not as much critical today with recent system as it could have been in the past when memory was still scarce and expensive.

For software optimization reasons, the maximum message depth is a software build parameter. It is presently set to 20. The most complex formats handled so far reached a depth of 12. The parameter can be increased if ever needed.

The implementation which was highly recursive in the beginning has been moved for a part to a state machine model in order to consume much less heap space. The consumption is now proportional to the maximum element depth in the message. Regular expression processing is another potential source of memory consumption; poorly formulated expressions (with for instance a repeated pattern that can match zero length literals) can actually eat much heap space.

The parser always executes the following 6 activities in sequence.

1. Load the given message definition: the so-called DEF file whose syntax is specified further in the present document.
2. Segment the message and decode the syntax. As the parsing itself proceeds, the extracted data is mapped into an internal hierarchical collection of elements. Looping structures can be promoted (i.e. creating additional nesting levels) or demoted (flattened), individual elements are added or removed, everything can be tagged and untagged at will, but the order of data element values from the original message is strictly preserved.
3. Once the parsing is complete, verify occurrence-loop constraints (min/max thresholds)
4. Next, verify all inter-dependencies (described as ’named conditions’ further)
5. Render, on demand, the internal tree of data elements into an XML document, or dump it in debug traces.
6. Report exceptions, on demand, and as applicable

www.reverseXSL.com

4. DEF File Structure

4.1 General Layout

A DEF file has the following general layout:

```
SET BASENAMESPACE "namespace" [SIGN.ATUR.E999.999]     optional

COND  …    specification of inter-dependency rules (conditions)
COND  …
      … more COND lines

 Comments may be written elsewhere; by convention, these are lines starting
   with a ' '(space) or tab character.

MSG    The top level segment is the message itself and is level 0 by definition
       The top level segment is noted MSG instead of SEG, although it is a
       SEGment
|SEG   first segment at level 1 with a single data piece
||D              data and other segments here are at level 2, hence prefixed ||
|SEG   second segment with 4 data pieces
||D        first data element
||D        second data element
||SEG      the third data element is a "sub-segment", i.e. a composite
|||D             first sub-element
|||D             second sub-element
||D        fourth data element, back to level 2
|SEG   third segment within the top-level segment, i.e. at level 1
||D
||D
|MARK    evaluates a condition on the fly and marks it at this point in the mes-
         sage
|GRP     first group at level 1, a group is only a logical grouping of segments or
         other groups that are then—by principle—at the next level
||SEG      a segment at level 2, first within the group
|||D
|||D
||GRP      a sub-group at level 2 within the level 1 group
|||D       Data elements may be placed directly in groups as well
|||SEG           a segment at level 3 within the previous level 2 group
||||SEG                a sub-segment, then at level 4
|||| etc…        the rest of level 4 shall contain data and sub-segments, and
                 sub-segments may again contain segments and groups!
|||SEG
||SEG      back to level 2, hence next to the previous group
||GRP      …more groups, etc…
END    marks the end of the message definition
```

The '|' characters are actually part of the syntax of the DEF file itself, and do control the nesting of definitions.

There are only six concepts to understand: SEGments, GRouPs, Data, MARKs and CONDitions. That is all the magic!

The DEF file contains an optional SET BASENAMESPACE statement, followed by a 'Named Conditions' block, followed by the MSG—END block that describes the hierarchical message structure. Conditions are also optional but there's seldom any real-world message definition without some.

4.2 What is a Segment (noted SEG)?

A segment at depth 3 is always followed by one or more sub-segments, data elements or sub-groups at depth 4, thus defining its child elements

Example:

|||	SEG	"^4:"	Text	M 1 1 ACC 1	R	F	"swTB Text Block"	CUT-ON-NL
segment depth is 3		identification pattern (starting with "4" followed by a semi colon)	XML tag is <Text>	Mandatory 1 min (specs) 1 max (specs) ACCept 1 in practice	Record exceptions	Fatal impact of errors	reference and description	The CUT pattern is here a simple built-in function that cuts on new line boundaries

A segment is the definition for a **string of characters** from the original input message **that gets cut into smaller strings**. In other words, a segment is used to cut sub-strings out of the syntax of the segment-string itself.

A segment is … segmenting input…

A segment always corresponds to a physical portion of the input message and corresponds altogether to the enclosed data elements and the associated framing syntax (tags, separators, terminators).

The top-level segment—by convention at level 0—is always the whole message itself.

A segment contains by definition:

Both the match with the identification pattern AND the cutting AND sub-element matching (of at least one piece) shall be OK for a segment to be considered "found".

- An *identification pattern* that is used to identify the segment;
- A *cutting pattern* or a built-in function to split the segment into smaller strings. The cutting logic is driven by capturing groups in regular expressions; see explanations further.

A segment may be associated to an XML tag (or NOTAG or RAW or SKIP)

A segment may be:

- **M**andatory, in which case the failure to match the *identification pattern* throws/records an exception
- **O**ptional, in which case the failure to match the *identification pattern* is interpreted as the absence of the segment
- **C**onditional, in which case the constraint is similar to the Optional case, but the absence or presence is now reported to a *named condition*.

A segment may contain data elements, groups, and sub-segments.

A segment may bear **min**imum, **max**imum and **acc**ept *loop counts* that drive looping logic in addition of being special kinds of conditions. They are combined with the Mandatory / Optional and Conditional (M/O/C) keywords. Further explanations will be found below.

A Mandatory or Optional Segment contains a *description* that will be attached to the exception thrown/recorded in case of violation of the mandatory or optional constraints, and with invalid loop counts. The description that is associated to Conditional rule violations is defined in the named condition statement.

A segment also bears an indicator whether to actually Record or to Throw any exception.

A segment may optional specify a *namespace suffix*, that is combined to the base namespace to form a namespace URI, which applies to this segment element and becomes the default namespace URI for all children elements.

4.3 What is a Group (noted GRP)?

A group at depth 2 is always followed by one or more segments, data elements or sub-groups at depth 3, thus defining its members (to become child nodes in XML)

Example:

||	GRP	"^:50[CL]"	IParty	O 0 1 ACC 2	R	W	"sw50CL Instr.Party"	/Parties
group depth is 2		identification pattern (starting by a semi colon followed by "50C" or "50L")	XML tag is <IParty>	Optional 0 min (specs) 1 max (specs) ACCept 2 in practice	Record exceptions	Warning in case of errors	reference and description	a suffix to the base namespace URI for this element and child ones

A group is pure virtual structure used to bind an arbitrary collection of segments, data elements, and (sub-)groups into a kind of association.

It is only a structural concept. It is used to associate conditions or tags to a group of segments (and data elements and sub-groups) instead of isolated segments, and to specify loop boundaries applicable to the relevant group of segments (and data elements and sub-groups).

A group may contain data elements, segments and sub-groups in any quantity, in any order.

A group has **no syntax element** of the input message **associated to the group structure itself**. It does exist only indirectly from the collection of underlying segments. A group can contain a single segment (with no tag) if so desired to associate some syntax framing to the group itself.

The use of a group has a fivefold effect:

- It is always a depth-level break; precisely, all direct group members are by definition down one nesting level.
- If any of the group members exist, the group itself will exist and may introduce a corresponding 'group' tag (an XML element name of type complex: sequence) into the output XML document
- A group can repeat and therefore define looping constructs than span over more than one segment.
- The group associates a Mandatory / Optional or Conditional constraint to its members as a whole, with possible min/max loop counts
- A specific *description* can be associated to a group.

A group may bear an *identification pattern*. This is never a requirement, but just a facility to immediately enter or skip a group structure (according to the match/non-match outcome) such as to speed up parsing. The use of a group identification pattern can become the origin of an exception, and thus the associated description may differ from the case the identification

pattern would not have been used at the group level but indirectly at the nested-segment or nested-data element level.

A group has Min, Max and Accept loop counts alike a segment, and a Mandatory, or an Optional, or a Conditional marker.

Like a segment, a group may optional specify a *namespace suffix*, that is combined to the base namespace to form a namespace URI, which applies to this group element and becomes the default namespace URI for all children elements.

Note: the Message itself as a whole is not a top-level group, but a top-level segment.

4.4 What is a Data Element (noted D)?

Example:

||||	D	"^:50C:(.*)"	BEI	M 0 1 ACC 1	T	F	"sw50C BEI code"	ALPHANUM [1..11]
data element at depth 4		matching and extraction pattern e.g. "987A" will be extracted out of ":50C:987A"	XML tag is <BEI>	Mandatory 0 min (indicates optional in specs!) 1 max (specs) ACCept 1 in practice	Throw exceptions	Fatal impact of errors	reference and description	validation pattern, here a built-in function Alphanumericals expected from 1 to 11 chars

A data element is a special instance of a segment that contains only one sub-string. In other words, the cutting process bound to segments stops with data elements. Data elements can only have the effect or removing syntax characters from the original string. **A data element yields always one single piece of data that will always fill one text node or one attribute node of one simple XML element** (i.e. with no children in case of a text node).

Unlike a segment, a data element is not associated to an *identification pattern* plus a *cutting pattern*, but bears instead a *validation pattern*. The validation pattern is also used as data value extraction pattern (the associated regular expression must contain **at least one** 'capturing group'; see explanations further).

Alike a segment, a data element will bear:

- An XML tag for the target XML element to fill up (NOTAG, RAW and SKIP are reserved names described further). If the tag name starts with '@', this element will become an attribute of the parent element
- Mandatory / Optional and Conditional (M/O/C) constraints with adjusted effects as follows:
  - **M**andatory: the element value cannot be empty. The failure to match the *validation pattern* always throws/records an exception.

Unlike segments and groups, a data element definition cannot specify a new suffix to the base namespace. It will automatically inherit that of the parent segment or group.

- **O**ptional: the whole element can be empty, or the element value alone can be empty[4]. If there are any characters, the failure to match the *validation pattern* throws/records an exception.
- **C**onditional: the constraint is similar to the Optional case, but the absence or presence of the element value is in addition reported to a *named condition*.

- An element may also bear **min**imum, **max**imum and **acc**ept *loop counts* in combination with the Mandatory / Optional and Conditional (M/O/C) keywords. Further explanations will be found below.
- A Mandatory or Optional Data element also contains a *description*.

4.5 What is a Named Condition (noted COND)?

Example:

COND	"[FAL]"	DEPTH 1	R	W	" Application ID! must be one of F A or L"
	matching pattern to verify the condition (condition feeds must yield one of "F", "A" or "L")	verification is performed at depth 1, i.e. for each individual message (possibly repeating). Depth 0 is the whole file	Record exceptions	Warning in case of errors	reference with error message when verification of the condition fails

Conditions are by convention listed at the top of the DEF file, above the message definition itself and next to the optional SET BASENAMESPACE statement.

Conditions express interdependencies between data elements, segments, groups or anything else.

Conditions can either be:

- *Global*, in which case the condition shall be valid once for the whole message, whatever the levels and repetition counts from which the dependent elements may come
- *Local*, in which case the condition shall be met for every instance of a specified depth level and the enclosed sub-levels.
  For instance, a group repeats at depth 2 and contains segments and data elements affected by inter-dependencies that respectively occur at depth 3, 4 and 5. A Local condition depth 2 will test for the inter-dependency condition to apply within every repetition of the group at depth 2.
  It does mean that a Global condition is a Local condition at depth 0.

[4] Note that a non-null element may happen to only contain syntax characters, i.e. separators and other stuffing or padding characters.

www.reverseXSL.com

Conditions are denoted 'named conditions' because the condition name is used to link all the inter-dependent elements.

Named conditions are **verified only after the complete parsing of the message**. The data for verifying a condition is collected while parsing the input message, but the verification itself and any throwing/recording of exception occurs only at the end.

Named Conditions are listed at the beginning of the DEF file because the loading of such conditions do trigger the internal creation of objects that record the condition feeds during the entire parsing. Ultimately all collected condition feeds are organized per nesting level and loop boundary, and matched against the condition itself.

The definition of a named condition contains:

- An indication of the *depth* level at which the condition must be verified. (see above)
- A *matching pattern* that shall be verified for the condition to hold true, see the mechanism below.
- An associated *description* of the exception to throw/record in case the verification fails.

How inter-dependency conditions are verified?

Conditions are verified using pattern-matching logic. The data feeding this matching process is constituted from string elements—called **tokens**—collected during the parsing of the message. The tokens can be arbitrary string constants, else (portions of) data element values. Each time a group, segment or data element is declared as Conditional, a specified string (another pattern which yields a constant or specifies a capturing group) is added to the named condition collection of tokens. The loop counts for every nesting level are added as attributes, as well as original element references and offsets in the message. When the parsing of the message is completed, each condition is then evaluated. The depth/scope level instructs how to proceed with the grouping of collected tokens (actually pieces of text string) and proceed with pattern matching against the concatenated string result. Any depth/scope above 0 do cause the verification process to loop at that one level.

The careful selection of string constants combined to the flexibility of patterns allows matching about every inter-dependency constraint.

A straight inventory of all expected keyword combinations separated by '|' (OR logic) in a regular expression would do the job in most cases.

Important note: **when no tokens at all were collected under a given name during the parsing, the associated named condition is not checked**. In fact, a named condition may often express interdependencies in between a few data elements, and all of them may be contained in an optional structure. Checking that the relevant condition would accept, in this case, the empty string has nothing to do with the interdependency itself but with the existence or absence of the whole containing structure. The latter is duly verified using min/max occurrence specifications in place of conditions.

www.reverseXSL.com

4.6 What is a Mark (noted MARK)?

Example:

||	MARK	@Priority	COND INpriority	"S"	"SYSTEM"	"NULL"
mark element at depth 2		yields an XML attribute Priority="…" attached to parent element (SEG or GRP)	Current feeds into the named condition "INpriority" are used for evaluation	evaluation pattern. Testing here for a match with the single letter "S"	mark element value if TRUE.	mark element value if FALSE. "NULL" is reserved value for *do-not-generate-the-element-at-all* whereas "" would yield an empty element.

