The reverseXSL Transformer

Main Software Manual

V2.1 Aug09	**web release**
V2.2 Apr12	Now as Open Source
V2.3 Dec12	Exception List can be retrieved in XML, added option `UnfoldPSCRMRemarks`

Foreword

This document describes the reverseXSL Transformer API, the associated command-line tools, and explains how to develop any-to-any structured message transformations with the software. A separate document describes the Parser DEF file format, one of the three meta-data elements that drive message Transformations.

Please refer to the web site www.reverseXSL.com for tutorials, tool-tips, samples, and advanced questions.

Table of Contents

updates available from:
www.reverseXSL.com

Formally…

1. What you should know

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

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

This manual describes how to configure and execute message transformations using either the command-line tools, or the Transformer API's (Application Programming Interfaces) in the Java language.

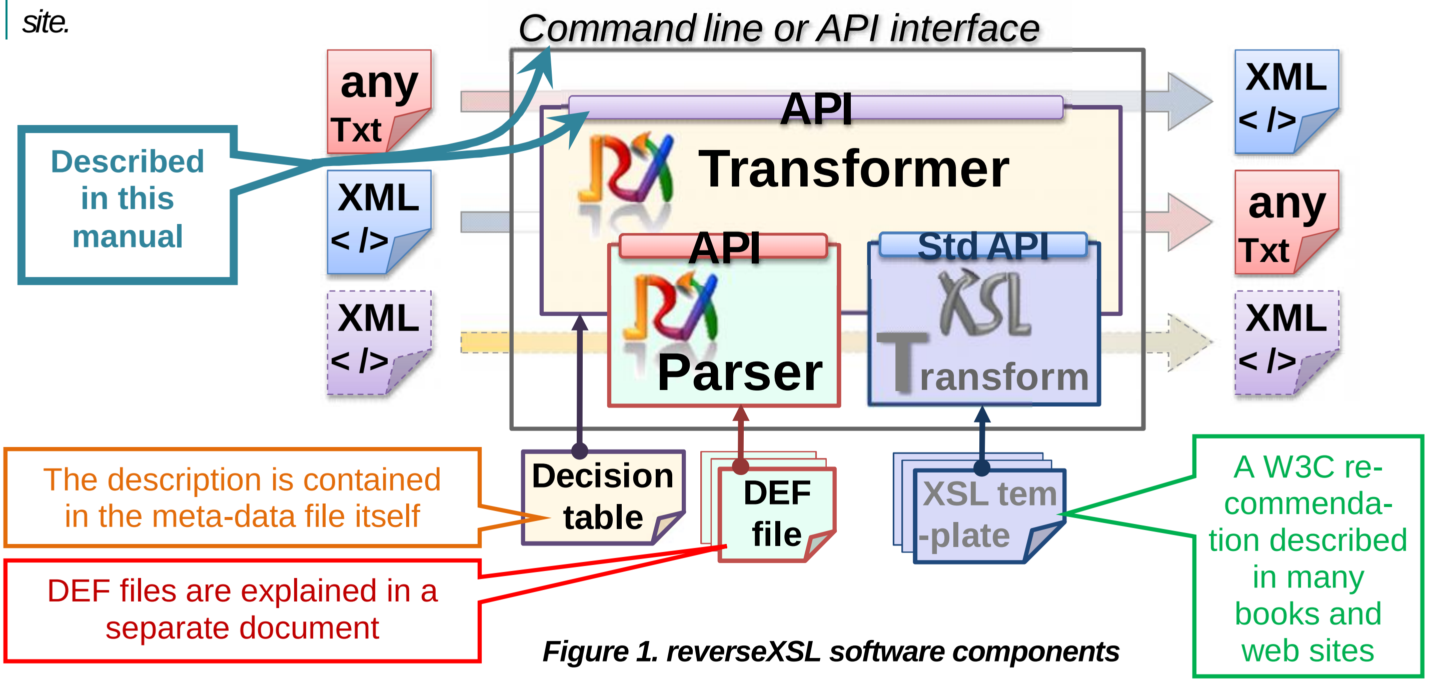
Command line or API interface
any Txt
XML < />
XML < />
Described in this manual
API
Transformer
API
Parser
Std API
Transform
XML < />
any Txt
XML < />
The description is contained in the meta-data file itself
DEF files are explained in a separate document
Decision table
DEF file
XSL tem-plate
A W3C recommendation described in many books and web sites

