

**Project IST-FP6-026476 SEAMLESS**  
**“Small Enterprises Accessing the Electronic Market of the**  
**Enlarged Europe by a Smart Service Infrastructure”**  
**STREP – Information Society Technologies (IST)**

**Deliverable D2.2.2**  
**Semantic Translation**  
**Technical Specification**

**Workpackage WP2 – Knowledge and Languages**  
**Task T2.2 – Semantic translator**

**Abstract**

*This document presents and describes the software module, the Semantic Translator, resulting from the design and development work carried out in task T2.2 (the software module is temporarily released as deliverable D.2.2.1). This software module realises the translation functionality that can convert a query or a business document (or any other datum) from one ontology to another by using the relative transformation file generated by the ontology Mapper described in deliverable D2.1.2.*

*The Semantic Translator takes the form of a web service in such a way that it can be invoked from other applications submitting the query or document to transform, and receiving back the transformed version. In order to perform the transformation the Semantic Translator applies the instructions of the transformation file to convert the data structure and translate the contents (terms) of the input document.*

*The technical specification reported in this document recalls the context where the Semantic Translator will be used, an analysis of the transformations the Translator is asked to provide, a survey on existing similar tools, a description of the chosen development environment, and finally a detailed description of the software module and its WS interface.*

<b>Start date of project</b>	Jan 1 <sup>st</sup> , 2006	<b>Duration of project</b>	30 months
<b>Deliverable due date</b>	Jan 21 <sup>st</sup> , 2007	<b>Actual submission date</b>	Jan 29 <sup>th</sup> , 2006
<b>Dissemination level</b>	PU	<b>Revision status</b>	Final
<b>Responsible partner</b>	KELYAN	<b>Authors</b>	T2.2 participants

**Change record**

<b>Rev. N.</b>	<b>Description</b>	<b>Author</b>	<b>Date</b>	<b>Review</b>
0	Early draft	M. Barbi (KELYAN) S. Dondi (U MODENA)	Dec 15, 2006	F. Bonfatti (U MODENA)
1	Full draft	M. Barbi (KELYAN) S. Dondi (U MODENA)	Jan 13, 2007	J.V. Vidagany (ANTARA) M. Borrás (ANTARA)
2	Final version	M. Barbi (KELYAN) S. Dondi (U MODENA)	Jan 26, 2007	F. Bonfatti (U MODENA)

## Table of contents

1	EXECUTIVE SUMMARY .....	4
2	REQUIREMENTS ON FUNCTIONALITY .....	6
2.1	SEAMLESS ontologies.....	6
2.2	Queries and documents to translate.....	8
2.2.1	<i>LOCL to COMM transformations</i> .....	8
2.2.2	<i>COMM to GLOB transformations</i> .....	9
2.2.3	<i>GLOB to GLOB transformations</i> .....	9
2.2.4	<i>GLOB to COMM transformations</i> .....	9
2.2.5	<i>COMM to LOCL transformations</i> .....	10
2.3	Requirements and issues .....	10
2.3.1	<i>Assumptions and definitions</i> .....	10
2.3.2	<i>Main issues addressed</i> .....	11
2.3.3	<i>Transformation file structure (recalled)</i> .....	12
2.3.4	<i>Basic mapping operations</i> .....	12
3	ANALYSIS OF ALTERNATIVE TOOLS .....	18
3.1	XSLT recalled .....	18
3.1.1	<i>XSLT and the Semantic Translator</i> .....	19
3.2	State-of-the-art .....	19
3.2.1	<i>MapForce 2007 (<a href="http://www.altova.com/products/mapforce">http://www.altova.com/products/mapforce</a>)</i> .....	20
3.2.2	<i>Delta 4.0 (<a href="http://www.softshare.com">http://www.softshare.com</a>)</i> .....	21
3.2.3	<i>Stylus Studio (<a href="http://www.stylusstudio.com">http://www.stylusstudio.com</a>)</i> .....	22
3.2.4	<i>Comparison and conclusion</i> .....	23
4	SEMANTIC TRANSLATOR MODULE.....	25
4.1	Document transformation .....	25
4.1.1	<i>Structural translation</i> .....	25
4.1.2	<i>Vocabulary translation</i> .....	32
4.2	Query transformation .....	33
4.2.1	<i>SEAMLESS requirements on XQuery structure</i> .....	34
4.2.2	<i>Query translation solution</i> .....	35
4.3	Translator architecture design and implementation .....	37
4.3.1	<i>Major design decisions</i> .....	37
4.3.2	<i>High level architecture</i> .....	37
4.3.3	<i>Semantic Translator core – converter component</i> .....	39
4.3.4	<i>Semantic Translator core – query component</i> .....	39
4.3.5	<i>Semantic Translator core – util component</i> .....	40
4.3.6	<i>Semantic Translator core – model component</i> .....	40
4.4	Translator WS interface .....	41
5	FINAL REMARKS .....	45
	APPENDIX: XSLT TRANSFORMATION FUNCTIONS .....	47

# 1 Executive Summary

This document presents the results of the work carried out in workpackage WP2 “Knowledge and Languages” and, specifically, in task T2.2 “Semantic translator”. In practice, it describes the functionality and provides the technical specifications of the software module, the SEAMLESS Semantic Translator, that is delivered simultaneously as deliverable D2.2.1 “Semantic translation tool”.

The Semantic Translator module has been developed starting from the analysis carried out in task T3.1 and documented in deliverable D3.1 “Overall Architecture Design”. It is intended to behave as a web service to be invoked for converting a query or a business document from its source ontology to a given destination ontology by executing the transformation instructions contained in the relative transformation file generated by the Ontology Mapper.

Then, the Semantic Translator functionality is strongly conditioned by those of the Ontology Editor and Ontology Mapper made available as deliverable D2.1.1 and documented in deliverable D2.1.2, both coming from the execution of task T2.1. In particular, the three software modules have been conceived together with the aim, on the one side, to minimise the ontology expert effort when using the Ontology Editor and the Ontology Mapper and, on the other side, to realise an effective unified translation service by the Semantic Translator for queries, business documents and all other possible data to be exchanged between partners.

This document is subdivided into three main chapters and an appendix:

- Requirements on functionality. The chapter recalls the role that SEAMLESS assigns to the Semantic Translator service when moving from one ontology to another in the ontology hierarchy. In particular, it summarises the queries, documents and other data types that are expected to be transformed, recalls the transformation file structure taken from the deliverable D2.1.2 and derives the spectrum of the elementary (basic) transformations the Semantic Translator is required to perform.
- Analysis of alternative tools. The chapter addresses the state-of-the-art in the field by analysing the existing ontology-based transformation tools, with special attention to the well-known Altova MapForce, and measuring their ability to meet the SEAMLESS requirements. The decision to realise a new Semantic Translator is accompanied by a description of the XSLT language adopted as development environment.
- Semantic Translator module. The chapter provides the technical specification of the Semantic Translator, the web service in charge of performing the transformation of queries and documents from their source ontology to a given destination ontology. The Translator is general enough to transform queries, business documents and all other data types to be exchanged between partners, and to move from each of the SEAMLESS ontologies to the previous or the next one in the ontology hierarchy.
- Appendix: XSLT transformation functions. The appendix collects and documents the stylesheets written in XSLT language to implement the basic transformation functions used by the SEAMLESS Semantic Translator.