A mark is the evaluation, on the fly, of a named condition, whose result is inserted in the output message at the depth and at the level where it is evaluated.

Alike for a data element, if the tag name of a mark begins with @, it will be promoted as an attribute to the parent element.

A mark behaves mostly like a data element definition but for the essential fact that it does not consume (and does not need to match) any input data.

A mark allows inserting a value in the message flow to 'mark'—literally—the verification or non-verification of a given pattern of occurrences in the message. A mark allows reporting explicitly in the output message (with XML tags or attributes and values) the result of evaluating a named condition. There are four possible purposes:

- One can attached a named condition to a data element in order to generate its value as token associated to such condition. Then, just next to the data element and at the same depth, one or several marks may be inserted in the message, each containing an evaluation expression that recognise the different patterns of the said token as, for instance, an email address, a fax number, a contact person or else, and consequently insert in the output message an explicit XML element (or attribute) whose value would 'mark' the type of data element just recognised.
- A variant of the above case is to pick-up the entire value of a coded data element in a named condition, and then evaluate one by one all possible code values in subsequent Marks, with the effect that the one Mark matching the code will be inserted in the output XML and can actually yield the verbose equivalent (or alternative coding) of the coded value. In other words, this mechanism allows performing code mappings on the fly at parsing time.
- One can explicitly report that 'no such element' or 'no such group' or 'no such segment' was found in the source message.
- One may decide to report within the XML output itself that some interdependencies or other conditions (alike min and max occurrences, or the presence or absence of specific data) were met in the message (for which a named condition is associated) and thus indirectly give processing instructions to whoever will handle the XML output.

Evaluating a mark is almost the same process as verifying a named condition but there are noteworthy differences:

- The evaluation is performed on the fly and not once the parsing is completed like named conditions. The evaluation is thus performed with whatever named condition tokens are already available **at the point** where it is evaluated, and **at the depth** that is that of the MARK element itself.
- A mark never throws nor records an exception: it is evaluated and the result of such evaluation could only be true or false. A corresponding **value** is inserted matching the true or false result, and that is all.
- The reserved **value** "NULL" may be used to suppress the production of an output XML element in case of the True or of the False outcome (both of them at the same time would not make sense but work anyhow, and yield a no-operation).

4.7 What is the Effect of min/max & Accept Loop Counts?

MINimum, MAXimum and ACCept loop counts apply to groups, segments and data elements.

ACC is a parsing-in-a-loop control tool, not a condition

Any ACCept count above 1 does trigger a loop attempt while parsing the message. The actual ACC count does limit the number of attempts to identify the relevant structure (the parsing loop is executed till a first failure to identify the structure occurs, else the ACC count is reached).

An ACC count above the MAX count does allow accepting more instances than necessary (in order to be able to continue parsing for instance) but will report an exception later on (recorded or thrown).

MIN / MAX / ACC specifications

MIN, MAX and ACC boundaries are specified as numbers, and only numbers.

A very large MAX or ACCept number shall be used for *nearly* unlimited counts. We mean 'nearly' unlimited because there's no convention like using a 'R' letter for instance for indicating a truly unlimited Repetition. One shall always enforce a value because machine representations of integers are also limited[5]. In case of doubt, a good max value is made of nine 9's (it's easy to remember), i.e. 999999999 or a billion instances (fitting a 32bit unsigned integer). In all circumstances, using such large number means that the message shall also be at least 1 gigabyte long at this position (if repeating only one single char). We do recommend calculating fair maxi-

[5] If a max number is too large, it will fail while loading the DEF file and not while parsing a message.

mums based on the expected maximum message size divided by the minimal item length in bytes.

The top level segment or message itself is always Mandatory, 1-min , 1-max, 1-acc.

Exception handling

Alike named conditions, Min/Max violations are checked and reported after the parsing of the input message is completed. Such verification has precedence over the verification of named conditions.

The reason for the delayed reporting of any min/max exception (in case they were all recorded of course and the maximum acceptable record count has not been reached) is to provide a clear grouped report about all repeating/looping problems at a single place.

Let's be clear: the verification of min/max constraints is **not** performed during the parsing. Only the M/O/C cardinality indicator and the ACC counts are used to control looping constructs while parsing.

So, the verification of M/O/C cardinality (Mandatory or Optional, and Conditional alike) must be distinguished from min/max constraints. M/O/C cardinality is, it, verified in the course of the parsing. The reason is indeed bound to the direct impact of Mandatory / Optional (or Conditional) constraints onto the correct parsing of the rest of the message.

Each time the parser enters a new group or segment or data element definition line (while progressing top down through the message definition and while looping during parsing), a min/max condition object is added to the global condition context. Repetition counts are recorded along with details on the group, segment or data element name, offset in the message, etc.

The parser exists from a group or segment or data element definition line (and associated min/max condition object) when either the ACCeptable count has been reached, either the *identification pattern* (or validation pattern for a data element) failed to match.

If there is no match and the Mandatory constraint is set, then an exception is raised immediately (which can be recorded or thrown). Recording a Mandatory violation is then a means to tolerate the absence of an element.

At the end of the parsing, the collection of all group, segment and data element instances is available for global tracing/debugging purposes in addition to the verification of min/max conditions as described below.

About the Interpretation and actual effects of loop counts

Min/Max/Acc counts are associated to Mandatory, Optional and Conditional constraints in somewhat subtle ways.

Let's illustrate through examples.

www.reverseXSL.com

Notation: the first letter stands for Mandatory, Optional and Conditional; the two numbers that follow respectively indicate the min and max loop counts. The whole is followed by the ACC keyword and the acceptable loop count value. ***Everything stated here for Optional applies to Conditional as well.***

Simple cases:

M 1 999999 ACC 999999	Means the unlimited repetition of at least one segment, or group, or data element
O 0 99999 ACC 999999	Means zero or more instances of the segment, or group, or data element
M 1 1 ACC 1	Means one and only one instance
O 0 1 ACC 1	Means zero or maximum one

Advanced cases:

M **3 10** ACC 999999	From 3 to 10 instances shall be found in theory but the parsing may continue if only 1 up to more than 10 instances are actually found in the message. An exception (e.g. thrown as fatal or else recorded as warning, as further specified) is raised for any count below 3 or over 10.
O 0 10 ACC **999999**	From 0 up to a nearly unlimited count of instances is accepted at parsing time. But any count over 10 will raise an exception when parsing is finished. This is recommended whenever the increase of ACC versus the formal Maximum will not cause the parsing to slip into misinterpretations. Indeed, many receiving applications may be able to accept more line items or other elements than formally allowed in the message standards.
O **1** 10 ACC 10	During parsing, 0 to 10 instances will be accepted. Next to parsing, anything below 1 will raise an exception. This is a way to relax Mandatory constraints whenever the parsing will not be affected. Missing elements will typically be reported as warnings anyhow.
M 1 10 ACC **1**	The parser is forced to accept only one instance. No more. This may be a way to temporarily restrict the message to a subset because the later mapping step is for instance unable to handle more than 1 instance. However, the official definition specifies up to 10 occurrences. If the input message actually contains 2 or more instances, the parsing will likely fail to match the repeated segments that follow.
SEG…A… M 1 10 ACC **1** SEG…B… O 0 10 ACC **9**	This is a variant to the above case, where different segment structures are used to absorb the first 1 and the next 9 instances. The MAX value (here 10) becomes useless but as a piece of documentation!
M 1 10 ACC **0** or O 0 10 ACC **0**	ACC O is a special case: the parser is forced to skip this group, segment or data element. Useful for testing or temporarily enforcing a subset.
O 0 **0** ACC 999	The parser accepts the relevant data or segment, but will report an error whenever a single instance is present in the message (MAX=0!). This is useful for instance to absorb data lines next to the formal end of a message instance and report this as a warning next to parsing.

4.8 How is Data actually Separated from the Message's Syntax?

The syntax of the message is by definition made of:

- Tags, not necessarily prefixing a segment but appearing anywhere else in accordance with relevant message syntax definitions;
- Segment terminators and separators;
- Data element separators (explicit delimiters);
- Implicit delimiters as implied by fixed size structure definitions or patterns (e.g. num to alpha transition);
- Patterns that differentiate an element from another one that could occur at that same place. For instance, a variable-size field that is longer or shorter than a given size can lead to different interpretations (e.g. in Cargo-IMP REF/… lines); another example is that of a data element starting with alphabetical characters instead of numerical ones, and that shall be interpreted differently.

Which can be nested to any level.

Such syntax shall be 'removed' from the raw input message in order to yield clean, trimmed, data element values. Such trimming can occur:

- While cutting a segment into constituent elements;
- While validating individual data elements.

There are then two points at which syntax can be thrown away. The balance between these two places is a matter of taste, development convention, or ease:

> Although equivalent in term of the end result (getting the output XML document filled up with the data from the original message), the point at which syntax is removed will potentially affect the error outputs: more or less of the original syntax will still be visible in the error messages while exceptions are generated from the depth of the very last data element validation steps. In addition, miss-placed separators or bad padding (field alignment) for instance will generate a 'cut' error while processed at a segment level, else a 'validation' error while processed at the data element level.

Segment cutting modes based on regular expressions

Regular expressions provide mechanisms for three cutting modes:

7. Repeating Pattern cutting mode

   In such mode, only capturing group 0 (the pattern itself) is used and the pattern itself would be selected such as to repeat from the beginning to the end of the segment. The successive matching elements of the original segment do define as many sub-segments/data elements.

   A / followed by zero or more not-/ chars

   For instance the pattern is: "`/[^/]*`"

   And the segment is: "`DIM/12345//ABCDEF/XYZ ABC`"

   It yields the following segment pieces: "`/12345`", "`/`", "`/ABCDEF`", and "`/XYZ ABC`".

   The "DIM" chunk is left out of the inventory, which is OK if DIM is the segment tag. However, in some cases, the first element is also a data element. The following variant allows capturing it:

A / followed by zero or more not-a-/ chars, or, the start of input followed by zero or more not-a-/ chars

The variant pattern is: "`/[^/]*|^[^/]*`"

And the segment is: "`LUX/12345//ABCDEF/XYZ ABC`"

It yields the following pieces: "`LUX`", "`/12345`", "`/`", "`/ABCDEF`", and "`/XYZ ABC`".

We shall observe that in both cases the syntax character "/" is still part of the generated data elements.

8. Straight Capturing-Group cutting mode

   The pattern must contain capturing groups (i.e. there's at least one pair of "()" in the pattern string) and capturing groups 1 to n (indeed not group 0 which is the pattern itself) and respectively yielding the successive matching elements from the original segment.

   Nested capturing groups are ignored.

…using reluctant mode for the ".", that means any char repeated.*

   The pattern is: "`^DIM/(.*?)/(.*?)/((.).*?)/(.*)$`"

   And the segment is: "`DIM/12345//ABCDEF/XYZ ABC`"

   It yields the following matching groups and thus four segment pieces: G1:"12345", G2:"", G3:"ABCDEF", G4 is ignored (nested), G5:"XYZ ABC".

   We shall observe that the syntax characters "DIM" and then "/" are **not** part of the generated data elements.

9. Repeating Capturing-Group cutting mode

   This mode combines the above two. Group 0 (the pattern match itself) is excluded from the inventory of data elements in proper but only representing the pattern-matching loop. Data elements are generated from group 1 to n within the group 0 loop. Nested capturing groups are ignored.

A series of three chars followed by a dash, then any number of not-a-/ char, followed by an optional /.

   For instance the pattern is: "(...)-([^/]*)/?"

   And the segment is: "123-4ABCDEF/456-XYZABC000/789-BBBCCC"

   We get a total of six data elements:

   in pattern-loop 1: G1:"123", G2:"4ABCDEF"

   in pattern-loop 2: G1:" 456", G2:" XYZABC000"

   in pattern-loop 3: G1:" 789", G2:" BBBCCC"

Again, the syntax characters are no longer part of the generated set of data elements.

www.reverseXSL.com

***Separator or Terminator**? cut-delimiters are handled as <u>separators</u> within any segment but the top-level segment (i.e. message) itself, where the semantic is that of a <u>terminator</u>!*

Segment cutting modes based on simple functions

The cutting patterns provide a very rich and flexible mean of cutting off a segment. This is more sophisticated than required for the great majority of cases. A set of built-in functions is available that greatly simplifies parsing.

For instance, the cutting function CUT-ON-(/):

Applied to: "`LUX/12345//ABCDEF/XYZ ABC`"

Yields immediately: "`LUX`", "`12345`", "", "`ABCDEF`", and "`XYZ ABC`".

The cutting function CUT-ON-NL (New Line) will typically be used to cut the top-level segment, the message itself, into sub-segments and groups. Each segment is indeed a text line.

The built-in segment cutting functions **do** remove the syntax characters from the generated data element list. Somehow, they provide the simplicity of a repeating pattern with an easy way of getting rid of syntax characters.

More functions of the kind are specified further in §5.1

The use of built-in functions is highly recommended and patterns shall be kept for dealing with the complex cases, as well as validating data elements (cf. just next).

Data Element cutting (and validation) modes

A piece of string that will be matched by the parser as corresponding to a data element may or may not still contain syntax characters. In the former case, such syntax shall be removed; in the second case, one can validate the data contents immediately.

The validation pattern must contain at least one capturing group—at least a pair of '(' ')'—that isolates the data portion from the syntax and padding stuff.