Figure 1. reverseXSL software components

As represented in the figure above, the reverseXSL software operations rely heavily on two components:

- A Parser component, which is an original piece of software;
- The Java API for XML Processing (JAXP) that notably contains XSL Transformations (XSLT).

  XSLT is a standard component of the Java Runtime environment. This component is therefore not duplicated in the ReverseXSL.jar.

The idea is simply that of augmenting the native XSLT transformation capabilities with the reverseXSL Parser capabilities, and bundle the whole as a new, extended, **Transformer**.

Your learning curriculum:

Evaluate

- If you just intend to play with the software, and try a few available samples, you will likely not need to study DEF files. The introduction to *Message DEFinition files* that is published on the web[1] site may be sufficient to grasp a good idea about the Parser component. You will anyhow encounter regular expressions within the tutorial examples. Let me insist, regular expressions are not intuitive; however, the apparent complexity is entirely contained in only 11 special characters: **.** ***** **+** **?** **(** **[** \ **{** **^** **$** and **|** —there's no more to study. You will save yourself from a lot of frustration, spare much time, and possibly begin to grasp their magical power by taking the time to browse the accelerated tutorial that we have prepared for that purpose (exclusively on the web site).

Develop

- If you intend to develop new transformations, the required knowledge goes as follows:
  - An understanding of the **message formats** that you have to process is indeed a must, whether these are application-specific, corporate conventions, or EDI standards.
  - Playing quickly with the **Transformer** and/or reading through this manual will help you plan your job (what functions the software readily supplies, how to integrate it, etc.).
  - Then you should learn about **regular expressions**, and study the **Message DEFinition** file format used by the Parser.
    Both topics are covered in the Message DEFinition Files documentation.
  - As you develop, it is quite efficient to immediately **test partial message DEFinitions**[2] against message samples. To do so, you will use the Parser via the command line interface 'Parse', and configure such commands to run **within your favourite development workbench**.
    Tips for automating the development cycle are published at the web site.
  - You may then need to perform an **XSL Transformation** next to the Parser Transformation step. Indeed, you will need to learn XSLT. You shall also consider acquiring an **XSLT editor** to increase productivity (free and pay-for solutions are available as plug-ins into your development workbench or as dedicated products). Again, you may want to test at the same time you develop the XSL template.
  - Last, you will edit the **Mapping Selection Table** in the reverseXSL Transformer, and create an entry that will match the new message against the required Parsing step and XSL Transformation step. This last operation may take just 10 minutes. You

The learning of XSLT and associated editors are covered in numerous web sites and books. We do not provide such information and software.

[1] http://www.reversexsl.com/j/index.php?option=com_content&task=view&id=27

[2] As soon as you have formalized a first segment with a few data elements you can start testing. This is a very productive facility.

will then configure a 'Transform' command to run **within your favourite development workbench** for a final check of the complete transformation.
Tips for automating the development cycle are published at the web site.

- At this point, you may consider building a JUnit test case over the transformer API in order to automate test series.
- Then it's time to **integrate the software** using the API or command line interface into your application. It may range from a simple Windows script or Excel Macro that triggers the transformer to the development of a dedicated adapter into an application server, or message broker, or EAI platform.
Tips for integrating the tool into various systems are published at the web site, as well as source code and scripts.

Later on, as you amend existing transformations or develop new ones, you will just need to deploy new and updated meta-data files into the target execution environment. This can take the form of a Java archive (.jar) containing all updated resources, else simply as files on disk, else as character streams retrieved from a database.