The document is public. The intended audience includes the following categories of possible readers:

- Technical partners in the SEAMLESS consortium. Deliverable D2.2.1 provides a software module whose functionality is of great interest for the development work to be carried out in task T3.3 “Distributed storage system” and workpackage WP4 “Applications and Services”. This deliverable assures the required technical documentation on such software module and the data structure it uses.
- The other projects of the DE cluster. The ontology-based functions realised by the SEAMLESS project, including the Semantic Translator, cover and overcome critical issues that are common to other projects in the DE cluster. The technical documentation provided by this deliverable is necessary and sufficient to fully share with them the adopted approach and solution.

- Academia and research institutions. The SEAMLESS ontology and multilingual support tools are a practical implementation of ideas and concepts coming from the research world on semantics. As such they represent a concrete opportunity of validation with respect to all the difficulties coming from adapting the theoretical approach to a hard on-field use.
- Possible followers. Finally, the deliverable is a way to publish the description of an important and critical component of the SEAMLESS platform that can enable third parties (e.g. software houses, service providers) to undertake parallel development initiatives.

## 2 Requirements on functionality

This chapter recalls the SEAMLESS ontologies structure along with the role that SEAMLESS assigns to the Semantic Translator service when moving from one ontology to another in the ontology hierarchy. In particular, it summarises the queries, documents and other data types that are expected to be transformed, and derives the spectrum of the elementary (basic) transformations the Semantic Translator is required to perform.

### 2.1 SEAMLESS ontologies

The SEAMLESS project is mainly addressed at providing a set of services and applications allowing companies to share and exchange with each other information regardless languages and data structures. This is done stepwise:

- If the company has its own information system, likely very simple and realised by a small local software house, the first step is supporting the possibility to export (import) data and business documents from (to) such system to (from) the mediator platform <sup>1</sup>. In this way data and business document overcome the limits of the local data model (we call it LOCL) and are accessible to the SEAMLESS collaboration environment through the common ontology adopted by the specific mediator (we call it COMM).
- If the company has no electronic information system, e.g. because it accounts on paper or uses the bookkeeping service of its mediator, SEAMLESS puts it in condition to work directly on the mediator platform and then start using its support to collaborate with partners. In this case it is forced to adopt the COMM ontology which, therefore, must be easily understood and accepted by the user company <sup>2</sup>. In particular the COMM must be expressed in the local language and contain the set of concepts the company is used to employ in its business transactions (common concepts on sectoral/regional basis).
- The collaboration condition provided by the mediator platform is perfect if both the partner companies are associated to one specific mediator, and still comfortable if they are associated to two mediators sharing the same language and most concepts and terms (that is, with overlapping COMMs). However, hard problems arise as soon as the two COMMs are based on different languages. The solution to force all user companies to use English is not affordable by most of them and, in any case, the number of combinations of pairs of languages increases with the square of the number of languages and requires an unacceptable translation effort.
- Then a upper level ontology, based on English as *lingua franca*, is required to provide the bridge<sup>3</sup> between different COMMs (we call GLOB this global ontology). In principle, every COMM is defined by deriving its concepts from the reference GLOB and translating its terms into the local language. Of course, the user company is unaware of this complex process since it keeps generating, sending and receiving data and business documents according to the COMM ontology of its own mediator.
- The SEAMLESS model accepts the existence of two or more GLOBs <sup>4</sup>, e.g. generic as well as sector-specific global ontologies that qualified organisation can decide to offer to the candidate mediators. In order to assure the required semantic roaming to the data and business documents generated by the user companies, proper mapping (cross-reference) functions must be used to relate homologous concepts and terms in the different GLOBs <sup>5</sup>. Once again, the complexity of this architecture remains hidden to the user company by the easy entry-level applications and services of the mediator platform it is put in condition to use.

---

<sup>1</sup> See requirements UC17, UC18, UC43, UC44 of deliverable D3.1.

<sup>2</sup> See requirements UC13, UC14 of deliverable D3.1.

<sup>3</sup> See requirement UC16 of deliverable D3.1.

<sup>4</sup> See requirements UC08, UC09 of deliverable D3.1.

<sup>5</sup> See requirements UC11, UC12 of deliverable D3.1.

The consequence of these considerations defines the SEAMLESS ontology hierarchy based upon the following relationships:

- LOCL-COMM annotation. The LOCL data model represents the only concepts (tables, fields) that are strictly necessary to exchange data and documents with partners. They are taken from an existing information systems and interpreted according to the concepts and terms of the mediator COMM ontology. Indeed, this mapping operation is a sort of annotation of the LOCL concepts with those available in the COMM<sup>6</sup>.
- COMM-GLOB derivation. The COMM ontology is the semantic basis of all the applications and services the mediator offers to its associated companies. For the sake of cost-effectiveness and homogeneity, a COMM is normally derived from the reference GLOB by selecting the GLOB concepts and terms that are of interest for that regional/sectoral SEAMLESS node and translating them into the local language<sup>7</sup>.
- GLOB-GLOB cross-reference. GLOBs are created independently of each other by qualified organisations aimed at defining reference data models, standard taxonomies and controlled vocabularies. Mapping concepts and terms of two different global ontologies is a joint task of the GLOB holders that want to assure the best possible cross-reference for semantic roaming between them<sup>8</sup>.

In synthesis, the SEAMLESS model devises the existence of few (units) global ontologies, many (tens, hundreds of) common ontologies for every GLOB, and many (hundreds, thousands of) local ontologies for every COMM.

Going deeper into the ontology structure, in general a SEAMLESS ontology is made of three main components, namely a data model, a taxonomy and a controlled vocabulary:

- Data model. A relevant part of the shared and exchanged knowledge deals with company profiles and business documents. In other words, there is a core of structured information representing company properties, demand and offer of products and services, processes, bids and orders, invoices and the like, which can be conveniently modelled in form of a class diagram where, of course, every concept (a class, an attribute, a relation) is described by a proper explanation. This data model is a fundamental component of the SEAMLESS ontology.
- Taxonomy. Some of the attributes of the data model are intended to classify products and services as well as types of materials, technologies and other aspects. This knowledge is normally given as a taxonomy, i.e., arranged as a hierarchy of terms (concepts). Examples of general standard taxonomies are NACE, UNSPSC and ECLASS, examples of sectoral taxonomies are BUSCATEx (TEX) and *BcBuildingDefinitions* (B&C). While these taxonomies include thousands of terms, the experience shows that hierarchies of (few) hundreds of terms, i.e. the first 3 or 4 levels, are sufficient to represent a company and its production (just for qualification) and start negotiations with potential partners.
- Vocabulary. In addition to the taxonomy concepts, other terms are often useful to specify the features of a certain product or service, its supply conditions or the payment means. Although significantly depending on the single context, it is worth including them into the ontology as a vocabulary. Unfortunately, widely known lexicons such as WorldNet are not really useful as they are huge and generic at the same time. The best solution is a controlled vocabulary where users can find already defined concepts or add new concepts (or new definitions) whenever necessary. No strict vocabulary controls are indeed required, just periodical checks can be carried out to remove signalled errors or contradictions.