In case multiple validation groups are contained the **data value is** defined as **the concatenation of all capturing groups** 1 to n in that order. Such technique can be used to remove syntax characters in the middle of the data element. Note that during concatenation, **nested** capturing groups are ignored; only the capturing groups following each other are concatenated (otherwise the data matching sub-capturing-groups will be duplicated in the result!).

In all circumstances **the validation pattern must match the entire input string** making the data element; i.e. the matcher.matches() method of Java would yield TRUE. Yet in other words, the interpretation of the pattern is enforced as <u>'possessive'</u>.

The validation pattern has a twofold purpose:

- As a means to remove syntax characters and padding;
- As a data value validation function, i.e. as performed by sub-pattern contained within the capturing group(s).

Note that this second role is only one of the two ways to validate the data element value:

www.reverseXSL.com

1. The first method as described above is to use the validation-pattern itself and enforce the necessary restrictions inside the capturing group(s).
2. The second method is to use a generic capturing group specification—alike '(.*)'—and rely on a built-in character-set validation function alike UPALPHA, NUMERIC, DIGIT and others defined in §5.2.

4.9 Delimiter or Data? Using a Release Char

Various legacy EDI standards use a variable-field-size syntax. Printable characters like + : ' * ~ # / - are often used as field separators, and segment or block terminators. But these are also commonplace characters that may occasionally appear in data values themselves.

In the American X12 standard, the characters used as delimiters are simply forbidden in data values. One has to choose delimiter characters so that they never occur in data and specify those delimiters in the ISA Interchange Header segment.

An alternative choice can be explicitly specified in the UNA service segment that will prefix the whole interchange.

In the ISO EDIFACT standard, the characters used as delimiters may also occur in data, provided that they are preceded by a so-called 'release character' thus forming an escape sequence. The default release character is '?'.

Assuming that +, :, and ' are delimiters, and ? the release char; a data value like:

```
JOHN'S SELECTION: BLUE? NO, GREEN+YELLOW
```

Shall be rewritten in EDIFACT:

```
JOHN?'S SELECTION?: BLUE?? NO, GREEN?+YELLOW
```

The Parser features a built-in mechanism for the correct handling of delimited data fields possibly containing delimiter characters in literal values. The release character is simply declared at the very beginning of the DEFinition file with a statement like:

```
SET RELEASECHARACTER '?'
```
for a single char

```
SET RELEASECHARACTER '\u2023'
```
for a Unicode value

or else

```
SET RELEASECHARACTER '\\'
```
for the \ char itself

Then, whenever a segment cut function that specifies delimiters (explicitly or with a regex) is invoked and matches an area of a single character, the Parser checks for an odd count of release characters immediately preceding it, and if so, skips this cut. Ultimately, when extracting data values (as well as while feeding CONDitions), the values are post-processed such as to remove release characters and render the plain original value in XML.

www.reverseXSL.com

4.10 Using Reserved XML tags: NOTAG, RAW and SKIP

There are three reserved tag values with diverse effects on the XML output. None of them can apply to the message level itself (i.e. top level segment).

NOTAG

Applies to segments, groups and data elements. NOTAG is a reserved name that asks to **flatten** the XML structure, or in other words, suppress the relevant element nesting.

For instance, the parsed result in native parser format is:

```
ROOT
    segment SEGMENTONE
        data DATA1 "hello"
        data DATA2 "world"
        group NOTAG
            data DATA3 "foo"
            data DATA4 "bar"
        data DATA5 "and"
    untagged "some"
    group GROUPONE
        data DATA7 "more"
END
```

will yield the XML

```
<ROOT>
    <SEGMENTONE>
        <DATA1>hello</DATA1>
        <DATA2>world</DATA2>
        <DATA3>foo</DATA3>
        <DATA4>bar</DATA4>
        <DATA5>and</DATA5>
    </SEGMENTONE>
    <RAW>some</RAW>
    <GROUPONE>
        <DATA7>more</DATA7>
    </GROUPONE>
</ROOT>
```

SKIP

Applies to segments, groups and data elements. "SKIP" asks to entirely suppress the relevant element from the output XML structure.

For instance, the parsed result in native parser format is:

```
ROOT
    segment SEGMENTONE
        data DATA1 "hello"
        data DATA2 "world"
        group NOTAG
            data DATA3 "foo"
```

www.reverseXSL.com

```
            data SKIP "bar"
        data DATA5 "and"
    untagged "some"
    group SKIP
        data DATA7 "more"
END
```

will yield the XML

```
<ROOT>
    <SEGMENTONE>
        <DATA1>hello</DATA1>
        <DATA2>world</DATA2>
        <DATA3>foo</DATA3>
        <DATA5>and</DATA5>
    </SEGMENTONE>
    <RAW>some</RAW>
</ROOT>
```

RAW

Applies to segments, groups and data elements.

If the parser method generating XML output is invoked with the boolean argument "withRAW" as **false**, both the structures tagged "RAW" and the rest of yet untagged structures (as may be left in the message consequent to failed parsing whose exceptions were recorded—hence trapped—and not thrown) are removed from the XML output.

For instance, the parsed result in native parser format is:

```
ROOT
    segment SEGMENTONE
        data DATA1 "hello"
        data RAW "world"
        group NOTAG
            data DATA3 "foo"
            data SKIP "bar"
        data DATA5 "and"
    untagged "some"
    group SKIP
        data DATA7 "more"
END
```

will yield the XML

withRAW==**false**	withRAW==**true**
<ROOT> <SEGMENTONE> <DATA1>hello</DATA1> <DATA3>foo</DATA3> <DATA5>and</DATA5> </SEGMENTONE> </ROOT>	<ROOT> <SEGMENTONE> <DATA1>hello</DATA1> <RAW>world</RAW> <DATA3>foo</DATA3> <DATA5>and</DATA5> </SEGMENTONE> <RAW>some</RAW> </ROOT>

www.reverseXSL.com

4.11 Using the Reserved NULL Value

The NULL value is only applicable to MARKs (cf 4.6 & 5.6). Note that it shall be written as the quoted string "NULL" as illustrated below.

For instance:

```
|||MARK AIRTAG COND CodeCheck "[0-9]{3}" "airline code" "NULL"
```

yields the XML element:

```
<AIRTAG>airline code</AIRTAG>
```

if the condition 'CodeCheck' evaluates to 3 digits exactly, and would not generate any XML output otherwise.

4.12 Exception Handling

Exceptions:

- are **Thrown** or **Recorded**: in the former case the module calling the parser shall trap the exception—the exception is effectively thrown as defined in java; in the later case the exception is recorded in an ordered list of exception objects and processing continues.
- have an error level as **Warning** or **Fatal**. The distinction has no particular semantics and no effect beyond the counting of:
    - number of fatal exceptions;
    - total number of exceptions, warning and fatal level.

  These two counters are compared with respective thresholds after which all exceptions are thrown—in java terms.

Types of exceptions

The following brands of exceptions can be raised while parsing the input file and building the target XML message:

A. Syntax errors while parsing

Can be recorded or thrown, can be fatal or warning

A.1 Violation of the message structure; i.e. sequence of segments and nesting, notably when:

- A mandatory segment or data element is missing
- No member of a mandatory group has been found
- Data is found in the source message for which no matching definition exists

A.2 Unable to break a segment into constituent elements; problems occurred related to cutting due to invalid regular expression syntax.

A.3 Unable to extract a data element value from the input syntax; typically, the validation pattern failed to match the whole input data element, or the regular expression syntax is invalid.

A.4 Invalid characters or size of the extracted data value.

www.reverseXSL.com

B. Conditions not being met

Can be recorded or thrown, can be fatal or warning

B.1 Found less than MIN or more than MAX instances of a repeating group, segment or data element

B.2 A Named condition fails to match the associated inter-dependency pattern

C. Built-in errors

Never recorded, always fatal

C.1 DEF file is corrupt: invalid GRP, SEGment, MARK or Data definitions are contained, and so forth.

C.2 Additional data is found after the end of the message; note that this is not taken as an error in category A because one can always accept extra data after the official end of a message by inserting a last segment of the kind (that also tells how to stuff it in the target XML message):

```
|SEG "" "GARBAGE" O 0 0 ACC 999 R W "found extra data"
```

This variant using a group may be preferred: it has the advantage to raise just a single warning-level exception for any number of lines found after the official message end. The target XML will also contain just one "Extra" element.

```
|GRP "" "Extra" O 0 0 ACC 1 R W "found garbage"
||SEG "" "GarbageLine" O 0 999 ACC 999 R W "extra line"
```

C.3 Java system errors: array out of bound, NULL object, or else… will require bug fixing indeed…

Exception Attributes

Defined by Parser-Exception:

Whenever an exception is thrown or recorded, the following attributes are available for tracing:

- The nature of the exception, featuring a reference number, e.g. "P0023 MAX allowed count exceeded". The code may be used to find a translation into a language resource file (see further).
- The description of the element being affected by the error, available as a reference-code plus text. The code may be used to find a translation into another language in a language resource file (see further).
- The Warning or Fatal error-level (W or F)
- The absolute or line-relative offset (in characters) in the file
- The absolute line position in the original message (whatever line terminators are used: LF, CRLF, CR alone or FF).
- A copy of the complete segment or the chopped-out data element at stake (string)
- A copy of the complete previous or parent segment (string) (allowing to say "next to…" or "within …" in the error message if so desired for a following segment, or for a data element respectively)

The contextual cascade of all repetition loops with associated tags, e.g. FFR(1),ULD_Group(7),ULD_ID(3),WeightCode(1), appears to bear limited usefulness and so has not been implemented.

www.reverseXSL.com

Description text structure

A piece of description text as specified in SEG, GRP, D or COND lines must always be included in DEF files in plain English.

The text string has itself a structure alike:

```
"keyword free text following it"
```

WARNING: this feature is not yet available in the public software release of the reverseXSL parser

Where the keyword is the first word (i.e. not containing space characters) of the description text itself. That keyword must use only characters in the ASCII character set. The parser will use it to find possible text substitutes in other languages whenever a language-map object is passed along as argument while invoking the methods that handle exceptions.

If no translation is found for a given error text (hence no matching keyword exists in the passed language-map object) the parser will keep the original error text from the DEF file itself.

For instance:

Multi-language support is prepared but has not been implemented according to initial development scope. Will be enabled on request. ***This section is informative.***

We have the following definition:

```
|SEG "^/OSI" "OSI" O 0 1 ACC 99 R W "FFR-6. OSI line"
```

And we have also a French-language-map object notably containing

```
"FFR-5." "Ligne SSR – Requête de service spéciale"
"FFR-6." "Ligne OSI"
"FFR-6.2" "Première ligne détails OSI"
"FFR-6.3" "Deuxième ligne détails OSI"
```

And an exception "P022 MAX allowed count exceeded" is raised.

That will trigger an exception message in English alike:

```
"ERROR (P022) Repeating not allowed : FFR-6. OSI line
       [L:22,0:17] OSI/ORIGINAL EXTRA LINE"
```

And in French alike:

```
"ERREUR (P022) Répétition non autorisée : FFR-6. Ligne OSI
       [L:22,0:17] OSI/ORIGINAL EXTRA LINE"
```

And if the translation was asked but not found:

```
"ERROR (P022) Repeating not allowed : FFR-6. OSI line
       [L:22,0:17] OSI/ORIGINAL EXTRA LINE
       (no translation found)"
```

5. DEF File Line Formats

All SEG, GRP and D lines are preceded by at least one '|' character indicating the nesting level of the relevant segment, group or data element.

5.1 SEG (segment) Definitions

SEGment definition lines have the following syntax:

SEG	"*pattern*" ""	*aTag* NOTAG RAW SKIP	M O C	*min*	*max*	ACC *nb*	R T	W F	"*description*"	*cut-function* CUT "*pattern*"	/*suffix*
							COND *name* "*feed*"				

Formally:

```
SEG {"<id-pattern>"|""} {<XMLtag>|NOTAG|RAW|SKIP} ../..
    {M|O|C} <MIN> <MAX> ACC <ACC> ../..
    {{{R|T} {W|F} }|{COND <name> "<feed>"} } ../..
    "<description>" ../..
    { <cut-function> | CUT "<cut-pattern>" }
    { /<suffix> }0..1
```

Where:

The id-pattern can be found elsewhere in the input segment, according to pattern specs.

`<id-pattern>` is the segment identification pattern. The segment is matched only if:
`Pattern.compile(id-pattern).matcher(segment).find()`
yields TRUE and further cutting and parsing of the segment contents itself yields at least some matching sub-elements.

In other words, some pieces of the content of a segment must be successfully identified in order to consider that an instance of it has been "found" in the source message.

`<cut-pattern>` is a pattern that will be used to cut a segment into sub-strings as explained in §4.8 here-above.

Terminator** or **delimiter?
*CUT functions assume a delimiter semantic (hence "<sep>data<sep>" yields "" and "data" and "") when applied to **segments**, and a terminator semantic when applied to the **message** itself (the top-level segment indeed). For instance a CUT-ON-(') of the simple Message "'data'" yields two elements: "" and "data".*

In addition to the built-in CUT functions we have the generic CUT "<pattern>" described in §4.8

Do not mix CUT "<pattern>" with CUT-ON-"<regex>": the former uses capturing groups, the later defines the particular string to consider as separator (and remove from the syntax).