2. The Software and its Installation

Software delivery format

The software is delivered within a single Java archive file (.jar), itself contained within a zip file along with documentation and samples. The software itself is optimized and compacted.

The jar is organized as follows:

ReverseXSL.jar			
	/META-INF	MANIFEST.MF, etc. Standard Java archive meta data	
	/com		
		/reverseXSL	java class files of the original reverseXSL software. Some source code is also included for customisation.
	/resources	**meta-data** used by the reverseXSL software	
		/DEF	Messages DEFinition files used by the Parsing step.
		/TABLES	contains the Mapping Selection Table.
		/XSLT	XSL templates for the XSLT step.
	/reference	Schemas and specifications associated to the Transformation libraries included in resources (varies much with the software delivery shipment at stake)	
	/samples	Sample messages for test runs, copies of expected outputs	

System requirements

If you download the Java runtime from http://java.sun.com, the JSE (Standard Edition) bundle is sufficient.

A Java runtime environment (JRE) from version 1.4.2 or later (1.5, 1.6…) must be already installed in the target computer systems.

These versions contain regular expression libraries and JAXP (XSLT) libraries by construction, so nothing else particular is therefore required.

Installation

Simply unzip the software distribution archive (reverseXSL.zip) to a directory of your choice (extract all files and preserve the relative directory hierarchies) with a zip extractor (Winzip™, WinRAR™, or the one embedded in MS-Windows™).

UNIX/LINUX users will prefer the Gzip'ped tar archive format available for download from the web site.

Quick test runs

You can immediately execute the *Transform Sample1/2/3* batch files.

On Windows command line and UNIX shell you redirect the standard output and error streams by adding 1>out.txt and 2>err.log at the end of the command line.

The scrolling output mixes execution logs with the transformed message in proper. If you edit the batch file to redirect the standard output stream to a file, you capture the output message data. If you redirect the standard error stream to a file, you capture the logs.

The command line programs as supplied are just handy development tools. You will want to customise them (source code is inside the jar), or better, integrate the software into an application environment via the API (cfr javadoc).

Classpath issues

TIP: within the Java runtime directories, there is a lib/ext directory that is designed to contain runtime extensions. Copy the ReverseXSL.jar there and you have done.

To facilitate further works, the ReverseXSL.jar file can then be copied to a directory elsewhere on the Java CLASSPATH, or this CLASSPATH must be updated to include the full path name of the ReverseXSL.jar file.

Checking and editing the CLASSPATH is explained elsewhere in Java documentation. You may also supply a "-classpath" (or –cp ...") argument to every Java command line.

Software development workbenches like Eclipse™ or the NetBeans™ IDE do supply local facilities to manage CLASSPATH definitions per project; you may like (or need) to adjust such definitions for the execution of the reverseXSL software from inside the workbench.

If you decide to move the ReverseXSL.jar instead of copying it elsewhere, the example transformation batch files may no longer work, as these contain commands like:

```
java –jar ReverseXSL.jar myInputSampleFile
```

...that assume that the jar is in the local directory; but you can update them into:

```
java com.reverseXSL.Transform myInputSampleFile
```

The documentation

In the directory where you extracted the software, you will find the software manuals in PDF format and the JavaDoc of the API (double-click or open the file `./javadoc/index.html` with an Internet browser).

The web site www.reverseXSL.com supplies tutorial samples and an accelerated tutorial on 'regular expressions'.

License

The ReverseXSL parser is an Open Source software available under the *Apache 2.0* license.

Managing transformation resources

As shipped, the software archive does contain sample meta-data resources (that drive transformations) within the .jar file itself. You may add/replace by your own message definitions and XSL templates. Alternatively, you may like to manage such meta-data in separate .jar files, and separate it from the software in proper. In the later case, remove the /reference and /samples directories from the original jar file, and move the /resources to a separate jar file (or a regular directory) that you will place on the CLASSPATH. The reverseXSL software will perform identically.