---

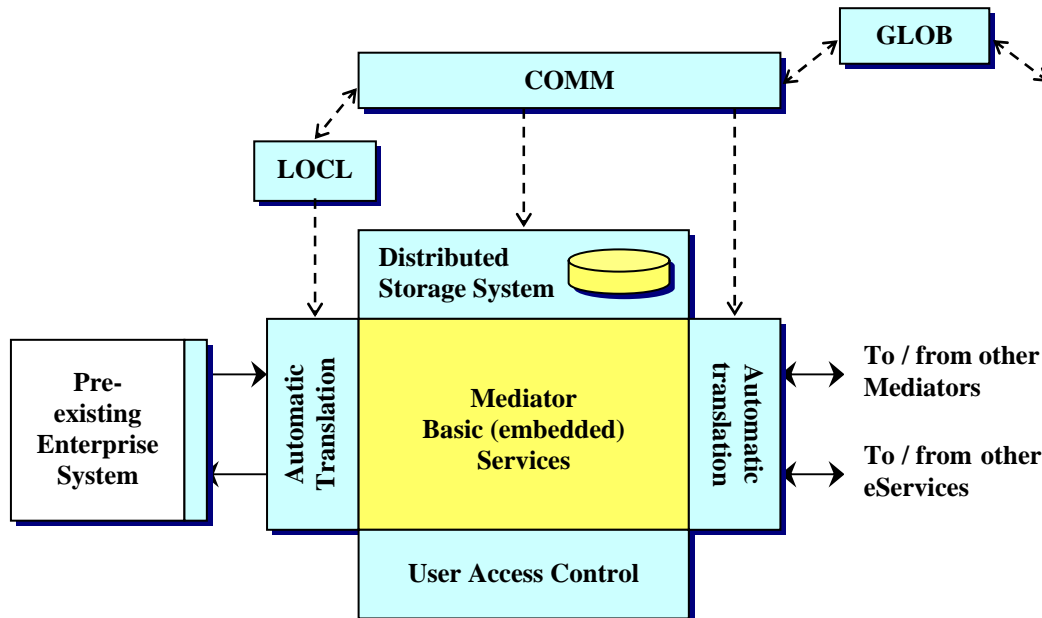
<sup>6</sup> See requirements UC17, UC18 of deliverable D3.1.

<sup>7</sup> See requirement UC16 of deliverable D3.1.

<sup>8</sup> See requirements UC11, UC11 of deliverable D3.1.

## 2.2 Queries and documents to translate

As explained in deliverable D3.1 “Overall architecture design” the SEAMLESS technological framework is realised by providing every node (mediator) with an ICT infrastructure whose general schema is depicted in this figure.



The picture emphasises two main translation phases, the one between the mediator COMM ontology and the LOCL ontologies of the enterprise systems (if any) of the user companies, and the other between the mediator COMM and the COMMs of other mediators. The former deals with moving documents and data from the company information system to the mediator platform so as to make them accessible by the SEAMLESS network (and vice versa). The latter deals with issuing queries that are forwarded to the SEAMLESS network nodes or exchange documents and data with their remote partners.

This introduces the different transformations that the Semantic Translator is required to perform under invocation from the interested SEAMLESS applications. In this section we give an overview of such transformations depending on the involved ontologies and pointing out the nature of transformed item (query, business document, other data type).

### 2.2.1 LOCL to COMM transformations

The devised transformations from the data structure of the company information system to the mediator common ontology include the following cases<sup>9</sup>:

- Export a business document. A business document is moved from the company information system to the mediator platform, e.g. as first step for being sent to a remote partner by the SEAMLESS network, then it must undergo an L2C transformation.
- Export a company profile. The profile of a partner, for instance a supplier or a customer, is exported to the mediator platform (e.g. as first step to check if it is a user of the SEAMLESS network), then it must undergo an L2C transformation.

<sup>9</sup> See requirement UC20 of deliverable D3.1.

## 2.2.2 COMM to GLOB transformations

The devised transformations from the mediator common ontology to the reference global ontology include the following cases <sup>10</sup>:

- Search for “near” partners. The query is prepared by a mediator user on the basis of its COMM, then the selection predicate refers to data model attributes and taxonomy/vocabulary terms of that ontology. The query can be executed as it is to search in the company registry of that mediator. Instead, if it has to be extended to other nodes under the same GLOB it must undergo a C2G transformation.
- Send a business document. The business document is available at the mediator platform, no matter if imported from a company information system or generated by a mediator application, then it is coded according to the relative COMM. If it has to be sent to a partner associated to another mediator (i.e. under another COMM) it must first undergo a C2G transformation.
- Return the query result. When the query is executed on the company registry of this mediator, the retrieved data (e.g. company profiles) are available in the form of its COMM. If the query was issued by a user associated to another mediator, the retrieved data must first undergo a C2G transformation.

## 2.2.3 GLOB to GLOB transformations

The devised transformations from one global ontology to another global ontology include the following cases <sup>11</sup>:

- Search for “far” partners. From the previous sub-section we know that the query has already been translated into the form of the reference GLOB. In order to be forwarded to the registries of mediators acting under another GLOB it must first undergo a G2G transformation.
- Send a business document. From the previous sub-section we know that the business document has already been translated into the form of the reference GLOB. If the addressee company is associated to a mediator acting under another GLOB, the business document must undergo a G2G transformation.
- Return the query result. From the previous sub-section we know that the retrieved data have already been translated into the form of the reference GLOB. If the inquiring company is associated to a mediator acting under another GLOB, the query result must undergo a further G2G transformation.

## 2.2.4 GLOB to COMM transformations

The devised transformations from the global ontology to the common ontology of a subscribing mediator include the following cases <sup>12</sup>:

- Search for partners. The query is available in GLOB form thanks to the (one or two) transformations mentioned above. In order to be executed on the company registry of each of the subscribing mediators the query must undergo a final G2C transformation.
- Send a business document. The business document is available in GLOB form thanks to the (one or two) transformations mentioned above. Since the addressee company is associated to a mediator acting under that GLOB, that business document must undergo a final G2C transformation.
- Return the query result. The retrieved data are now available in GLOB form thanks to the (one or two) transformations mentioned above. Since the inquiring company is associated to a mediator acting under that GLOB, that business document must undergo a final G2C transformation.

---

<sup>10</sup> See requirement UC22 of deliverable D3.1.

<sup>11</sup> See requirement UC24 of deliverable D3.1.

<sup>12</sup> See requirement UC22 of deliverable D3.1.

### 2.2.5 COMM to LOCL transformations

The devised transformations from the data structure of the company information system to the mediator common ontology include the following cases <sup>13</sup>:

- Import a business document. A business document is moved from the mediator platform to the company information system, e.g. to archive it together with those not generated by a mediator application, then it must undergo a C2L transformation.
- Import a company profile. The profile of a partner found through the partner search application is imported into the company information system (e.g. to archive it together with the other suppliers or customers), then it must undergo a C2L transformation

## 2.3 Requirements and issues

The translation process and the design of the involved modules is based on specific requirements since it is asked to address different issues. This chapter provides an analysis on the functionalities requested by the SEAMLESS translator as well as the related definitions and explanations.

### 2.3.1 Assumptions and definitions

The elementary (basic) transformation cases the Semantic Translator is required to consider can be identified according to the following assumptions:

- The document or the query predicate to transform is provided in form of XML file and the result of the transformation process is in turn an XML file.
- In order to assure the above mentioned transformations it is not necessary to study all the possible mapping situations on a theoretical basis, but consider just those practically occurring in “normal” business documents and “simple” queries.

The following definitions will be used in the rest of the document:

- Source ontology. The ontology edited by means of the Editor module to which the Semantic Translator input belongs.
- Destination ontology. The ontology edited by means of the Editor module to which the Semantic Translator output belongs.
- Tree node. It is the general structure of an XML-based document, no matter if subject to or resulting from the transformation process.
- Resulting tree node. The tree node computed by the Semantic Translator by applying the structural and linguistic translations to the source tree node. This operation is automatically driven by the mapping information produced by means of the Mapper module. It is important to point out how the translation process does not actually modify the source tree node but produces a new structure.
- Attribute, element, tag, text. Terminology brought in by the use of the XML language.
- XPath expression. A text based syntax for pointing tree nodes within a XML file.
- Structural mapping. The complete set of instructions to transform from the XML schema of the source ontology and the XML schema of the destination ontology.
- Vocabulary mapping. The complete set of instructions to transform from the vocabulary of the source ontology and the vocabulary of the destination ontology.
- Taxonomy mapping. The complete set of instructions to transform from the taxonomy of the source ontology and the taxonomy of the destination ontology.