`<cut-function>` is a built-in function that cuts the segment into data elements or sub-segments. The Built-in functions are:

- CUT-ON-(<char>) *any single printable ASCII, for instance* CUT-ON-(/), CUT-ON-(,), *etc.*
- CUT-ON-NL: *The function cuts the given segment into as many sub-segments as lines in the input string. Any of CR, CRLF, LF or FF, are valid line terminators (CR=\u000D, LF=\u000A, FF=\u000C).*
- CUT-ON-TAB
- CUT-ON-1NBSP *a single non-breaking space is assumed; i.e. two following spaces define an intermediate empty field*
- CUT-ON-RNBSP *repeating NBSP can be used as separator; i.e. two following spaces, or 'a space plus a tab plus two spaces' are both interpreted as one separator.*
- CUT-FIXED-(<n>) *where <n> is an integer. This function cuts the segment in fixed size elements of n characters (not bytes, think Unicode).Example: CUT-FIXED-(3), CUT-FIXED-(15), CUT-FIXED-(1000)… Attention: the last sub-element may contain less than n characters.*
- CUT-ON-"<regex>" *where <regex> is the specification of the **separator**. This function allows specifying a set of different characters as separator, or a combination of several characters. Example:*
  - CUT-ON-"[./]" *cuts a segment on every single '.' or single '/'*
  - CUT-ON-"--" *will cut a segment on sequences of two hyphens '--'.*
  - CUT-ON-"[\t]+" *is equivalent to* CUT-ON-RNBSP

The Built-in functions **do remove** the relevant syntax characters.

Attention: these syntax characters are always assumed as being **separators**; i.e. "/REF/123//ABC/" contains not four but six fields separated by a '/':
"", "REF", "123", "", "ABC" and ""

However, the above delimiter-semantic is turned into a terminator-semantic in case of the message itself (the top level segment indeed). If the above example is the whole message, "ABC" becomes the last of a set of 5 pieces only.

<description>	is a string whose first word can be used a key to search a matching translation in a language-map file.
<XMLtag>	is the XML data element name to create; i.e. it will for instance yield <myXMLtag>…</myXMLtag>.
<name>	is the name of a named condition defined by a COND line (see further).
<feed>	is the piece of string to be fed into the relevant named condition collection. It is one of: • a plain text string without a single ’(’ , in which case that is the string to feed into the condition collection. • a string containing at least one ’(’ in which case it is interpreted as a pattern with capturing groups and the feed-string is the concatenation of all capturing groups of only the first pattern match (almost like a data element except that the pattern is not interpreted as possessive).
{R|T} {W|F}	stand for Record versus Throw, and Warning level versus Fatal error level. Please refer to §4.12.
/<suffix>	is an **optional** suffix to be appended to the default or explicit namespace URI as explained in §6. Example: "/Order/Body" (without quotes on DEF line, always starting with a "/", and no spaces).

5.2 D (data element) Definitions

D’ata element definition lines have the following format:

D	"*pattern*"	*aTag* RAW SKIP NOTAG	M O C	*min*	*max*	ACC *nb*	R W T F COND *name* "*feed*"	"*description*"	*char-spec* ASMATCHED	**[***min***..***max***]**

Formally:

```
D "<valid-pattern>" {<XMLtag>|RAW|SKIP|NOTAG} ../..
    {M|O|C} <MIN> <MAX> ACC <ACC> ../..
   { {{R|T} {W|F}"}|{COND <name> "<feed>"} } } ../..
   "<description>" { <char-spec> | ASMATCHED } { [<lmin>..<lmax>] }0..1
```

Where:

www.reverseXSL.com

Data validation is implicitly 'possessive', i.e. the specified validation pattern shall match entirely, and once, the whole input data piece. No characters can be left away from such matching. It's all or nothing. The beginning and end of the validation pattern must match the beginning and end of the data.

`<valid-pattern>` is the data element validation and cutting pattern. This pattern must contain **at least** one capturing group, the group that yields the data element value.

The data element is valid only if:

```
Pattern.compile(valid-pattern)
.matcher(dataElement)
.matches()
```

yields TRUE.

In addition, the data element value (next to extraction) must bear characters from the set specified as <char-spec>.

This pattern is responsible for extracting data.

Examples:

- "(.*)" to accept anything but later restricted by <char-spec>.
- "^/(.*)" to remove a leading '/'
- "^(.+?) *$" to trim trailing spaces

In case the pattern contains more that one capturing group, the extraction procedure works as described in 'Data Element cutting (and validation) modes' on page 21.

`<char-spec>` is a keyword that defines the set of characters that can be accepted in the data element **value** (i.e. after extraction). We have:

- ALPHA *is A-Z a-z (no space char)*
- UPALPHA *is A-Z (no space char)*
- ALPHANUM *is A-Z a-z and 0-9 (no space char)*
- UPALPHANUM *is A-Z 0-9 (no space char)*
- IATA *is UPALPHANUM plus '-' (hyphen), '.' (full stop) and space; it applies to Cargo-IMP and AHM standards and matches the 't' free-form-text specification.*
- DIGIT *is 0-9 (no space char accepted)*
- NUMERIC *is 0-9 '+' '-' ',' '.' and ' ' (space)*
- ASMATCHED *implies that no additional validation is made but as already performed inside the capturing group of the <valid-pattern> itself.*
- REPEATED-"<regex>" *allows using a regular expression on the extracted value (NOT the original input element). To be valid, the data element must match exactly* {<regex>}$_{0..n}$ *i.e., the value is a repetition of <regex> from 0 to any number of times.*
  For instance: REPEATED-"AB(C)?" *yields true for the value* "ABCABABABC"*, as well as* "".

Beware that special characters like '.' and '-' do not mean the same thing when written in between square brackets [ ] and outside of such square brackets in regular expressions.

And a validation against an explicit character set can be simply noted:
REPEATED-"[**A-Z0-9_/:.+=_-**]*"

- ASCII, *stands for all ASCII printable characters (with space), i.e. between U+0020 and U+007E inclusive.*
- DATE-"<dateFormat>" *uses the dateFormat pattern as argument to the SimpleDateFormat.parse() method.*

`[<lmin>..<lmax>]` is an optional specification for the minimum and maximum length of the data element **value** (as resulting from the extraction). This specification may be written for instance as:

[1..10] for a value containing 1 to 10 characters

[..10] for a value up to 10 characters

[2..] for a value containing at least 2 characters

The unit of length is the Unicode character, and not bytes.

Note that the "[]" characters are required: "[]" are terminal symbols and not just notation syntax; the fact that this specification is optional is noted { }0..1

Other fields are similar to the SEG definition.

5.3 GRP (group) Definitions

GRouP definition lines have the following format:

GRP	"*pattern*" ""	*aTag* NOTAG RAW SKIP	M O C	*min*	*max*	ACC *nb*	R T COND *name* "*feed*"	W F	"*description*"	/*suffix*

Formally:

```
GRP {"<id-pattern>"|""} {<XMLtag>|NOTAG|RAW|SKIP } ../..
    {M|O|C} <MIN> <MAX> ACC <ACC> ../..
   { {{R|T} {W|F}"}|{COND <name> "<feed>"} }} ../..
   "<description>"
   { /<suffix> }0..1
```

The syntax is just like a SEG line without the cut-function or cut-pattern specifications.

www.reverseXSL.com

A group is only a wrapper for segments and other sub-groups, as explained in §4.3.

5.4 MSG & END Definitions

The whole message is a top-level segment.

A MSG definition is a SEG definition with additional restrictions as follows:

MSG	"*pattern*" ""	*rootTag*	M	1	1	ACC 1	R T	W F	"*description*"	*cut-function* CUT "*pattern*"	*/suffix*

Formally:

```
MSG {"<id-pattern>"|""} <XMLroot> ../..
    M 1 1 ACC 1 ../..
    {R|T} {W|F} "<description>" ../..
    { <cut-function> | CUT "<cut-pattern>" }
   { /<suffix> }0..1
```

The keyword is changed from SEG to MSG, and a large part of the rest of the line is imposed. The syntax is however matching exactly that of a segment.

The END line is the last of the message definition, as follows:

```
END
```

Another difference is that the MSG definition is always starting in column 0; in other words, there is no '|' level indicator in front of such line.

The optional namespace suffix must start with a "/" . It will be appended to the default or explicitly SET namespace as described in §6. This will become the top-level namespace for the root element of the generated XML document, and all descendant elements, unless explicitly amended within groups or segments.

5.5 COND Definitions

Named conditions are declared as follows:

COND	*name*	"*pattern*"	DEPTH *level*	R T	W F	"*description*"

Formally:

```
COND <name> "<pattern>" DEPTH <level> ../..
    {R|T} {W|F} "<description>"
```

Where:

`<name>`	the name of this condition (unique within a given DEF file).
`<pattern>`	is the verification pattern for the condition. The condition is verified only if: `Pattern.compile(pattern) .matcher(inputString) .matches()` yields TRUE. Where the input string results from the concatenation (without extra separators) of the strings returned by the various GRP, SEG or D definitions that contained COND <name> <feed> in their definition. The concatenation is further limited to all GRP, SEG and D elements for repeating instances of the specified depth level and at all child levels.

Examples:

These two data elements are mutually exclusive:

```
|||D "(.*)" D1 C 0 1 ACC 1 COND exclusive "X" ALPHA
|||D "(.*)" D2 C 0 1 ACC 1 COND exclusive "X" ALPHA
```

The following condition does check it:

```
COND exclusive "X" DEPTH 3 R W "only one of D1 or D2 allowed"
```

The first element is mandatory, and the second data element is required when the first is coded KG:

```
||D "(.*)" D1 C 1 1 ACC 1 COND depend "(.*)" ALPHA
||D "(.*)" D2 C 0 1 ACC 1 COND depend "OK" NUMERIC
```

The following condition does check it:

```
COND depend "KGOK" DEPTH 2 R W "D2 required when D1 is 'KG'"
```

Note that the MIN occurrence count of D1 is set to 1 in order to raise an exception whenever the first element is missing.

5.6 MARK Definitions

Marks are declared as follows:

MARK	*aTag*	COND *name* "*pattern*"	"*value-if-true*" "NULL"	"*value-if-false*" "NULL"

Formally:

```
MARK <XMLtag> COND <name> "<pattern>" ../..
           "<value if true>" "<value if false>"
```

Where:

`<XMLtag>` is the XML data element name to create; i.e. it will yield

`<myXMLtag>value if true or value if false</myXMLtag>`

`<name>` the name of the relevant named condition to evaluate (unique within a given DEF file).

`<pattern>` is the verification pattern to apply to the condition. The evaluation is the result of:

```
Pattern.compile(pattern).matcher(inputString)
.matches()
```

and yields TRUE or FALSE.

`<value if true>` `<value if false>` are the text values to attach to the XML tag according to the evaluation of the pattern just above. The value can be "" (the empty string) in which case a *nillable* XML element (i.e. <tag />) is inserted in the output. The value can also be set to the reserved keyword "NULL" (quotes included please) in order to suppress any XML output for either true or false instances.

It's important to note that the MARK borrows the name of a declared named condition (so as to collect tokens generated under such condition) although it does not use any parameter from the declaration of the condition itself:

- The MARK bears its own—local—evaluation pattern;
- The depth scope is implicitly that of the mark element itself
- The Record/Throw and Warning/Fatal indicators are irrelevant, because a MARK would never generate an exception. There's always a true or false outcome, even if the collection of tokens is empty (patterns allow accepting or rejecting empty strings as needed).

5.7 Comments

Any line starting with a space or tab character is assumed to be a comment and is ignored by the parser.

This is very convenient as a scratch pad zone to store copies of various definitions, other sample structures, message specs cut from manuals, sample messages, etc.

The parser also ignores any line after the END keyword, whatever the line format. The line may even begin with a non-space character.

The DEF file section after the END keyword is a very good place for saving draft definitions, pasting a sample message, discussing alternative parsing, and in general documenting the works.

www.reverseXSL.com

6. Namespaces (SET BASENAMESPACE)

Namespace URIs are essential elements of an XML document.
XML documents without any namespace can be produced by the reverseXSL Transformer, although poor practice.

Namespaces play an essential role in XML. Every XML node with a name, namely elements and attributes, can have a namespace attached to it. The namespace is formally a Uniform Resource Identifier (URI) or Uniform Resource Name (URN). The difference doesn't matter because it is simply a string that qualifies the 'vocabulary' to which the XML tags used in a particular document belong. The namespace often looks like a web site URL although it does not need to.

Namespaces allow differentiating for instance an XML element like ' Address ' as defined by CompanyA, EnterpriseB, and AssociationC. The respective and qualified element names may indeed be like "http://www.CompanyA.com/xml/ce:Address", "EnterpriseB.edi.XML.global:Address", and "urn:AssocC:tradebook3:Address".

The most common practice is to qualify all element names with namespaces, but not the attribute names. In fact, attributes are attached to elements and can therefore enjoy a distinguished semantic by such unambiguous relationship to a qualified element.

Namespaces are most often rather long strings which would significantly increase the size of an XML document if attached to every XML tag. Therefore, their representation in an XML document proceeds in two steps: a) the declaration of namespace prefixes, b) the qualification of element tags with the declared prefixes.

One shall remind that:

- The value of a namespace prefix is meaningless in itself. They are just a shorthand notation for namespaces, purely local to an XML document, and often only to a fragment of an XML document. Therefore, namespace prefixes in output XML documents are allocated by the XML processing libraries (JAXP, XSLT engines, etc.) and not by the application.
- Namespaces are explicitly attached to each XML document element. There is no inheritance scheme but only shorthand notations that for instance allow promoting a namespace as the default one and consequently omit the namespace prefix in front of XML tags. Therefore, the declaration of namespace prefixes, the decision to make one as default instead of another, and the optimization of the scope of each prefix is also under control by the XML processing libraries (JAXP, XSLT engines, etc.) and not by the application. One can notably observe, dependent from your configuration, the repeated declaration of the same namespace under different prefixes within the XML documents generated by the Parser or XSLT. Although some XML representations may lack obvious simplifications in the distribution of namespace prefixes, XML documents with different prefix values and prefix declaration points are perfectly equivalent with each other whenever they yield the same 'expanded' version (if you replace all prefixes by the full namespace URIs and suppress all namespace prefix declarations).