The exact way the reverseXSL software looks up for meta-data resources is explained in the next section.

3. Operations

3.1 Transformer overview

With each new message that arrives, the reverseXSL Transformer executes from 1 up to 3 different activities (see the figure on page 3 and below):

1. A first step, using pattern recognition, matches a transformation profile to the arbitrary input message. A transformation profile contains zero, one, or both of the next two steps, plus optional message handling parameters.
2. The Parsing step is triggered (or not) according to the selected profile. It decodes any structured character-data message and produces an XML document.
3. The XSL Transformation step is triggered (or not) according to the selected profile. It converts XML documents into other XML documents, else flat file structures.

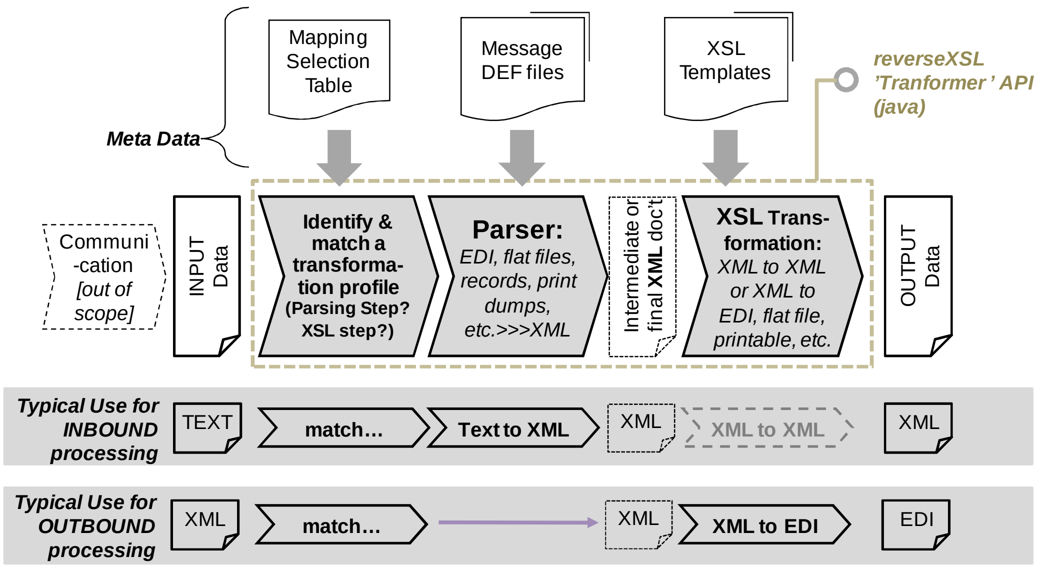
Mapping Selection Table
Message DEF files
XSL Templates
reverseXSL 'Tranformer' API (java)
Meta Data
Communi-cation [out of scope]
INPUT Data
Identify & match a transforma-tion profile (Parsing Step? XSL step?)
Parser: EDI, flat files, records, print dumps, etc.>>>XML
Intermediate or final XML doc't
XSL Trans-formation: XML to XML or XML to EDI, flat file, printable, etc.
OUTPUT Data
Typical Use for INBOUND processing
TEXT
match…
Text to XML
XML
XML to XML
XML
Typical Use for OUTBOUND processing
XML
match…
XML
XML to EDI
EDI

Figure 2. The flow of data

Bytes or characters?

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.

These byte streams are read from and written to disk files (case of command-line operations), else passed as input/output stream arguments to methods in the API.