---

<sup>13</sup> See requirement UC20 of deliverable D3.1.

### 2.3.2 Main issues addressed

The main issues addressed throughout the Semantic Translator development are:

- Tree structure conversion. Taking into account that the source ontology and the destination ontology could sensibly differ in terms of tree structure, the Semantic Translator has to interpret the structural mapping in order to take the proper contents from the source tree, process them, copy the result of this computation in the destination tree by applying the tree node schema.
- Linguistic conversion. The Semantic Translator is in charge of performing a linguistic translation of the content, according to the provided dictionary. The dictionary provides a mapping between source ontology terms and destination ontology terms such that a source term is always related to one and only one destination term. This design choice has been introduced thus removing any possible ambiguity throughout the translation process. The same rationale is applied both to vocabulary terms and taxonomy terms, where the latter have to be considered as a path-like list of the former. In line of principle, this assumption could be considered too limiting since it does not allow one-to-many mappings, e.g. English “glass” to Italian “vetro” (window glass) and “bicchiere” (glass for drinking). The SEAMLESS ontology design overcomes this situation by always considering a term as a composition of two parts: the term value and the term definition. Therefore, according to the previous example, the English ontology should define two different terms, “glass :: window glass” and “glass :: glass for drinking” and, then, the one-to-one requirement does not limit the mapping against the Italian ontology.
- Data types. The concepts defined by the SEAMLESS Editor module are typed according to the capabilities provided by the XSD syntax. Since the applicable data types follow a fine grained definition, it could happen that the same concept is represented in the source ontology by a data type different from the destination ontology one (e.g. issue date represented as date vs. date-time). XML represents all the information in a text string-like manner, then on a strictly formal point of view the Semantic Translator should perform a data type conversion in order to ensure that the yielded string conforms to the destination data type representation. This feature is not considered as major requirement because, on the semantic viewpoint, for a human reader a different representation format of the same concept often does not take to particular problems. Hence, the current version of the Semantic Translator does not supply such a behaviour that will be included in the next release. This will provide a more specific management of this issue by supporting data type format conversion functions like date-to-datetime or decimal-to-integer.
- Multiplicity. The Editor module, together with the underlying XSD, let the user associate a multiplicity attribute, both minimum and maximum, to a concept. On the other end, the actual multiplicity of a concept in an input document cannot be foreseen (excluding when minimum and maximum multiplicity are the same value). This situation leads to the following consideration. If a source concept is one-to-one mapped to a destination concept, any instance of the source concept is translated into the destination one. Since the two ontologies could differ in terms of multiplicity definition, it could be possible that the resulting document breaks the destination ontology multiplicity contracts. In detail, the problem can occur anytime the source ontology maximum cardinality for a precise concept is greater than the maximum destination cardinality for the same mapped concept. In order to respect the destination cardinality, the Semantic Translator should choose a subset of the source concepts and copy only this subset. Since this is considered a simple formal aspect rather than a real semantic issue, the implementation of such a feature is deferred to the next release of the Semantic Translator.
- Constant output values. A peculiar mapping situation occurs when the destination ontology requires a concept not defined by the source ontology since always represented by a constant, then implicit, value (e.g. currency). Hence, both the Mapper module and the Semantic Translator MODULE have to provide a facility by which the Mapper users associate a constant value to a specific set destination tree nodes. On the translation point of view, this means that those destination tree nodes have to be associated to provided constant values.

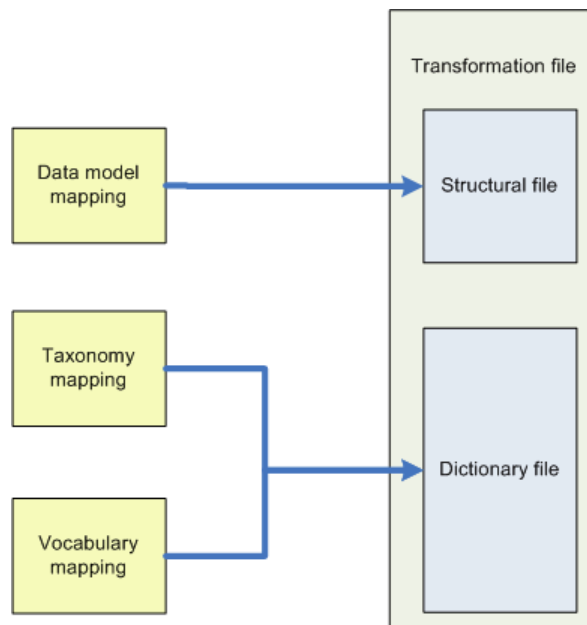
### 2.3.3 Transformation file structure (recalled)

Let us recall the main aspects of the transformation file, intended as the set of instructions the Translator must apply to the input document or query, as they result from deliverable D2.1.2.

From the operational point of view, mapping two different ontologies implies the following actions:

- Data model mapping. The data structures of the source ontology should be compared with the data structure of the destination ontology in order to find out all the correspondences in terms of homologous concepts. The Mapping module supports transformations that allow users to express these correspondences in a structured way.
- Taxonomy mapping. Both the source ontology and the destination ontology should provide their own taxonomies. The taxonomy mapping consists on finding out the correspondences among terms in the source taxonomies and relevant terms in the destination taxonomy.
- Vocabulary mapping. Similarly to taxonomy mapping, term correspondences should be defined for all the vocabulary terms.

The outcomes of these actions are persisted according to the following figure:



The deliverable D2.1.2 goes through the detailed view on the two transformation files. Instead, this paper is in charge of describing which transformations are currently supported by the Semantic Translator and how they are mapped against the high level mapping operations defined in the previous sections. The next section provides these clarifications.

### 2.3.4 Basic mapping operations

Once defined and recalled the involved requirements, the following section depicts the basic mapping operations supported by the Semantic Translator. For each operation, a table with the required parameters, a description of what output should the operation yield and an example are provided.

#### Concatenation

<b>Input nodes</b>	Set of source tree nodes.
<b>Output node</b>	Single destination tree node.

<b>Concatenation text strings</b>	Set of text strings representing the items by which concatenating the input nodes.
-----------------------------------	--

This operation concatenates the contents of the selected input nodes by placing the specified concatenation string as separator. Therefore, the number of separator should be the same as the number of input nodes (the blank text string is allowed). The result of the operation is placed in the specified output node.

The following example concatenates three nodes by means of the following concatenation terms: “-“, “(“, “)”.

Input document	Output document
<pre>&lt;address&gt;   &lt;street&gt;via Roma, 21&lt;/street&gt;   &lt;zip_code&gt;41100&lt;/zip_code&gt;   &lt;country&gt;Italy&lt;/country&gt; &lt;/address&gt;</pre>	<pre>&lt;address&gt;   via Roma, 21 - 41100 (Italy) &lt;/address&gt;</pre>

### Copy

<b>Input node</b>	Single source tree node.
<b>Output node</b>	Single destination tree node.

It copies the content of the selected input node in the selected destination tree node without altering the schema structure. This represents the simplest applicable operation.