With the above reminders, we can now explain how the reverseXSL software handles namespaces.

The lower-level Parser API provides an equivalent method to set the base namespace.

Beware, base namespace URIs are case sensitive.

The good use of namespaces may considerably simplify the management of XML schemas, the generation of Java classes with JAXB, and the development of XSLT mappings.

[1] Every DEF file contains the implicit or explicit declaration of a single base namespace.

- Implicit case: the DEF file does not contain a SET BASENAMESPACE statement. The base namespace URI is then either one set here, else the default "http://www.reverseXSL.com/FreeParser".
- Explicit case: the DEF file contains the statement:

```
SET BASENAMESPACE "<namespaceURI>"
```

For instance:

```
SET BASENAMESPACE "www.xyz.biz/xml"
```

Where:

`<namespaceURI>` is simply the URI string, e.g. "www.xyz.biz/xml".

[2] The base namespace URI is then used to qualify all elements of the XML document generated by the parser. However, Groups and Segments provide the option to extend the base URI with a suffix (cfr §5.1, 5.3, 5.4), yielding for instance namespaces like: www.xyz.biz/xml/GeneralHeader, www.xyz.biz/xml/CommonElts, www.xyz.biz/xml/order/type1, and so forth.

[3] Early XML documents had no namespaces, and some applications may still require XML documents without any namespace declaration. This is still permitted by the XML standards. The reverseXSL Parser provides the option of using the reserved URI value "NoNamespace", with the effect of suppressing namespaces from the output XML document (no suffix shall be used indeed, only the statement `SET BASENAMESPACE "NoNamespace"` in the relevant DEF file).

www.reverseXSL.com

7. Advanced Questions

1. Are Named Conditions always verified? 39

2. Can we use reserved characters like " (double quotes) within regular expressions?..39

3. In case the source message misses important data elements of a given structure, when would the parser assume that a segment (or group) structure is anyhow existing, but incompletely matched? .40

4. Is the parser able to skip a bad line in the input message and resume parsing as if it never existed? (Backtracking)....................41

5. Can we use 'negative' identifiers for segments and groups? In other words, specify a group of segment to be identified when the input data is NOT matching a specified pattern?.............................41

6. What is the meaning of 'optional' cardinality? 'Empty' or 'not-there' at all? Can we distinguish empty data elements values from optional data elements?..42

6+. Remove Non-Repeatable Nil Optional Elements 46

7. How can we define a data element to match the empty string or else a precise pattern? 47

8. Can we verify that a data element value matches a limited set of codes? (Code lists)..48

www.reverseXSL.com

Are Named Conditions always verified?

When no tokens at all were collected under a given name during the parsing, the associated named condition is not checked.

In fact, a named condition may often express interdependencies in between a few data elements, and all of them may be contained in an optional structure. Checking that the relevant condition would accept, in this case, the empty string has nothing to do with the interdependency itself but with the existence or absence of the whole containing structure.

Can we use reserved characters like " (double quotes) within regular expressions?

Indeed, there are no means to escape the regular expression quoting character. For instance, to perform a CUT on double quotes one can write:

`CUT-ON-(")` using the simple one char cutting function;

Note that there are no restrictions in using:
\n for a LF
\r for CR
\t for TAB
\f for FF
\e for ESC
\cx for ctrl-x
\xhh for hexhh
\uhhhh for Unicode

But one cannot write:

`CUT-ON-"""` i.e. using the CUT-ON-"<regex>" function.

Similarly, while attempting to parse a CSV input with quoted string values alike: 123,"ABC","some more text",12.5,"DEF"

We may wish to follow a segment CUT-ON(,) by a set of data element validation functions that would remove the double quotes; something alike:

```
||SEG ….CUT-ON-(,)
|||D ""(.*)"" …
```

The above example definitions featuring a " within the framing "…" would simply fail to load. Attempting to escape the double quote as in "\"" would also fail.

However, the DEF file loader does accept other characters than the " (double quote) to frame regular expressions. In other words, where a regular expression is written "ABC" one can also use /ABC/ or 'ABC' or #ABC#, else !ABC! and so forth. Regular expressions that may be noted as such are used for segment or group identification, data element validation, CUT-ON-specs, CUT patterns, REPEATED-"…", condition feeding, and condition evaluation expressions.

Actually, the first single non-space character encountered in front of a regular expression will be the one used to delimit the end of the expression as well. Any printable character can be used.

Re-writing the above examples, the DEF loader will perfectly accept:

```
CUT-ON-'"'
CUT-ON-/"/
CUT-ON-("( 
```
[6]

[6] CUT-ON-("(is a variant of CUT-ON-"**regex**" whereas CUT-ON-(") is a variant of CUT-ON-(**char**) !

www.reverseXSL.com

```
CUT-ON-*"*
|||D /"(.*)"/ …
|||D '"(.*)"' …
|||D -"(.*)"- …
```

And so forth…

In case the source message misses important data elements of a given structure, when would the parser assume that a segment (or group) structure is anyhow existing, but incompletely matched?

The same rules apply to both groups and segments

The rules are as follows:

- **If the first mandatory piece of a group or segment is not found, and no previous optional piece existed or was already matched, the whole group or segment structure is assumed as entirely missing**.
- Vice-versa, any successful attempt in matching a piece of a segment or group before or at the first encounter with the first mandatory piece as above would entail the existence of the whole structure and therefore the attempt to match the rest of this structure.

*As a consequence, two successive groups or segments cannot be distinguished from each other by just using different combinations of mandatory and optional cardinalities on sub-elements. But one can instead use attribute **Marks** to explicitly indicate which sub-element combination is which variant.*

Let's illustrate.

The following definition:

```
|SEG "^/" ServiceInfo          …          CUT-ON-(/)
||D "A(.*)"     ServiceA       O 0 1 ACC 1
||D "B(.*)"     ServiceB       O 0 1 ACC 10
||D "C(.*)"     ServiceC       M 1 3 ACC 3
||D "D(.*)"     ServiceD       O 1 3 ACC 3
||GRP "^E"      SubGroupE      O 1 3 ACC 3
|||D "(.*)"     SpecialE       M 1 10 ACC 10
|GRP "^/" SupplementalInfo     …
||D "(.*)"  ServiceA       M 1 1 ACC 1
```

Would cause the segment to exist if any of ServiceA, ServiceB or ServiceC data piece is found. But if neither of them were found, the parser will not look further (data ServiceD or SubGroupE are just ignored); the parser will assume that the segment ServiceInfo does not exist, and jump directly to the next definition, here 'SupplementalInfo', and then attempt to match the source data against that one.

On the other hand, an ability to match the segment identification (here a starting "/" char) followed by a successful match on ServiceA, or on ServiceB for instance will cause an instance of the segment ServiceInfo to exist. That may in turn cause exceptions to be raised for any further mandatory piece of that segment that would be missing in the source data message, but such missing mandatory pieces will not cause the existence of segment ServiceInfo to be reconsidered even if discovered while matching optional data pieces in the first place.

Is the parser able to skip a bad line in the input message and resume parsing as if it never existed? (Backtracking)

Usually, a bad input line in a source message would cause the parser to try every possible subsequent definition against it till the complete list of candidate element definitions is exhausted. A validation exception will be recorded for every mandatory sub-element (seen missing because unable to match it) till there are no more definitions to try or the maximum count of recorded exceptions is reached. At such a point, all the rest of the input message starting with the bad line would be considered a raw data for which no definition is left.

That was the behaviour of a first version. This behaviour has been amended to provide more flexibility. The parser proceeds as follows:

- The parser counts successive failed attempts to match definitions without any progress being made.
- The parser also keeps a trace of the first unmatched definition before such repetitive failures.
- When the relevant failure count reaches a given threshold (by default after three unsuccessful attempts to match the same input data piece against different definitions), the parser will assume that what is wrong is not about the message definition, but about the message data.
- At that point, the parser decides to skip the bad input piece of data (still it does exist in the generated tree as RAW untagged data), and tries to resume parsing as if such piece of data were not there. Henceforth, the parser backtracks into the message definition to the point of the first unmatched definition before the problem occurred.
- While backtracking, the parser would also 'unwind' any MARKs that were generated on the path forward through unmatched definitions.

The net result is that a piece of input data is skipped when too many errors are raised in sequence about it.

Note that anyhow, the relevant exceptions reporting repetitive failed attempts to match the bad data are kept on records within the Parser and made available in the total counts of errors at the end of the parsing.

Can we use 'negative' identifiers for segments and groups? In other words, specify a group of segment to be identified when the input data is NOT matching a specified pattern?

Let's remind first the positive cases:

The simplest form of segment or group identification is indeed to match a given pattern like:

> "^/" for a group or segment starting with a / character followed by anything else.
>
> "^DIM" for a group or segment starting with "DIM" (usually denoted as the segment or group tag)

www.reverseXSL.com

"ABC" for a group or segment containing the pattern "ABC" anywhere else (beginning, end, middle or even at several places).

"^A.*;$|^B.*;$" for a segment or group starting with either A or B and terminating with a semi-colon ';' in either case.

All the above examples are positive cases: the identification pattern is one amongst a limited set of possible cases.

Now, assume that we are willing to match a structure that is neither of the potentially following ones. E.g. we have a segment or group that is defined as neither starting by DIM, nor by OSI. Can we specify that?

The solution is to use in the regular expression non-capturing groups with logic modifiers.

The solution to the above case (assuming a segment) is:

```
|||SEG "^(?!(?:DIM)|(?:OSI))" …
```

Will fail against DIMsomething…

Will fail against OSIsomething…

Will match against LUXsomething… (because it starts neither with DIM, nor with OSI)

Decomposed:

- **^**… indicates to match from the start of data only
- (**?**…) is non-capturing
- (?**!**X) matches X via zero-width negative look ahead
- (?**:**X) is simply X as non-capturing group
- (?:X)**|**(?:Y) is either X or Y, both as non-capturing groups

What is the meaning of 'optional' cardinality? 'Empty' or 'not-there' at all? Can we distinguish empty data elements values from optional data elements?

There is a significant difference in XML between a data element that is absent and one that exists but with a null value. In the later case, the element is said to be 'nillable', as declared in XML schemas by:

```
<xs:element name="D1" type="xs:string" nillable="true"/>
```

And so an XML parser would accept:

```
<S1>
    <D1></D1>
</S1>
```

as well as

```
<S1>
    <D1/>
</S1>
```

By default, an 'optional' element, be it a segment or data element, will be considered missing (zero occurrence) when 'not there' at all, and only so.

In case of a group, a group being just the logical grouping of other groups, segments or data elements, the group is missing or existing according to the existence or absence of its member elements.

If there is an empty (zero-length) text element 'facing' the relevant optional segment or data element, and such optional element cannot accept matching against a zero length text element, then the element is skipped (zero occurrence) and the next definition is tested, again for accepting or not a zero length text element.

Let's illustrate the implications of these rules.

Consider the following dummy message:

`AAA.12.34.55....99...'`	*Can accept up to 10 elements with 2 digits.*
`BBB+001+9+876'`	*Features 3 data elements, all of them optional.*
`BBB+002++543'`	
`BBB+003'`	*Trailing '+' separators can be omitted.*
`BBB'`	*A TripleB segment with no elements.*
`'`	*A completely empty TripleB segment.*
`DDD/T12/K25/F32'`	*Each element is 'sub'-tagged by T, K or F*
`DDD/K17/F5'`	*with the consequence that '/' separators*
`DDD/T44/F92'`	*can also be omitted.*
`EEE/END'`	*Marking the end of the message.*

And the following message definition:

```
MSG   "^AAA"   ROOT   M 1 1  ACC 1  R W "whole message" CUT-ON-/'\r\n/

|SEG  "^AAA"   TripleA M 1 1  ACC 1    R W "triple A" CUT-ON-(.)
||D   "(AAA)"  SKIP    M 1 1  ACC 1    R W "triple A tag" ASMATCHED
||D   "(.*)"   TA1     M 0 10 ACC 99   R W "data TA1" DIGIT

|SEG  "^(BBB)?"  TripleB O 0 10 ACC 99 R W "triple B" CUT-ON-(+)
||D   "(BBB)?"   SKIP    M 1 1  ACC 1  R W "triple B tag" ASMATCHED
||D   "(.*)"     TB1     O 0 1  ACC 1  R W "data TB1" DIGIT
||D   "(.*)"     TB2     O 0 1  ACC 1  R W "data TB2" DIGIT
||D   "(.*)"     TB3     O 0 1  ACC 1  R W "data TB3" DIGIT

|SEG  " (^CCC.*$)|(^$)" TripleC O 0 10 ACC 99 R W "segC" CUT "CCC(.*)"
||D   "(.*)"            TC1 O 0 1 ACC 1 R W "data TC1" ASMATCHED

|SEG  "^DDD"   TripleD M 1 10 ACC 99   R W "triple D" CUT-ON-(/)
||D   "(DDD)"  SKIP    M 1 1  ACC 1    R W "triple D tag" ASMATCHED
||D   "T(.*)"  T       O 0 1  ACC 1    R W "data T" DIGIT
||D   "K(.*)"  K       O 0 1  ACC 1    R W "data K" DIGIT
||D   "F(.*)"  F       O 0 1  ACC 1    R W "data F" DIGIT

|SEG  "^EEE"   SKIP    O 0 10 ACC 99   R W "triple E" CUT-ON-(/)
||D   "(EEE)"  TripleE M 1 1  ACC 1    R W "triple E tag" ASMATCHED
||D   "(END)"  END     O 0 1  ACC 1    R W "triple E filler" ASMATCHED
END
```

At the segment level

The question is: looking at the above message and definition, shall we consider the empty line between BBB and DDD instances as a missing instance of TripleC because TripleC is optional? The answer is: it all depends from which definition can accept this empty structure.