Default input and output char set is UTF-8

The **byte** stream is decoded into a **character** stream according to a specified input character set. Similarly, the output character stream is returned encoded as a byte stream according to a specified output character set. The reason why the inputs/outputs are based on byte streams is fivefold:

- Preserving the ability to apply byte-oriented 'de-pollution' functions to the input message before undertaking the identification and transformation steps in proper
- Paving the way to software extensions in the area of envelope processing, where such envelopes can bear a byte-oriented design and/or imply specific transfer encodings on a byte-oriented payload.
- Keeping the possibility for functional extensions into zip/unzip, or encrypt/decrypt before/after transformation, and similar functions working on bytes and not characters.
- Being more flexible with regard to the actual source/target of message data: a buffer in memory, a sequential file on disk, a data stream, a blob in a database.
- And, keeping control over an area which is a frequent source of miss-handlings (taking bytes for characters, mixing multiple encodings, using java methods that perform implicit conversions, implementing char-oriented operations on byte arrays, still assuming 8-bit characters, and so on).

Restrictive-listening versus automated mapping selections

Numerous third party integration platforms provide all means to assemble and deploy sophisticated message processing chains, but each of these is often capable to process only one type of input message. If a different message enters through the exact same channel, it would likely be rejected as "bad format". For instance, reading message files from directories, different message types shall match different file name patterns so that each brand is picked up by the one process that can handle it. This ***restrictive-listening*** approach is even encouraged by the use of web services. Each web service operation has an associated message type. Each operation is handled as an event that triggers a specific process flow[3], thus handling the specific payload associated to the operation.

The reality, in particular in the context of electronic trading, is different: all message are likely to arrive, in **mixed-types**, through the same communication channel (MQ-series, SMTP or AS1, HTTP or AS2, ebMS, sectorial network connections like SWIFT or SITA, links to community systems, etc.). The support of such mixed-types flows often implies the purchase of additional third party 'EDI' modules such as to acquire the facility to sort out which input message is of what type and trigger an adequate process in each case.

The reverseXSL software can be integrated in both environments:

Restrictive-listening mode

A. A Transformation can be invoked that enforces a specific Parsing step, and/or a specific XSLT step. Whenever the input message does not match the syntax implied by either the Parsing DEFinition, else XSL template, exceptions arise.

Mixed-types mode

B. On the other hand, one may simply invoke 'Transform' and delegate to the software the selection of the one transformation profile that matches the actual input brand.

The selection mechanism is dreadfully simple but appallingly efficient: up to the first 1500 characters[4] of the input message are presented, in sequence, to an ordered set of mapping selection entries. Each entry contains a regular expression that is tentatively matched[5]. On the first match, the entry is selected and the associated Parsing step and/or XSL Transformation step are carried on.

Such Mapping Selection mechanism is suitable for small collections of messages up to a hundred entries; performance will degrade with larger sets. One must then either manage a few different Mapping Selection tables, each associated to a data exchange context, else use 'external'

[3] or enters a suspended process instance, selected via correlation properties. Again, a specific payload content is expected by the process that now resumes execution.

[4] Characters, not bytes, hence implying that the source character set has been used for decoding input bytes into characters. The value 1500 is a software build parameter. By experience, 1500 is large enough for all headers the authors have encountered during the last 20 years…

[5] Precisely, the pattern is searched within the first 1500 characters of each file. It shall match a chunk of these 1500 bytes for a selection of the corresponding mapping table entry.

means to determine a suitable profile and invoke the Transformer with the specific parsing DEF and XSL template in this profile.

Combining the Parser with XSLT: doing none, one, or both

In the case B above (mixed-types), one can even mix XML and non-XML inputs, and let the reverseXSL Transformer select the proper combination of Parsing and XSL transformation applicable to each case. On the output side, one can ensure a flow of neatly distinguished XML document types, or even the production of a 'universal' XML document envelope inside which the various payload types are wrapped.