Input document	Output document
<pre>&lt;address&gt;   &lt;street&gt;via Roma, 21&lt;/street&gt;   &lt;zip_code&gt;41100&lt;/zip_code&gt;   &lt;country&gt;Italy&lt;/country&gt; &lt;/address&gt;</pre>	<pre>&lt;address&gt;   &lt;location&gt;via Roma, 21&lt;/location&gt;   &lt;zip_code&gt;41100&lt;/zip_code&gt; &lt;/address&gt;</pre>

### Division

<b>Input nodes</b>	Set of source tree nodes.
<b>Output node</b>	Single destination tree node.
<b>First operand</b>	The source tree node, selected among the input nodes, that represents the first operand of the arithmetic expression.
<b>Constant value</b>	A constant value that can be inserted in the arithmetic expression.

The operation performs an arithmetic division among the specified source nodes and, if present, the constant value. The result of the operation is copied in the specified output node. The first operand parameter specifies which operand should lead the arithmetic expression.

The operation can be executed only if all the specified input nodes represent a valid number format, either integer or decimal. If so, the result is an integer/decimal value. The following example represents a division operation where *tot\_price* plays the role of first operand.

Input document	Output document
<pre>&lt;accounting&gt;   &lt;tot_price&gt;     20.00   &lt;/tot_price&gt;   &lt;quantity&gt;5&lt;/quantity&gt; &lt;/accounting&gt;</pre>	<pre>&lt;spec_account&gt;   &lt;single_price&gt;     4.00   &lt;/single_price&gt; &lt;/spec_account&gt;</pre>

### Multiplication

<b>Input nodes</b>	Set of source tree nodes.
<b>Output node</b>	Single destination tree node.
<b>Constant value</b>	A constant value that can be inserted in the arithmetic expression.

The operation performs an arithmetic division among the specified source nodes and, if present, the constant value. The result of the operation is copied in the specified output node.

The operation can be executed only if all the specified input nodes represent a valid number format, either integer or decimal. If so, the result is an integer/decimal value. The following example represents a multiplication operation between *single\_price* and *quantity*.

Input document	Output document
<pre>&lt;spec_account&gt;   &lt;single_price&gt;     4.00   &lt;/single_price&gt;   &lt;quantity&gt;5&lt;/quantity&gt; &lt;/spec_account&gt;</pre>	<pre>&lt;accounting&gt;   &lt;tot_price&gt;     20.00   &lt;/tot_price&gt; &lt;/accounting&gt;</pre>

### Split by delimiter

<b>Input node</b>	Single source tree node
<b>Output nodes</b>	Set of destination tree nodes
<b>Delimiter text strings</b>	Set of text strings representing the delimiter by which splitting the input node.

This function split the content of the selected input node into the selected output nodes. The splitting function is based on the list of provided delimiter according to the following algorithm. The first token is the leading substring of the source text truncated at the first occurrence of the first delimiter (delimiter excluded). The second token is obtained by the next token of source string truncated at the first occurrence of the second delimiter. The process is iterated until the source string end is reached.

The first yielded token is then copied in the first output node, the second token in the second output node, etc. until tokens are available and empty output nodes exists. The following example split the *address* in two fields by specifying the following delimiter: “,”.

Input document	Output document
<pre>&lt;address&gt;   via Roma, 21 &lt;/address&gt;</pre>	<pre>&lt;address&gt;   &lt;street&gt;via Roma&lt;/street&gt;   &lt;street_nbr&gt;21&lt;/street_nbr&gt;</pre>

	</address>
--	------------

### Split by field length

<b>Input node</b>	Single source tree node.
<b>Output nodes</b>	Set of destination tree nodes.
<b>Field lengths</b>	Set of text strings representing the lengths of characters of each expected tokens.

This function split the content of the selected input node into the selected output nodes. The splitting function is based on the list of provided lengths according to the following algorithm. The first token is the leading substring of the source text truncated at the *n*th character, where *n* is the first number of the field length list. The second token is obtained by the next token of the source string truncated at the *m*th character, where *m* is the second number of field length list. The process is iterated until the source string end is reached or no more field are available.

The first yielded token is then copied in the first output node, the second token in the second output node, etc. until tokens are available and empty output nodes exists. The following example split the *address* in two fields by specifying the following field lengths: “10”, “5”.

Input document	Output document
<pre>&lt;address&gt;   via Roma  21 &lt;/address&gt;</pre>	<pre>&lt;address&gt;   &lt;street&gt;via Roma  &lt;/street&gt;   &lt;street_nbr&gt;21&lt;/street_nbr&gt; &lt;/address&gt;</pre>

### Subtraction

<b>Input nodes</b>	Set of source tree nodes.
<b>Output node</b>	Single destination tree node.
<b>First operand</b>	The source tree node, selected among the input nodes, that represents the first operand of the arithmetic expression.
<b>Constant value</b>	A constant value that can be inserted in the arithmetic expression.

The operation performs an arithmetic subtraction among the specified source nodes and, if present, the constant value. The result of the operation is copied in the specified output node. The first operand parameter specifies which operand should lead the arithmetic expression.

The operation can be executed only if all the specified input nodes represent a valid number format, either integer or decimal. If so, the result is an integer/decimal value. The following example represents a subtraction operation where *tot\_weight* plays the role of first operand.

Input document	Output document
<pre>&lt;freight&gt;   &lt;tot_weight&gt;     20.00   &lt;/tot_weight&gt;   &lt;net_weight&gt;15&lt;/net_weight&gt; &lt;/freight&gt;</pre>	<pre>&lt;frg&gt;   &lt;tare_weight&gt;     5.00   &lt;/tare_weight&gt; &lt;/frg&gt;</pre>

**Sum**

<b>Input nodes</b>	Set of source tree nodes.
<b>Output node</b>	Single destination tree node.
<b>Constant value</b>	A constant value that can be inserted in the arithmetic expression.

The operation performs an arithmetic sum among the specified source nodes and, if present, the constant value. The result of the operation is copied in the specified output node.

The operation can be executed only if all the specified input nodes represent a valid number format, either integer or decimal. If so, the result is an integer/decimal value. The following example represents a sum operation between *tare\_weight* and *net\_weight*.

<b>Input document</b>	<b>Output document</b>
<pre>&lt;frg&gt;   &lt;tare_weight&gt;     5.00   &lt;/tare_weight&gt;   &lt;net_weight&gt;15&lt;/net_weight&gt; &lt;/frg&gt;</pre>	<pre>&lt;freight&gt;   &lt;tot_weight&gt;     20.00   &lt;/tot_weight&gt; &lt;/freight&gt;</pre>

**Dictionary based structural translation**

<b>Input</b>	The dictionary edited by the COMM Manager by means of the Editor Module.
<b>Output</b>	The destination document.

So far, a set of structural operations have been outlined, i.e. operations that wires the concepts of the source ontology with the concepts of the destination ontology by means of the instructions provided by the mapping activity.

However, it should be taken into account that the overall SEAMLESS architecture prefigures that no explicit mapping activity is performed between GLOB and COMM. Indeed, the COMM ontology is derived by the GLOB ontology and the most relevant activity performed by the COMM manager is translating the ontology contents from English (lingua franca) to local language. This means that the GLOB concepts adopted by the COMM share the same data model structure but with different languages. The result of the translation task is stored in a dictionary file.

Hence, in case of either GLOB to COMM or COMM to GLOB translations, no structural translation is needed except for the XML tag linguistic translation. The Semantic Translator provides a facility for enabling such kind of structural translation. The same dictionary is used for performing both the translations.