Looking closer at the definitions of TripleB and TripleC, both can accept an empty line, and moreover, TripleB accept count is up to 99. The conclusion is that a repetition of Triple B will be matched against the empty line.

This definition yields the XML output (augmented with comments in italics):

```
<?xml version="1.0" encoding="UTF-8"?>
<ROOT messageID="1234567890">
  <TripleA>                 matched against 'AAA.12.34.55....99...'
    <TA1>12</TA1>
    <TA1>34</TA1>
    <TA1>55</TA1>
    <TA1/>
    <TA1/>
    <TA1/>
    <TA1>99</TA1>
    <TA1/>
    <TA1/>
    <TA1/>
  </TripleA>
  <TripleB>                 matched against 'BBB+001+9+876'
    <TB1>001</TB1>
    <TB2>9</TB2>
    <TB3>876</TB3>
  </TripleB>
  <TripleB>                 matched against 'BBB+002++543'
    <TB1>002</TB1>
    <TB2/>
    <TB3>543</TB3>
  </TripleB>
  <TripleB>                 matched against 'BBB+003'
    <TB1>003</TB1>
  </TripleB>
  <TripleB/>                matched against 'BBB'
  <TripleB/>                matched against ''
  <TripleD>                 matched against ' DDD/T12/K25/F32'
    <T>12</T>
    <K>25</K>
    <F>32</F>
  </TripleD>
  <TripleD>                 matched against 'DDD/K17/F5'
    <K>17</K>
    <F>5</F>
  </TripleD>
  <TripleD>                 matched against ' DDD/T44/F92'
    <T>44</T>
    <F>92</F>
  </TripleD>
</ROOT>
```

Note that segment tags are matched onto SKIP elements!

The entire TripleE segment is SKIP!

Where we see not one but two Triple B instances noted `<TripleB/>` because the tag is parsed as an element SKIP.

It TripleB identification expression is changed to "^BBB" then the parsing will yield a TripleC segment noted in XML:

```
..<TripleC>
```

```
  <TC1/>
</TripleC>
```

Because TC1 as data element is also defined to accept empty values.

If we modify further the definition of TC1 as follows (empty values no longer accepted):

```
||D "(.+)" TC1 O 0 1 ACC 1 …
```

Then we get the XML output:

```
…
<TripleB/>                  matched against 'BBB'
<RAW L="6" O="0"/>          ??? against ''
<TripleD>                   matched against ' DDD/T12/K25/F32'
  <T>12</T>
  …
```

The absence of any element in TripleC provokes the absence of TripleC itself (see "I*n case the source message misses important data elements …*" pg 38). The parser is then unable to match the empty line against any definition and leaves it untagged as 'RAW' data. Relevant exceptions are of course raised.

The same problem would occur if neither segment (TripleB or TripleC) is adjusted to accept a match against an empty line.

At the data element level

It's worth noting that:

- The TripleA segment in XML alternates empty with non-empty elements in the same sequence as in the original message. If TA1 elements are positional, the empty elements '<TA1 />' play an important marker role. The TA1 element definition must accept empty values.
- On the other hand, the data elements in TripleD do not accept empty values and are actually distinguished from each other by the first letter, leading to an output where only the elements being present are contained.
- In the middle of these two, we have the TripleB segment: Data element TB2 in the TripleB segment is neither always present even if empty like TA1, neither always absent if empty like the T, K, F elements in TripleD. Indeed the 3rd, 4th and 5th instances of TripleB do miss entirely the TB2 element due to trailing separator omission rules, whereas we have an nil element <TB2 \> in the second TripleB. Consequently, a test on the presence of the TB2 element in the XML output would not reflect the actual presence or absence of this element in the original message, but the fact that it is followed or not by more data!

This is the behaviour by default! It can be changed, see further.

At first, we may think to solve this issue by removing all empty elements from the message; but then we would break the positional rank of data elements TA1 in TripleA.

We can in theory apply two opposite rules:

A. Forcing the generation of all empty optional data elements, even when 'not there' at all.

www.reverseXSL.com

or instead,

This rule is also applicable to Segments! (ignoring the min size constraint)

B. asking to remove nil data elements under certain conditions:
when the element is optional or conditional (indeed!)
and the element cannot repeat (i.e. ACC 1)
and its specified minimum size is greater than 0

Rule A is not practically feasible, because we can mix data elements, sub-segments and sub-groups within the same segment or group, and it is therefore very difficult to set a rule by which these sub-segments and sub-groups should also be included (or not) into the scope of nil element generation. Moreover, that rule may yield very lengthy XML messages, heavy and complex to manipulate in XSL given the large number of nillable elements that will match XPath expressions but bring no data.

Remove Non-Repeatable Nil Optional Elements

Rule B is actually quite useful on messages based on the principle of positional data elements within 'segments' (e.g. EDIFACT, TRADACOMS, X12, etc.), like TripleB in the above sample. Indeed, most positions (think 'slots') in such segments are occupied by optional data elements, all unique and distinguished by their relative position in the 'segment'. Every unoccupied position will yield a corresponding NIL data element in XML, which can be suppressed from the XML output if Rule B is applied.

The interest is further strengthened by the ability to specify fix size optional elements like `[3..3]`, without being forced to turn it to `[0..3]`, as a means to prevent insufficient-length exceptions.

NIL data elements are suppressed only if they have a min/max size specification (of the kind `[1..15]`) with a minimum of at least 1. Obviously, if 0 is an acceptable size, there's no reason to suppress the element.

Moreover, the element must be non-repeatable otherwise there is a risk to suppress first and intermediate elements causing undesirable rank shifts.

Rule B shall be applied to both data elements and segments, because one can always decide to nest structures[7] at a further stage and the rule must be independent from such decision.

Rule B is activated through the method call:

```
Parser myparser = Parser(…);
myparser.removeNonRepeatableNilOptionalElements(true);
```

The method must be invoked before the execution of the parsing in itself (i.e. `myparser.parse(…)`).

If so invoked, we get:

```
…
<TripleB>                    matched against 'BBB+002++543'
  <TB1>002</TB1>
  <TB3>543</TB3>
</TripleB>
<TripleB>                    matched against 'BBB+003'
  <TB1>003</TB1>
</TripleB>
```

<TB2 /> removed

[7] We shall apply the same regime to simple and composite data elements in EDIFACT for instance, hence to data elements and segments in the present parser, because an EDIFACT composite element is modeled here as a sub-segment.

TripleC matched against '' and removed!

```
<TripleB/>                 matched against 'BBB'
<TripleD>                  matched against ' DDD/T12/K25/F32'
  <T>12</T>
  <K>25</K>
  <F>32</F>
</TripleD>
…
```

Provided that we changed the definition of the Triple B segment as follows:

```
|SEG   "^BBB"     TripleB O 0 10 ACC 99  R W "triple B" CUT-ON-(+)
||D    "(BBB)?"   SKIP    M 1 1  ACC 1   R W "triple B tag" ASMATCHED
||D    "(.*)"     TB1     O 0 1  ACC 1   R W "data TB1" DIGIT [3..3]
||D    "(.*)"     TB2     O 0 1  ACC 1   R W "data TB2" DIGIT [1..1]
||D    "(.*)"     TB3     O 0 1  ACC 1   R W "data TB3" DIGIT [3..3]
```

And for Triple C, we had to set ACC 1:

```
|SEG  " (^CCC.*$)|(^$)" TripleC O 0 10 ACC 1  R W "segC" CUT "CCC(.*)"
||D   "(.+)"           TC1 O 0 1 ACC 1 R W "data TC1" ASMATCHED
```

How can we define a data element to match the empty string or else a precise pattern?

The context is typically about positional data elements, optional and non-repeating, that may contain a piece of structured data alike a telephone number (from which we want for instance to extract only digits) or nothing. According to the above rules for suppressing Nil XML elements, one must also allow to capture the empty string as input.

The 'telephone number' solution is as follows:

```
||D  "\+?(?:00)? ?(\d*) ?\(? ?(\d*) ?\)? ?(\d*) ?(\d*) ?(\d*)
 ?(\d*)|()"  TEL O 0 1 ACC 1 R W "telephone" DIGIT [5..]
```

This pattern is able to remove a leading + or 00 or even +00, remove a pair of parenthesises, and remove single space chars between up to 4 groups of digits next to the closing parenthesis.

Note that it is important to:

- Put the empty-string match at the end of the regular expression: the order of regular expressions separated by | (OR symbol) is not neutral.
- Not forgetting to have a minimum size above zero if we want to ever benefit from the rule `removeNonRepeatableNilOptionalElements` (cf previous question).

Can we verify that a data element value matches a limited

set of codes? (Code lists)

The solution is to provide the list as OR'ed capturing groups; for instance:

```
D "(RED)|(GREEN)|(BLUE)" ColorCode … ASMATCHED
```

An alternative is to verify the set of values with a named condition. The later provides the option, by selecting a proper condition feeding pattern, to verify only a portion of the entire data value.

8. Sample Program Code

The following sample demonstrates the use of the 'low-level' parser API. We do recommend that you consider using the Transformer-Factory and Transformer API (see the corresponding software manual) for a more functional environment, less coding, and increased productivity.

Please refer to the documentation of all methods of the Parser class in the JavaDoc (included in the software distribution jar).

```
//variables
int totalRecordedExceptions;

//Constants: number of exceptions that we can accept to record before
//interrupting message parsing

final int MAXFATAL = 5; //max count of exceptions of type FATAL
final int MAXTOTAL = 10; //max count of all exceptions

//Load a Definition object from a file

FileReader fr = new FileReader("sample.DEF");
LineNumberReader lnr = new LineNumberReader(fr);

Definition def = new Definition();
try {
    //loading a definition can throw exceptions
    /(no way and non-sense to record them silently)
    def.loadDefinition(lnr);
} catch (Exception e) {
    System.out.println(e.toString());
    //One would put more exception handling code here;
    //not in this sample
}
```

one can also use the variant constructor method with one more int argument telling the acceptable count of successive matching errors that will trigger a backtracking attempt (default is 3).

```
//Create a parser for this Definition

Parser parser = new Parser(def,MAXFATAL, MAXTOTAL);

//The parser can get the message instance from a LineNumberReader or a
//String; Let's assume that the message text to parse is in
//messageStr, that the message ID is in messageIDStr, and that
//(for reference) the message text was extracted after a number of
//lines specified in headerLineNb

//at option, we may want to call methods like (cfr §7[6+]):
```

www.reverseXSL.com

```
parser.removeNonRepeatableNilOptionalElements(true);

try {
    //The parser throws exceptions if a MAX count is reached while
    //parsing: MAXFATAL or MAXTOTAL.
    totalRecordedExceptions = parser.parse(    messageIDStr,
                                               messageStr,
                                               headerLineNb );
} catch (Exception e) {
    System.out.println(e.toString());
    //One would put more recovery code here; not in this sample
}

if (totalRecordedExceptions>0) {

    //We got errors; let's iterate through Exceptions
    ExceptionListIterator exIter = parser.exceptionIterator();
    //Trace exception counts; one could use .size() but
    //There are extra specialized methods:
    System.out.print("Fatal Exception count: "
                    + exIter.fatalExceptionsCount()
                    +" (of " + exIter.totalExceptionsCount()
                    +" total)\n");
    //This specific 'Exception' ListIterator allows to iterate
    //only over the subset of fatal exceptions:
    while (exIter.hasNextFatal()) {
        System.out.println(exIter.nextFatalException().toString());
        };
    //We can also dump the parser state;  as simple as that:
    System.out.println(parser.toString());

    //We can well decide to continue processing the message after
    //such error reporting activities.
    //… see sample code further below… you may want to mark to message
    //or use an alternative channel for these suspicious messages.
}
else {
    //Everything went fine; generate XML; the first argument (as true)
    //instructs to preserve RAW data in output; there shall be none
    //because parsing went fine, except for XML tags explicitly set to
    //the reserved value "RAW".

    //Set the namespace for XML output
    parser.setBaseNamespace("http://www.artofe.biz/broker");

    //getXML returns a StringWriter object
    String xmlOutput = parser.getXML(true,false).toString();

    //We can invoke the post-parsing methods (.toSTring(), .getXML(),
    //.setNameSpace(), .exceptionIterator()) as many times as desired;
    //the same parser object can be re-used to parse many messages;
    //the parser state is implicitly reset with every new
    //parsing: parser.parse(...)

}
```

www.reverseXSL.com

9. Command Line Tools

We assume that the reverseXSL .jar file is available on the CLASSPATH, else you will add the necessary –classpath *declaration to all java command lines below*

These command line tools are also contained in the software distribution jar.

Regex Tester

com.reverseXSL.util.RegexCheck

```
Usage :
java com.reverseXSL.util.RegexCheck
```

also available as an MS-Windows executable `RegexCheck.exe`

Allows experimenting with regular expressions in general, and lists the Parser behaviour if the relevant expression is used for identification, or data validation, or extraction pattern, or else as segment cutting function (CUT "pattern").