Input (ALL TYPES MIXED)	RXSL Parsing?	XSLT?
non-XML	yes: any input syntax is transformed to a clean XML document	yes or no: as needed to reorder and/or combine elements in the message, else to match an imposed output schema
XML, whose schema is not acceptable by the target	no	yes: transform such as to match the imposed output schema
XML, already compliant to a supported target schema	no	no: as option, used for XML schema validation

In all cases, when the reverseXSL Transformer has finished with the transformation (in zero, one or two steps), the calling application may use the `myTransformer.getName()` method to retrieve the name associated to the mapping table entry that has been matched and executed. This name can then be used to branch to diverse onward processing flows. Unique names are not enforced in the Mapping Selection table.

3.2 Using the command line interface

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

We assume that the reverseXSL .jar file is available on the CLASSPATH, cfr 'Installation' and 'Classpath issues' from pg 6.

If not, you can use an explicit –cp argument as in:

```
java –cp ReverseXSL.jar com.reverseXSL.Transform
```

Command-line Transformer

`com.reverseXSL.Transform`

This tool is a command-line wrapping of the Transformer API.

Execute the command alone to get help, as follows:

```
java com.reverseXSL.Transform
```

and, for more options,

```
java com.reverseXSL.Transform HELP
```

as described below.

Beware that variant command line forms exhibit different policies for searching and loading meta-data resources.

1. Simplest use, with a single jar containing software and transformation meta-data (meta-data comprises Parsing DEFinitions, XSL templates, and the mapping selection table)

   Executed from the jar (does not require the jar to be on the classpath!):

   ```
   java -jar ReverseXSL.jar myInputDataFile
   ```

   Variant, executed from the CLASSPATH.

   ```
   java com.reverseXSL.Transform myInputDataFile
   ```

   In both above cases, meta-data resources are loaded from directories and/or multiple jar's using the CLASSPATH[6].

   Precisely, the software first tries to load the file named `resources/TABLES/mapping_selection_table.txt` else just `mapping_selection_table.txt` (this is a fixed, built-in, name) using the CLASSPATH, thus comprising the ReverseXSL.jar contents itself. When found, it proceeds to the mapping selection step in itself. If an entry matches the input message, then the pathnames specified within the selected mapping table entry are used to load the DEF and/or XSL resources, again using the CLASSPATH. If the mapping selection table is not found in the very first step above, an I/O exception is thrown.

   Both these command lines followed by `>MyOutputFile` do capture the transformed result to a file; indeed, output is written on standard output whereas execution logs are written on standard error.

The variant command line options and arguments are all features of the Transform command line application, itself wrapped around the Transformer API. The source code is supplied in the jar. So if you do not like the command line interface, create one that matches your style and requirements.

2. Advanced use: enforce a parsing DEF and XSL Transformation:

   ```
   java com.reverseXSL.Transform  myDEFFile myXSLFile myInputDataFile
   ```

   The Mapping Selection Table is ignored. The parser DEF file and the XSL template file instruct to, respectively, parse the input with the associated Parsing DEFinition, and then transform the resulting XML with the specified XSL template. If any of those two resources are not found relative to the current working directory[7], the corresponding transformation step is omitted (and indicated in logs).

   Tip: simply use placeholder DEF or XSL arguments like 'NoDEF' and 'NoXSL' to skip the corresponding transformation step.

   The same advanced command with all arguments looks like:

   ```
   java com.reverseXSL.Transform myDefinitionFile myXSLFile
       myInputDataFile [<InputCleansing> <MaxFatalExceptions>
       <MaxExceptions> [<true|false> [<indent>] ] ]
   ```

   where optional command-line arguments are noted within '[' ']', possibly nested.

[6] Classpath order defines the loading precedence.

[7] The Classpath is therefore not used for loading meta-data resources.