<b>Input document (GLOB)</b>	<b>Output document (COMM, Italian)</b>
<pre>&lt;address&gt;   &lt;street&gt;via Roma,   21&lt;/street&gt;   &lt;zip_code&gt;41100&lt;/zip_code&gt;   &lt;country&gt;Italy&lt;/country&gt; &lt;/address&gt;</pre>	<pre>&lt;indirizzo&gt;   &lt;via&gt;via Roma, 21&lt;/via&gt;   &lt;cap&gt;41100&lt;/cap&gt;   &lt;nazione&gt;Italy&lt;/nazione&gt; &lt;/indirizzo&gt;</pre>

### Linguistic content translation

<b>Input</b>	The dictionary edited by the COMM Manager by means of the Editor Module (for COMM to GLOB and GLOB to COMM translation). Otherwise, the dictionary edited by means of the Mapper module.
<b>Input</b>	The destination ontology.
<b>Output</b>	The destination document.

Regardless of which structural translation is applied, the Semantic Translator has to apply a linguistic translation of the destination document tree node contents. The translation is driven by:

- The destination ontology. The Semantic Translator queries the destination ontology in order to find out which nodes have to be translated according to their relevant data types.
- The dictionary. The dictionary provides the actual translations.

For simple terms, the translation process replaces the source terms with their translations, if available. If no translation has been supplied, the term is presented according to its original form. For taxonomy terms, the replacement occurs only if the whole term path is found within the dictionary.

<b>Input document (GLOB)</b>	<b>Output document (COMM, Italian)</b>
<pre> &lt;product&gt;   &lt;type&gt;shoes&lt;/type&gt;   &lt;availability&gt;     in stock   &lt;/availability&gt; &lt;/ product &gt; </pre>	<pre> &lt;prodotto&gt;   &lt;tipo&gt;scarpe&lt;/tipo&gt;   &lt;disponibilita&gt;     a magazzino   &lt;/disponibilita&gt; &lt;/prodotto&gt; </pre>

## 3 Analysis of alternative tools

This chapter addresses the state-of-the-art in the field by analysing the existing ontology-based transformation tools, with special attention to the well-known Altova MapForce, and measuring their ability to meet the SEAMLESS requirements. The analysis section and the decision to realise a new Semantic Translator is preceded by a synthesis of the XSLT language features. The XSLT language is a well-known standard, recommended by the W3C (World Wide Web Consortium), and used by each of the tools examined below.

### 3.1 XSLT recalled

XSLT stands for XSL Transformation. XSL (eXtensible Stylesheet Language ) started to be developed by the W3C because there was a real need for an XML-based style sheet language. Style sheets describe how documents are displayed, pronounced, or printed. The XSL language consists of three parts: XSLT, XPath, and XSL Formatting Objects.

Below the attention is focused on the XSLT, that is the most important part of the XSL, rather than to the XPath and XSL Formatting Objects.

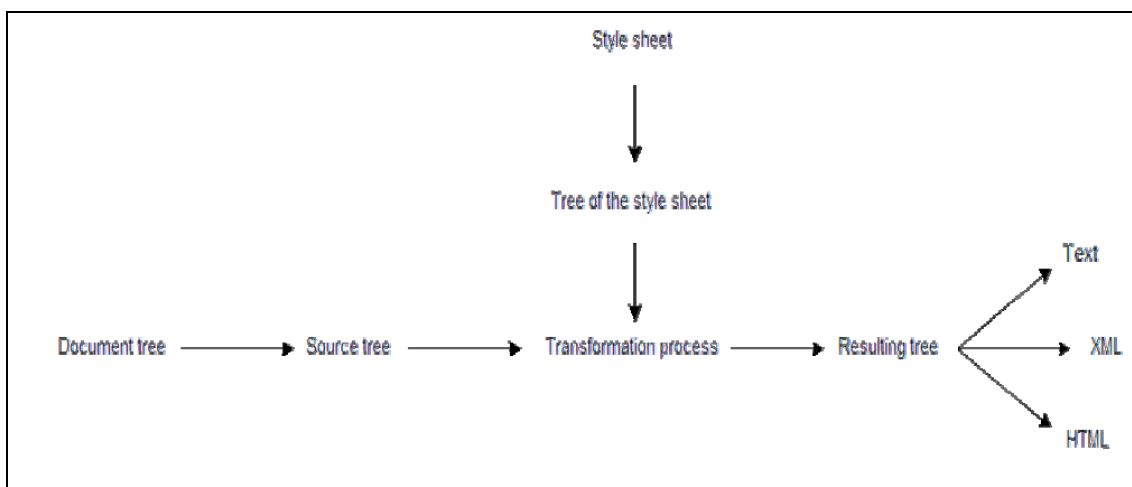
XSLT is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X)HTML element.

XSLT applications are manifold: for example, all the organizations, investing time and money in the information source creation to comply with XML standards, should be able to transform those XML formats in order to send them to the web browser as well as to mobile terminal and TVs. Moreover, the transformation of XML data, sent as message, compliant to the DTD schema adopted by the addressee, is an other issue.

The XSLT style sheet processing consists of two different phases:

- Structural transformation: input XML data are converted into a structure that is conform with the desired output.
- Formatting: the new structure is converted in the output format, for example HTML o PDF.

The structural transformation consists of operations as data selection (that is the choice of some of the input elements and attributes contained in the XML input document), data ordination and arithmetic conversion execution (for example the conversion of inches in centimeters). The figure below shows the data flow of the XSLT transformation.



Finally, it can be interesting to underline another XSLT feature, quite independent from the other topics contained in this paragraph: XSLT and database. In fact XSLT resembles to some query languages used for the relational databases. Indeed, the XSLT role when operating with relational databases (but not only relational) consists in transforming the database information into an XML data format. Then, the preference for XSLT is generally justified by its capacity to transform heterogeneous data source rather than the relational databases information alone.

### 3.1.1 XSLT and the Semantic Translator

This first version of the Semantic Translator relies on the capabilities provided by the XSLT 1.0 language. The choice of adopting it as implementation language for these functions has been taken according to the following reasons:

- XSLT is a XML-based transformation language, well-known within the developers community. It provides transformation predicates along with procedural patterns (if-condition, loops, variable assignment, parameters, functions, etc.).
- XSLT is an interpreted language, it does not introduce any additional compilation step throughout the development process.
- According to the previous points, it can be considered a good prototypal language for defining simple transformation procedures. Many graphical tools for authoring and testing XSLT code are nowadays available.
- A XSLT procedure can be simply embedded and executed within a JAVA programming language due to the availability of many libraries acting as XSLT processors. Typically, few lines of code are enough for loading the style sheet, applying it and managing the outcome.

In case the usage of an interpreted language as XSLT significantly affects the translator performance, the transformation code can be re-factored adopting the JAVA language. Indeed, converting a XSLT procedure into JAVA code is a straightforward task

## 3.2 State-of-the-art

In order to justify the decision to develop a brand new Semantic Translator, this section proposes a survey and perform an objective comparison of the currently available tools. To this purpose the following list of requirements is preliminarily defined to measure the compliance level of such tools with the identified SEAMLESS needs.

<b>On-line translation service</b>	Capability of providing the translation services by means of a Web Service interface.
<b>Structural translation based on transformation files</b>	Facility for performing a structural translation driven by associations among source and destination data structures (transformation file).
<b>Structural translation based on dictionary file</b>	Facility for performing a structural translation driven by associations among source and destinations terms (dictionary file).
<b>Linguistic content translation</b>	Capability of translating documents content according to associations among source and destinations terms (dictionary file) and content typology (data types).
<b>Independent translator implementation</b>	The implementation of the translator functions is independent from the XML file providing the translation instructions set. This XML file should represent the separation interface between the Semantic Translator and the Mapper module implementations.
<b>XML query translation by re-using already defined</b>	Facility for translating XML queries re-using the already defined mapping instances, e.g. dictionary and data model mapping.