The higher level `Transform` *command line tool is documented in the main reverseXSL software manual*

The command-line Parser

com.reverseXSL.Parse

```
Usage :
java com.reverseXSL.Parse myDefinitionFile
```

Creates a sample XML message matching the definition

```
or :
java com.reverseXSL.Parse myDefinitionFile myInputMessageFile
```

Executes the Parser from the command line and outputs a dump of the parser state followed by the generated XML.

```
or :
java com.reverseXSL.Parse myDefinitionFile myTOKEN
```

This is a variant of the previous command where the actual message file name is taken from inside the DEF file itself[8]*. This mode is very useful to automate tests while developing new message transformations (e.g. via a run configuration in Eclipse, or as custom commands in StylusStudio).*

```
or :
java com.reverseXSL.Parse myDefinitionFile myInputMessageFileOrToken
                     <MaxFatalExceptions> <MaxExceptions> <true|false>
```

where the last argument tells to removeNonRepeatableNilOptionalElements

Note that the XML message is written on stdout, whereas the parser state dump is written on stderr

[8] if for instance the token is FILE01, you may insert a comment (i.e. with a space or tab as leading character) in the DEF file like:

#FILE01=myMessage.txt;

whose format is formally:

<at least one space or tab>**#**<token>**=**<filename>**;**

www.reverseXSL.com

It is then easy to capture just the XML output by calling:

```
java Parse myDefinitionFile myInputMessageFile >>out.xml
```

10. Philosophical Considerations

Recommended practice in using Fatal or Warning exception impacts, and Maximum exception counts

We propose the following very simple rules:

- Specify a **Fatal** exception **whenever the parsing is in danger of interpreting data wrongly**; e.g. mixing up a quantity with price info; taking one segment for another, and so forth.
- By extension, those errors at parsing time that may still preserve the proper decoding of the message but risk causing misinterpretations on the receiving application side shall also be recorded as fatal exceptions. Henceforth, specify a **Fatal** exception for all missing or miss-formatted elements that belong to the **semantic core** of a given message. For instance, the container number is obviously essential in a container movement message; without it, the message is useless. A shipment date is similarly essential, but the container contour code or a sender's reference may not affect the correct processing, especially when the target application can retrieve this information from its database or earlier messages.

*If the parser is at risk of slipping into wrong matches, this must be **Fatal**. If parsing remains on track, this is **warning**.*

- Whenever the Fatal Exception count is above 0, reject the corresponding message and report this back to the sender via a negative acknowledgement as applicable.
- The only interest in recording (and not immediately throwing) any fatal exception (even a single one) would be in testing or QA such as to speed-up corrections by fixing more would-be-fatal errors at once.
- Vice versa, use **Warning** exceptions for any deviation from the strict syntax **that would anyhow let the parser identify properly which data means what**, and let the receiving application process it. For instance:
    - Accept more occurrences of a group, segment or data element than formally allowed, given that the distinction with any following piece of data is clear and the receiving application can handle such extra data.
    - Let elements (segments, data elements) become optional where the formal definition tells that these are mandatory (e.g. use **O** **1** 1 ACC 99), provided too that suitable default values are available in the receiving application context.
    - Loosen data identification and extraction patterns, e.g. accepting lower case (and other characters) at the point of identification but checking for uppercase (or specific character sets) at the point of data value validation with an associated warning exception.

- Accept smaller and bigger size data elements than formally specified whenever the target can also handle it, and report any discrepancy as warnings at the point of [min..max] size validation.

- Use a high total maximum exception count like 15 or 20 (and max fatal exception count of 0) and kindly report any warning back to the sender while keeping on processing his message.
- Consider any number of exceptions in excess of this threshold as a sufficient reason to reject the whole message (and return negative acknowledgements). In other words, more than 15 or 20 warnings are worth one fatal exception…and that shall cause message rejection.

How extensively shall we enforce data validation?

Consider a few questions:

Shall the parser check that, for instance, an article ID or a flight number exists? Obviously no, because that is bound to business information and associated rules; e.g. the flight number exists but there's no such flight on that one day which is a holiday in the country where the airline is based…the article ID exists but is no longer shipped…validating such information is not the job of a data conversion engine.

Shall the parser check that a date at some position is after or before the date specified (or implied) at another place in the message? One would say no because that sounds again as a business rule…

Shall the parser verify that a given field is a valid date, e.g. unlike 32.13.2007? I would say no once more, because the parser does not need such test to effectively parse the message correctly! Moreover, applications can conventionally use dates like 99.99.9999 or 00.00.00 to say "unspecified". And would we check that Tuesday March 13th 2009 is actually not valid (it's a Friday!)?

Shall the parser check that the date is a valid date because such date can be equally written in the message as: 20061231 or 31122006? Yes here! Because the validity test is, in the present case, the only way to parse correctly this field and know which part is the year, which part is the month, and which one is the day…

> *To do such complex parsing one would define a mandatory group made of 2 optional data elements, each one bearing an identification pattern respectively as "20[0-9][0-9][01][0-9][0-3][0-9]" and "[0-3][0-9][01][0-9]20[0-9][0-9]" (valid between 2000 and 2099), followed a MARK to qualify the date format. One can also use the same patterns for two segments, each containing a different cut of the date into three data elements.*

The proposed rules are indeed that:

- **A parser shall use the message syntax to the extent necessary for the correct interpretation of which piece of data is which field.**

- **Beyond that, it is the responsibility of end applications to check the validity of data values.**

Note that the control of character sets by a parser is a common practice although in many places such controls would not contribute at all to the correct interpretation of message fields. It is indeed accepted that a violation of the standard character sets of data fields shall be reported as a message syntax error along with the verification of mandatory / optional and other cardinality constraints. Hence, the right place is also at parsing time.

However, controlling the validity of a date, or the fact that, for instance, a colour code belongs to a given set of values, or that a number fits within a given range, all that shall be avoided as much as possible.

Moreover:

- **One shall move as much as possible any data value validation to the `<char-spec>` functions in data element definitions**.

Let us clarify this principle with an example:

The following data:

```
CV7747/7751/21.LXKCV.LHR
```

Will match the following definition:

```
===== Flight section =====
|GRP    ""      FlightIdentification   M 1 1 ACC 1  R F "UCM-2.FI"
||SEG   ""      NOTAG       M 1 1 ACC 1 R F "FI" CUT-ON-(/)
|||SEG  ""      NOTAG       M 1 1 ACC 1 R W "FI1" CUT "^(..)(.{1,5})$"
||||D   "(.*)"  CarrierCode M 1 1 ACC 1 R W "CCode" ASMATCHED
||||D   "(.*)"  FlightNbr   M 1 1 ACC 1 R W "FFN" ASMATCHED
|||D    "([^.]{1,5})"  FlightNbr2  O 0 1 ACC 1 R W "SFN" ASMATCHED
|||SEG  ""  NOTAG   M 1 1 ACC 1 R W "DFA Grp" CUT-ON-(.)
||||D   "(..?)"     DayOfMonth    M 1 1 ACC 1 R W "DoM" DIGIT
||||D   "(.{5,7})"  Registration O 0 1 ACC 1 R W "AR" ASMATCHED
||||D   "(...)"     AirportCode   M 1 1 ACC 1 R W "ACoM" ASMATCHED
```

If we now parse the following input data:

```
CV7747/7751/2x.LXKCV.LHR
```

We get the error message:

```
ERROR P013: Parsing error about element <DayOfMonth>(DoM), context
[2x] at L:12 O:12, impact [Warning].
Caused by: ERROR P005: Data value invalid versus [Numerical-Digit]!
(validating this [2x] against "[0-9]*").
```

…which is what we shall indeed expect.

Let's now change the definition of the DayOfMonth element to:

```
||||D "([0-9][0-9]?)"  DayOfMonth   M 1 1 ACC 1 R W "DoM" DIGIT
```

The error message becomes:

```
ERROR P013: Parsing error about element <NOTAG>(DFA Grp), con-
text [2x.LXKCV.LHR] at L:12 O:12, impact [Warning].
Caused by: ERROR P007: Missing mandatory segment [NOTAG]!
(identifying this "" in [2x.LXKCV.LHR]).
```

…which would likely confuse the user about the real cause of his error.

www.reverseXSL.com

The fact is that the pattern of a data element, here "([0-9][0-9]?)" is meant as a validation and an extraction pattern. Validation means answering the question: "is this the right data element?" and indeed, the alternative definition here above yields NO! Because the pattern does not match the input string.

Consequently, this element being the first Mandatory piece of a segment, the Parser suspects that it entered into the segment by mistake and would give a chance to the following definition. But then the parser realizes that the segment definition that it is leaving is mandatory (and it just failed to match it because the first mandatory piece failed to...), hence the error message.

In the above example, because of the explict "." separators between data elements, a restrictive data element validation pattern is not helping at all the parsing process in itself, and so any validation must be moved to the <char-specs> functions.

There are however cases where the parsing of non-positional elements can only work with the help of validation; for instance, these three elements:

```
/1234/LHR/15122006
```

Could only be dissociated from the case where the middle one is absent

```
/1234/15122006
```

By testing that the second—optional—element is exactly 3 letters whereas the third (becoming second) element always starts with a digit!

In that later case, some extended data validation is required to drive a correct parsing.

Note that if we had the variant:

```
/1234//15122006
```

There is no problem to parse data elements using "(.*)" because the separators provide data elements with an explicit and fixed position in the segment.

What is the interest in ACCepting more than the maximum number of occurrences? e.g. 'O 0 3 ACC 99'

The interest is very high!

Indeed, if we have extra instances of some data structure in an input message (be it a data element, segment or group) and the parser stops at an ACCept count = MAX expected number, the parser will then try to match these extras against any of the following definitions till a mandatory one is found that does not match it, and then report an error about that one mandatory piece (possibly far down in the message) being missing! Although, the real cause is too many instances of the very first structure under consideration.

So the rules shall be:

- Provided that the parsing is not at risk of misinterpreting data, do use ACCept counts as high as possible, such as to benefit from explicit indications of excess occurrences.

- By the same token, relax the Mandatory conditions and turn them to Optional cardinality whenever the parsing could not slide into a wrong track of interpreting input data (e.g. there are explicit tags in front of every segment and the same segment does not reappear later in the message with another meaning).
- But in all cases do align <MIN> and <MAX> cardinality with the exact requirements in the standard message syntax.

Doing so, one would always get the proper error messages when instances are missing or in excess of the specifications.

11. Known Issues and Limitations

XML output indentation

The Parser has been originally developed for JRE 1.4. It works too under JRE 1.5 and later versions.

Between the JRE 1.4 and 1.5 changes have been introduced in JAXP with the consequence that the indentation of the XML output requires a different procedure in JRE 1.5 versus 1.4. The system adapts itself automatically to the environment.

However, we observed that indentation is not yielding similar results on all system configurations encountered so far and bearing either JRE 1.5 or 1.4. The most common variation is that of an XML output where indentation length is reset to 0. The differences worsen when followed by an XSLT transformation step. Yet, in all cases the variations are only visible in text based editors, given that the generated XML document is valid and formally identical in all cases. Moreover, the document is re-indented for display by most specialized XML editors, thus masking the differences in the distribution of space-only text nodes that count for indentations.

The transformer provides a special 'printableTransform()' method that compensates eventual differences in indentation logic. The later is a utility method that shall only be used for unit testing and manual inspection of XML outputs.

Using the same named condition in different parent structures

Consider the following message samples (truncated):

Sample with no Incoming and no Outgoing ULD data

```
UCM
…
IN
.NIL
OUT
.NIL
```

Sample with only Incoming ULD data

```
UCM
…
IN
.PCM11111CV.AHM22222LH.DKP33333AF.RET44444.UYT55555XQ
.PCM22222CV.AHM99999LH.DKP44444AF.ABC12345XQ
OUT
.NIL
```

Sample with only Outgoing ULD data

```
UCM
…
IN
.NIL
OUT
.AKL12345CV/JFK/B.AHM12345LH/SFO/A.RET12345
```

Sample with both Incoming and Out-going ULD data

```
UCM
…
IN
.PCM11111CV.AHM22222LH.DKP33333AF.RET44444.UYT55555XQ
OUT
.AKL12345CV/JFK/B.AHM12345LH/SFO/A.RET12345
.UKF54321CV/LUX/B.AHM12345CV/BRU/A
```

We see that the IN and OUT sections always contain a ’dot’-line, either with NIL alone, either full of ULD data. In the later case, the line can be repeated.

Let’s look at the following definition (truncated):

```
COND NILorFULL "X" DEPTH 2 R W "exclusive elements"

MSG     "^UCM$"         ROOT        …
…
    ===== Incoming ULD section =====
|GRP    "^IN$"  IncomingULD  M 1 1 ACC 1 R W "Incoming ULD section"
||D     "(IN)"  SKIP         M 1 1 ACC 1 R W "IN tag" …
||D     "(.NIL)"    NoIncoming  C 0 1 ACC 1 COND NILorFULL "X" …
||GRP   "^\."       NOTAG       C 0 1 ACC 1 COND NILorFULL "X" …
|||SEG  "^\."           IncomingULD M 1 99 ACC 99 …
||||D   …
||||D   …
    ===== Outgoing ULD section =====
|GRP    "^OUT$"  OutgoingULD     M 1 1 ACC 1 R W "Outgoing ULD section"
||D     "(OUT)"  SKIP         M 1 1 ACC 1 R W "OUT tag" …
||D     "(.NIL)"    NoOutgoing  C 0 1 ACC 1 COND NILorFULL "X" …
||GRP   "^\."       NOTAG       C 0 1 ACC 1 COND NILorFULL "X" …
|||SEG  "^\."           OutgoingULD M 1 99 ACC 99 …
||||D   …
||||D   …
…
END
```

The symmetry of the IN and OUT sections makes it rather tempting to share the same named condition (depth level 2) within IN and OUT sections, the latter being at depth level 1.

www.reverseXSL.com

However, assuming a good message alike:

```
UCM
…
IN
.NIL
OUT
.AKL12345CV/JFK/B.AHM12345LH/SFO/A.RET12345
```

The named condition tokens and occurrences will be recorded as:

```
-2-COLLECTED NAMED CONDITIONS CRITERIA:
Name: NILorFULL (2 token(s))
- 1- 1- 1 . . . . . . . . . . . . . . . . . . = [X]
- 1- 1- 1 . . . . . . . . . . . . . . . . . . = [X]
```

Causing the parser to report an exception because "XX" resulting from a depth level 2 evaluation will fail to meet the requirement for a single "X" as specified in the COND line.