- The <InputCleansing> is one of `NONE`, `ToCRLF`, `ToLF`, `ToUPPER`, `FullyTrimmed`, or `UnfoldPSCRMRemarks`. You may combine multiple cleansing options with a '+' as in `ToCRLF+fullytrimmed+TOUPPER`. Tokens are not case sensitive. `FullyTrimmed` removes leading & trailing spaces/tabs, as well as empty lines and control characters (more options via the java API). `UnfoldPSCRMRemarks` removes the arbitrary '`<CR><LF>.RN/`' remarks-extension-line-breaks that can jeopardize the correct parsing of Remarks elements in IATA PSCRM messages (a legacy feature inherited from TELEX maximum line length at 69 chars).
- `<MaxExceptions>` & `<MaxFatalExceptions>` are integers that define the Parser exception recovery thresholds. Cfr the manual about *Message DEFinition Files*, else the javadoc.
- The `<true/false>` argument is one of `true` or `false` (ignoring case) and tells to *removeNonRepeatableNilOptionalElements*. Please refer to the manual about *Message DEFinition Files* for an explanation of this option.
- The `<indent>` string instructs to produce printable transformed outputs with the given indentation pattern. Try " |" and you'll see the effect!

This one variant is quite useful to automate testing while developing new transformations because you can take advantage from the directory hierarchies in the workspace in order to manage global (consolidated) and local (specific to one or a few messages) mapping selection tables.

3. Advanced use for Test automation:

```
java  com.reverseXSL.Transform  AUTO  SELECT  myInputDataFile
    [  <InputCleansing>  <MaxFatalExceptions>  <MaxExceptions>
    [<true|false> [<indent>] ] ]
```

In which case a file named `mapping_selection_table.txt` (fixed name) is searched up in the directory hierarchy[8] such as to dynamically resolve which parsing DEF and/or XSL template to apply to the given input data.

> *Precisely, the Classpath is not used to load the mapping selection table. When found, it proceeds to the mapping selection step in itself. If an entry matches the input message, then the pathnames specified within the selected mapping table entry are used to load the DEF and/or XSL resources by first trying to load them from the Classpath and second, trying to load them relative to the directory in which the mapping selection table file was found.*

Note that the output is always written on stdout whereas execution messages and logs are written on stderr; the transformed output is thus captured using command-line output redirection. For instance:

```
java -jar ReverseXSL.jar myInputDataFile >myOutputFile
```

[8] starting from the current working directory, which you can configure to any directory of choice when executed from development workbenches

In addition, when there are Parsing errors, a report about each error is added to execution logs generated by the Transformer.

Command-line Parser

`com.reverseXSL.Parse`

This tool is a command-line wrapping of the **Parser** API. The Parser is a subset of the Transformer. Parsing comes second in the set of 3 operations executed by the Transformer: a mapping decision, an optional Parsing step, an optional XSL Transformation step.

The command-line Parser is very convenient during the development of new non-XML message DEFinitions. Indeed, one can get a detailed dump of the Parser state next to both successful and non-successful parsing. The dump supplies quite useful information about all the pieces that were actually identified in the input message, and how they were cut, and how the conditions were fed during the parsing.

Execute the command alone to get help, as follows:

```
java com.reverseXSL.Parse
```

We assume, in this command and the following that you have placed the ReverseXSL.jar on the CLASSPATH, else use an explicit –cp argument as in:

```
java –cp ReverseXSL.jar com.reverseXSL.Parse
```

1. Simple use

If you get a No Class Def Found error, add the classpath argument to the java command line with the complete path to the ReverseXSL.jar file.

Executed with the ReverseXSL.jar on the CLASSPATH:

```
java com.reverseXSL.Parse myDefinitionFile myInputDataFile
```

Such command line outputs both the Parser state dump and the generated XML document. It is a good idea to follow it by `1>MyOutputFile` and/or `2>MyLogFile` in order to capture the transformed result to a file, or the logs to a file, respectively; indeed, output is written on the standard output stream whereas execution logs are written on the standard error stream.