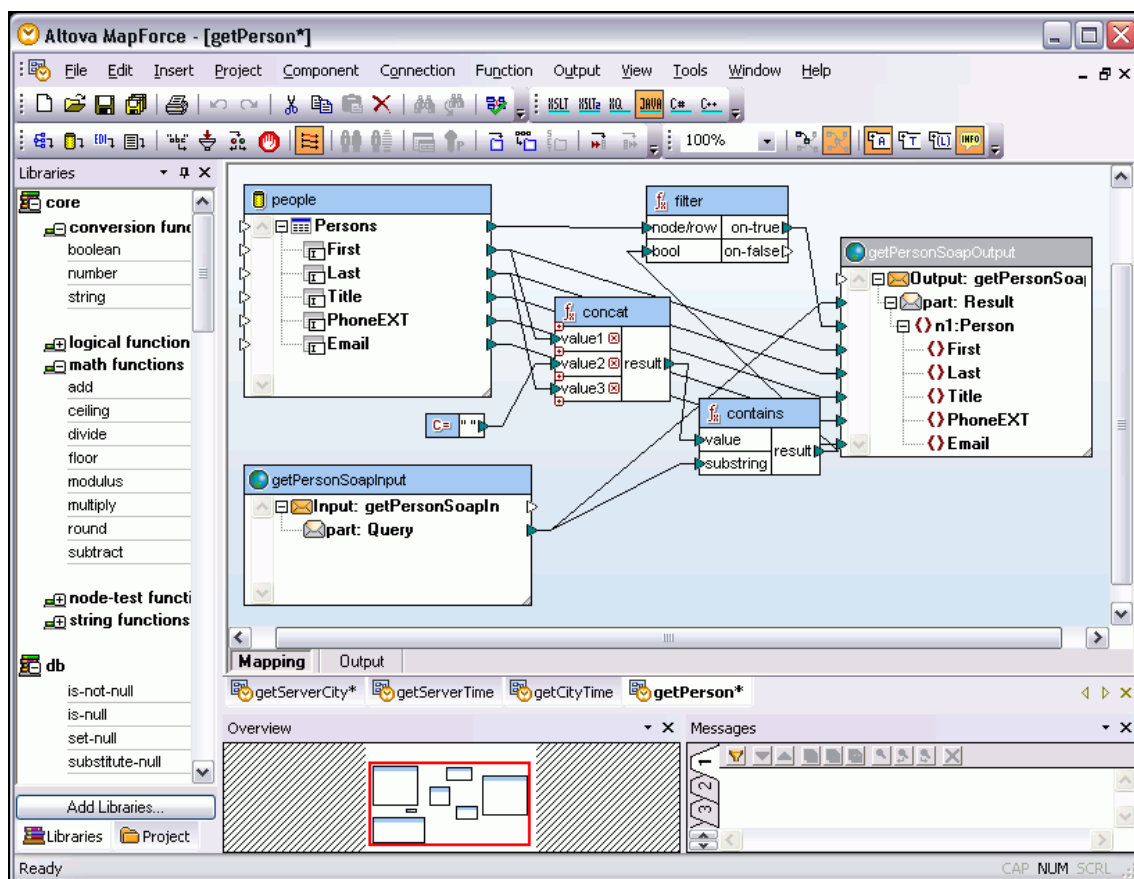
mapping instances	
Open source license	Possibility of modifying the product source code according to the project requirements.

### 3.2.1 MapForce 2007 (<http://www.altova.com/products/mapforce>)

The Altova MapForce software (“Altova MapForce 2007 Standard Edition”) is currently the commercial leader product for XML mapping tools, thus representing the state-of-the-art of this sector.

The rationale of the product is allowing the final users to graphically draw the connections between two different XML Schema thus providing the actual mapping instructions. The output of the tool is a XSLT document that can be applied to any XML document compliant with the source schema in order to obtain a document compliant with the destination schema.

A typical screenshot of the application is:



In detail, the user is firstly asked to open both the source and the destination schemas. MapForce will perform a graphical representation of both, according to the above visualisation. Then, the user performs the mapping activity by graphically connecting the attributes of the two schemes, in case by placing a function block within for carrying out specific computation.

The advanced features provided by MapForce are:

- Automatic association between homonymous attributes children of an already mapped concept.
- Visualisation of attributes data types.

- Function library for specifying the mapping between attributes by means of conditional tests, Boolean logic operations, text string operations, numeric operations. Functions can support a parametric behaviour if required.
- XSLT function library for specifying the mapping between attributes by means of XSLT 1.0 and 2.0 functions.
- Capability of composing new functions by arranging existing function blocks.
- Possibility of managing many source and destination schemes within the same mapping activity.
- The mapping result can be represented in many output formats: XSLT, Xquery, Java, C#.

### 3.2.2 Delta 4.0 (<http://www.softshare.com>)

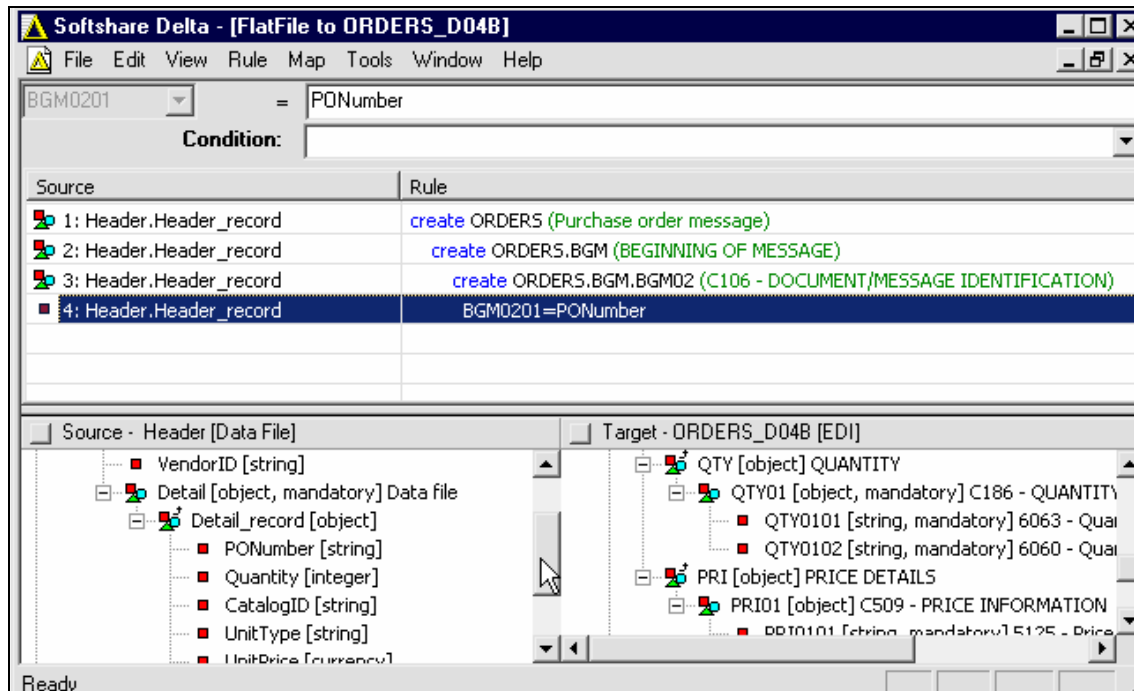
As a universal data translator, Delta not only supports EDI standards (X12, EDIFACT and TRADACOM), but a variety of other data formats as well, including data (flat) files, database tables (via OLE DB), and XML. Delta also supports mapping to free-form text document formats such as HTML to aid in Web integration. When modelling and mapping, Delta treats each data format individually, leveraging the unique strengths of each one with features and built-in intelligence designed specifically for that format. In general, source allowed data formats include EDI, Flat File, Database, XML, while the target data format includes EDI, Flat File, Database, XML, Text.