We see that there's no break in occurrence numbers. Indeed, at depth level 1 (underlined above) we have the first IN occurrence as well as first OUT occurrence.

The isolation of Named Conditions for evaluation is **solely based on occurrence breaks**. The evaluation does not take into account any underlying structure name changes. This is **by design**, in order to allow named conditions to span across many different structures, whatever their names, tags or else.

The solution to our problem is very simple: one must, by design, force different names to different condition contexts or semantics, and refrain from the kind of re-use illustrated above.

The correct message definition becomes:

```
COND NILorIN "X" DEPTH 2 R W "exclusive within the IN section"
COND NILorOUT "X" DEPTH 2 R W "exclusive within the OUT section"

MSG     "^UCM$"        ROOT        …
…
    ===== Incoming ULD section =====
|GRP    "^IN$"  IncomingULD  M 1 1 ACC 1 R W "Incoming ULD section"
||D     "(IN)"  SKIP        M 1 1 ACC 1 R W "IN tag" …
||D     "(.NIL)"   NoIncoming  C 0 1 ACC 1 COND NILorIN "X" …
||GRP   "^\."      NOTAG       C 0 1 ACC 1 COND NILorIN "X" …
|||SEG  "^\."          IncomingULD M 1 99 ACC 99 …
||||D   …
||||D   …
    ===== Outgoing ULD section =====
|GRP    "^OUT$"  OutgoingULD    M 1 1 ACC 1 R W "Outgoing ULD section"
||D     "(OUT)"  SKIP        M 1 1 ACC 1 R W "OUT tag" …
||D     "(.NIL)"   NoOutgoing  C 0 1 ACC 1 COND NILorOUT "X" …
||GRP   "^\."      NOTAG       C 0 1 ACC 1 COND NILorOUT "X" …
|||SEG  "^\."          OutgoingULD M 1 99 ACC 99 …
||||D   …
||||D   …
```

We clearly have a IN section and a OUT section like pears and apples; their structural similarities shall not lead us to think that inside exclusivities could extend their meaning outside these respective contexts.

...

END

12. APPENDIX: Regular Expressions

Regexp tips:

- http://wesnerm.blogs.com/net_undocumented/2005/05/regular_express.html

  An article about performance of regexp's

- http://regexadvice.com/ & http://regexlib.com/ & http://www.regular-expressions.info/

  Online collections of tips

Essentials:

- The regexp tutorial at www.reverseXSL.com, dedicated to the fastest ramping up on Parsing tasks and DEF file development.
- http://java.sun.com/developer/technicalArticles/releases/1.4regex/

  Article: Regular Expressions and the Java Programming Language

- http://java.sun.com/docs/books/tutorial/essential/regex/groups.html

  A tutorial on **Capturing Groups**

- http://java.sun.com/docs/books/tutorial/essential/regex/quant.html

  Understanding greedy, reluctant and possessive behaviors

The rest of the present document is an excerpt of the standard java API specifications of the `Pattern` class. It is purely informative and supplied unmodified, as a convenience for quick reference. ***This information is copyrighted by SUN Microsystems****. It may have changed and you shall refer to the original documents and license published on* ***java.sun.com***. *We do not endorse this documentation as much as their authors do not endorse the reverseXSL products. No rights are granted nor transferred in any form, no claims are made.*

Class Pattern:

`java.util.regex.Pattern`

A compiled representation of a regular expression.

A regular expression, specified as a string, must first be compiled into an instance of this class. The resulting pattern can then be used to create a `Matcher` object that can match arbitrary character sequences against the regular expression. All of the state involved in performing a match resides in the matcher, so many matchers can share the same pattern.

A typical invocation sequence is thus

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

A `matches` method is defined by this class as a convenience for when a regular expression is used just once. This method compiles an expression and matches an input sequence against it in a single invocation. The statement

```
boolean b = Pattern.matches("a*b", "aaaaab");
```

is equivalent to the three statements above, though for repeated matches it is less efficient since it does not allow the compiled pattern to be reused.

Instances of this class are immutable and are safe for use by multiple con-

current threads. Instances of the Matcher class are not safe for such use.

Summary of regular-expression constructs

Construct	Matches
Characters	
x	The character x
\\	The backslash character
\0n	The character with octal value 0n (0 <= n <= 7)
\0nn	The character with octal value 0nn (0 <= n <= 7)
\0mnn	The character with octal value 0mnn (0 <= m <= 3, 0 <= n <= 7)
\xhh	The character with hexadecimal value 0xhh
\uhhhh	The character with hexadecimal value 0xhhhh
\t	The tab character ('\u0009')
\n	The newline (line feed) character ('\u000A')
\r	The carriage-return character ('\u000D')
\f	The form-feed character ('\u000C')
\a	The alert (bell) character ('\u0007')
\e	The escape character ('\u001B')
\cx	The control character corresponding to x
Character classes	
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)
Predefined character classes	
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]
POSIX character classes (US-ASCII only)	
\p{Lower}	A lower-case alphabetic character: [a-z]
\p{Upper}	An upper-case alphabetic character:[A-Z]
\p{ASCII}	All ASCII:[\x00-\x7F]
\p{Alpha}	An alphabetic character:[\p{Lower}\p{Upper}]
\p{Digit}	A decimal digit: [0-9]
\p{Alnum}	An alphanumeric character:[\p{Alpha}\p{Digit}]
\p{Punct}	Punctuation: One of !"#$%&'()*+,-./:;<=>?@[\]^_‘{|}~
\p{Graph}	A visible character: [\p{Alnum}\p{Punct}]
\p{Print}	A printable character: [\p{Graph}]
\p{Blank}	A space or a tab: [\t]

\p{Cntrl}	A control character: [\x00-\x1F\x7F]
\p{XDigit}	A hexadecimal digit: [0-9a-fA-F]
\p{Space}	A whitespace character: [\t\n\x0B\f\r]

Classes for Unicode blocks and categories

\p{InGreek}	A character in the Greek block (simple block)
\p{Lu}	An uppercase letter (simple category)
\p{Sc}	A currency symbol
\P{InGreek}	Any character except one in the Greek block (negation)
[\p{L}&&[^\p{Lu}]]	Any letter except an uppercase letter (subtraction)

Boundary matchers

^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

Greedy quantifiers

X?	*X*, once or not at all
*X**	*X*, zero or more times
X+	*X*, one or more times
X{*n*}	*X*, exactly *n* times
X{*n*,}	*X*, at least *n* times
X{*n*,*m*}	*X*, at least *n* but not more than *m* times

Reluctant quantifiers

X??	*X*, once or not at all
*X**?	*X*, zero or more times
X+?	*X*, one or more times
X{*n*}?	*X*, exactly *n* times
X{*n*,}?	*X*, at least *n* times
X{*n*,*m*}?	*X*, at least *n* but not more than *m* times

Possessive quantifiers

X?+	*X*, once or not at all
*X**+	*X*, zero or more times
X++	*X*, one or more times
X{*n*}+	*X*, exactly *n* times
X{*n*,}+	*X*, at least *n* times
X{*n*,*m*}+	*X*, at least *n* but not more than *m* times

www.reverseXSL.com

Logical operators

XY	X followed by Y
X|Y	Either X or Y
(X)	X, as a capturing group

Back references

n	Whatever the *n*th capturing group matched

Quotation

\	Nothing, but quotes the following character
\Q	Nothing, but quotes all characters until \E
\E	Nothing, but ends quoting started by \Q

Special constructs (non-capturing)

(?:*X*)	*X*, as a non-capturing group
(?idmsux-idmsux)	Nothing, but turns match flags on - off
(?idmsux-idmsux:*X*)	*X*, as a non-capturing group with the given flags on - off
(?=*X*)	*X*, via zero-width positive lookahead
(?!*X*)	*X*, via zero-width negative lookahead
(?<=*X*)	*X*, via zero-width positive lookbehind
(?<!*X*)	*X*, via zero-width negative lookbehind
(?>*X*)	*X*, as an independent, non-capturing group

where flags are:

- i *ignore case*
- d *'unix lines', i.e. only \n accepted as line terminator*
- m *MULTILINE mode, see further*
- s *DOTALL mode: '.' can match line terminators, see further*
- u *Unicode aware case folding*
- x *permit comments with #*

Backslashes, escapes, and quoting

The backslash character ('\') serves to introduce escaped constructs, as defined in the table above, as well as to quote characters that otherwise would be interpreted as unescaped constructs. Thus the expression \\ matches a single backslash and \{ matches a left brace.

It is an error to use a backslash prior to any alphabetic character that does not denote an escaped construct; these are reserved for future extensions to the regular-expression language. A backslash may be used prior to a non-alphabetic character regardless of whether that character is part of an unescaped construct.

Backslashes within string literals in Java source code are interpreted as required by the Java Language Specification as either Unicode escapes or other character escapes. It is therefore necessary to double backslashes in string literals that represent regular expressions to protect them from interpretation by the Java bytecode compiler. The string literal "\b", for example, matches a single backspace character when interpreted as a regular expression, while "\\b" matches a word boundary. The string literal "\(hello\)" is illegal and leads to a compile-time error; in order to match the string (hello) the string literal "\\(hello\\)" must be used.

Character Classes

Character classes may appear within other character classes, and may be composed by the union operator (implicit) and the intersection operator (&&). The union operator denotes a class that contains every character that is in at least one of its operand classes. The intersection operator denotes a class that contains every character that is in both of its operand classes.

The precedence of character-class operators is as follows, from highest to

lowest:

1	Literal escape	`\x`
2	Grouping	`[...]`
3	Range	`a-z`
4	Union	`[a-e][i-u]`
5	Intersection	`[a-z&&[aeiou]]`

Note that a different set of metacharacters are in effect inside a character class than outside a character class. For instance, the regular expression `.` loses its special meaning inside a character class, while the expression `-` becomes a range forming metacharacter.

Line terminators

A *line terminator* is a one- or two-character sequence that marks the end of a line of the input character sequence. The following are recognized as line terminators:

- A newline (line feed) character (`'\n'`),
- A carriage-return character followed immediately by a newline character (`"\r\n"`),
- A standalone carriage-return character (`'\r'`),
- A next-line character (`'\u0085'`),
- A line-separator character (`'\u2028'`), or
- A paragraph-separator character (`'\u2029`).

If `UNIX_LINES` mode is activated, then the only line terminators recognized are newline characters.

The regular expression `.` matches any character except a line terminator unless the `DOTALL` flag is specified.

By default, the regular expressions `^` and `$` ignore line terminators and only match at the beginning and the end, respectively, of the entire input sequence. If `MULTILINE` mode is activated then `^` matches at the beginning of input and after any line terminator except at the end of input. When in `MULTILINE` mode `$` matches just before a line terminator or the end of the input sequence.

Groups and capturing

Capturing groups are numbered by counting their opening parentheses from left to right. In the expression `((A)(B(C)))`, for example, there are four such groups:

1	`((A)(B(C)))`
2	`(A)`
3	`(B(C))`
4	`(C)`

Group zero always stands for the entire expression.

Capturing groups are so named because, during a match, each subsequence of the input sequence that matches such a group is saved. The captured subsequence may be used later in the expression, via a back reference, and may also be retrieved from the matcher once the match

www.reverseXSL.com

operation is complete.

The captured input associated with a group is always the subsequence that the group most recently matched. If a group is evaluated a second time because of quantification then its previously-captured value, if any, will be retained if the second evaluation fails. Matching the string `"aba"` against the expression `(a(b)?)+`, for example, leaves group two set to `"b"`. All captured input is discarded at the beginning of each match.

Groups beginning with `(?` are pure, *non-capturing* groups that do not capture text and do not count towards the group total.

Unicode support

This class follows *Unicode Technical Report #18: Unicode Regular Expression Guidelines*, implementing its second level of support though with a slightly different concrete syntax.

Unicode escape sequences such as `\u2014` in Java source code are processed as described in 3.3 of the Java Language Specification. Such escape sequences are also implemented directly by the regular-expression parser so that Unicode escapes can be used in expressions that are read from files or from the keyboard. Thus the strings `"\u2014"` and `"\\u2014"`, while not equal, compile into the same pattern, which matches the character with hexadecimal value `0x2014`.

Unicode blocks and categories are written with the `\p` and `\P` constructs as in Perl. `\p{`*prop*`}` matches if the input has the property *prop*, while \P{*prop*} does not match if the input has that property. Blocks are specified with the prefix `In`, as in `InMongolian`. Categories may be specified with the optional prefix `Is`: Both `\p{L}` and `\p{IsL}` denote the category of Unicode letters. Blocks and categories can be used both inside and outside of a character class.

The supported blocks and categories are those of *The Unicode Standard, Version 3.0*. The block names are those defined in Chapter 14 and in the file Blocks-3.txt of the Unicode Character Database except that the spaces are removed; `"Basic Latin"`, for example, becomes `"BasicLatin"`. The category names are those defined in table 4-5 of the Standard (p. 88), both normative and informative.

* * *