For instance, if you open the ReverseXSL.jar file itself with a zip extractor and extract the sample DEF file FWB15.def, you can execute:

```
java com.reverseXSL.Parse FWB15.def Sample2_TypeB_FWB.txt 2>logs.txt
```

2. Test automation

Executed with the ReverseXSL.jar on the CLASSPATH:

```
java com.reverseXSL.Parse myDefinitionFile myToken
```

Where myToken is a keyword found in the DEFinition file itself that indirectly supplies the file name, alike in those lines:

```
  #ONE=TypeB_FWB_01.txt;
```

```
#TWO=TypeB_FWB_02b.txt;
```

Defining two possible tokens: “ONE” and “TWO”.

This facility allows keeping unit-test file names within the DEF files themselves, while at the same time tokens may bear the same series of names across all message DEFinitions, which is very convenient in automating regression tests.

3. Additional options

Executed with the ReverseXSL.jar on the CLASSPATH:

```
Java   com.reverseXSL.Parse   myDefinitionFile   myInputDataFile
          <MaxFatalExceptions> <MaxExceptions> <true|false>
```

This command line features all possible arguments, namely:

These three Parsing options are explained in the documentation on Message Definition files.

`MaxFatalExceptions` is a number that fixes the tolerance of the Parser to fatal exceptions.

`MaxExceptions` is a number that fixes the tolerance tolerance of the Parser to the overall count of exceptions (fatal and warnings).

`true|false` indicates whether the Parser shall remove non-repeating nil optional elements from the XML output or not (default is false).

4. Generate a sample output XML document

Assuming the ReverseXSL.jar on the CLASSPATH:

```
Java com.reverseXSL.Parse myDefinitionFile
```

This command will also generate the schema of the output XML in a forthcoming release.

When executed with just a DEF file argument, the Parser generates a dummy sample message with all data element descriptions as data values. This is a convenient means of documenting the generated XML output.

3.3 Using the Transformer API in Java

The use of the Transformer is extremely simple.

Please refer to the documentation of all methods of the TransformerFactory and Transformer classes in the JavaDoc (included in the software distribution jar).

It is best illustrated through the following sample code. It takes only 3 effective lines of Java code to run a transformation.

```
//references to TransformerFactory and Transformer classes
// in the sample code below are NOT from the package
//javax.xml.transform but from com.reverseXSL.transform !
import com.reverseXSL.transform.Transformer;
import com.reverseXSL.transform.TransformerFactory;
...
try {
   //you must catch exceptions, as any failure to transform
   //the input message results in an exception. No exception
   //means OK.

   //1. Get a default transformer factory with:
   TransformerFactory tf = TransformerFactory.newInstance();
       //That one will use all meta-data resources embedded
```

```
        //in the main jar file, else on the classpath.
        //Alternative factory methods support variant
        //sources of meta-data.

    //2. Instantiate a transformer:
    Transformer t = tf.newTransformer();

    //Use an Input Stream and an Output Stream of whatever
    //type for your input message and generated output:
    //Byte Array, File or else
    ByteArrayInputStream inStream = new
                  ByteArrayInputStream(myInputMessage);
    ByteArrayOutputStream outStream =
                  new ByteArrayOutputStream();

    //3. Execute the transformation
    int parserWarnings = t.transform(inStream,outStream);
    //DONE! just take the transformed message from the output
    //stream. For instance, to dump it out:
    outStream.close();
    System.out.println(outStream.toString());

    //The parser, when invoked in the transformation, is able
    // to tolerate minor message syntax deviations (this is
    // entirely parametrizable) up to given thresholds (set
    // via the factory API). The integer that is returned
    // tells how many syntax violations were actually
    // detected with a count still below thresholds: by
    // default, 10 warnings are accepted, and zero major
    // error.

    //You may be interested in execution logs:
    StringBuffer sb = t.getLog();
    //and, to iterate over all recorded Parser exceptions,
    ListIterator li = t.getParserExceptionListIterator();

} catch (Exception e) {
    //put your exception handling code here.
    //The exceptions caught here relate to:
    //  -  an exceeding count of Parser exceptions (cfr
    //     Parser tolerance and exception thresholds)
    // -   I/O Exceptions
    // -   XSL Transformation exceptions
}
```

* * *