No back-end application should be an island. Using Delta, you can derive additional benefit from back-end systems such as SAP, Oracle, JDE, Epicor, the Sage MAS products, and many others by integrating disparate data sources into one powerful e-commerce enterprise.

The main features of the product are enlisted below:

- Any-to-any mapping supported. The advantages of any-to-any mapping are twofold: not only are people able to integrate all incoming and outgoing electronic documents with their internal applications, but they are also able to integrate between their internal applications (regardless of data format), allowing for total enterprise integration.
- Source and target-driven mapping. Delta's ability to perform rules based upon the order of their source or target data is invaluable when they are dealing with disparate source and target data formats whose layouts cannot otherwise be reconciled.
- Built-in-intelligence. Delta's expression builder to incorporate functions, conditions, variables, and constants into your mapping rules.
- Mapping rules testing and validation (post process) .

A typical screenshot of the application is:



### 3.2.3 Stylus Studio (<http://www.stylusstudio.com>)

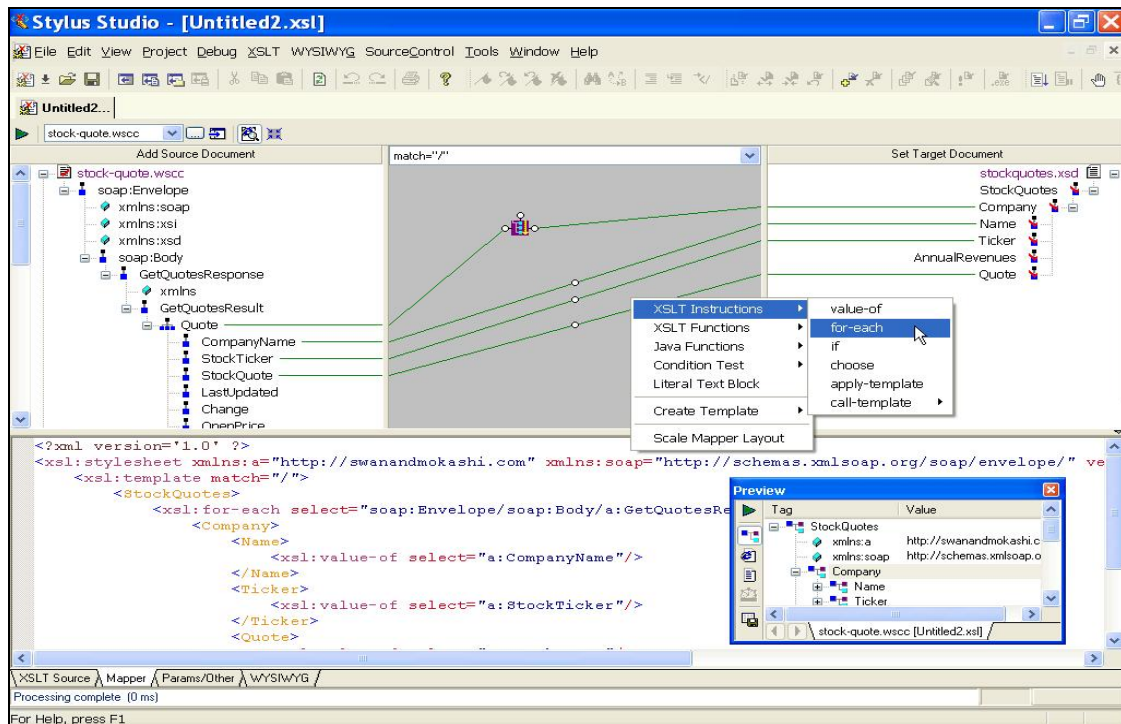
Stylus Studio 2007 XML Enterprise Suite provides a comprehensive set of XML tools and features for working with XML, XQuery, SQL/XML, Web services, XML publishing, and other XML technologies.

Stylus Studio provides a set of XML mapping tools. XML Mapper supports seamless mapping from databases, XML files, EDI Mapping, DTDs, XML Schema Mapping, mapping live Web service data, or getting any other kind of legacy data into an XML format of your choosing. Users can create advanced mappings that incorporate any number of data sources mapped to a target destination, and can create custom data processing functions written in programming languages such as Java. Stylus Studio is standards based — users can just open an existing XSLT or XQuery file and map away. And once you are done creating your XML mappings, you can take your XSLT or XQuery to go — an integrated Java Code Generator will generate the Java code needed to deploy their code to a live server application.

The main features of the product are enlisted below:

- Support mapping from XML-to-XML, relational database-to-XML, Web service data-to-XML, and much more. Stylus XML Mapper can handle virtually any input and output format, and perform the mapping in XSLT or XQuery. The following input formats are supported by this XML Mapper: XML Instance Document, XML Schema, Document Type Definition (DTD), Relational Database, Live Web Service Data, EDI, X12, EDIFACT, IATA or any other non-xml data format. Of course, Stylus Studio supports advanced multi-data-source data integration scenarios involving multiple input sources.
- XML-Schema to XML-Schema mapping. Visual mapping tool that allows user to easily implement sophisticated XML data mappings involving multiple data sources and customized data processing using either XSLT or XQuery code. Advanced XML Schema mapping involving multiple schemas.
- Advanced XML technologies like XQuery supported.
- XSLT instruction. The XSLT mapper represent the following XSLT instruction: “xsl:value-of”, “xsl:for-each”, “xsl:if”, “xsl:choose”, “xsl:apply-templates”, “xsl:call-template”.
- XSLT function. The XSLT mapper represent many operations working on ‘string’ data type.

A typical screenshot of the application is:



### 3.2.4 Comparison and conclusion

This section concludes the analysis by showing the deficiencies of the considered software products respect to the SEAMLESS Semantic Translator:

- ‘ YES ’ : it means that the product manages the feature;
- ‘ NO ’ : it means that the product does not manage the feature.

SEAMLESS requirements	MapForce 2007	Delta 4.0	Stylus Studio
On-line translation service	NO	NO	NO
Structural translation based on transformation files	YES	YES	YES
Structural translation based on dictionary file	NO	NO	NO
Linguistic content translation	NO	NO	NO
Importable translation instructions	NO	NO	NO
XML query translation by re-using already defined mapping instances	NO	NO	NO
Open source license	NO	NO	NO

The comparison points out the following aspects:

- The three analysed tools do not provide an on-line, Web based service for performing translations. Translations are only possible if performed by means of the tool GUI.

- The three tools focus the translation process on the data structure, i.e. they provide advanced features for manipulating XML schemas. Instead, they do not implement proper facilities for performing translation based on the linguistic contents (taxonomy and vocabulary). Generally speaking, they do not supply the required integration level with already edited ontologies.
- The main achievement of these tools is providing a user interface for performing a mapping activity rather than supplying a translation service. This consideration is supported by the implementation choice of representing the mapping instructions in a single XSLT file which is intended as the only translation means. This solution implies that the only possible translation implementation is a XSLT transformer.
- Differently from the previous point, the overall SEAMLESS requirements suggests the introduction of open communication interfaces. In other terms, the mapper module should provide the translation instructions while the translator should implement the logic for carrying out them.
- The absence of open source licenses means no capability of manipulating the current versions of the three products, thus limiting the possibility of integrating them within the overall SEAMLESS architecture.

In conclusion of this analysis we can see that none of these tools provides the Translator features that are expected by the SEAMLESS project. This is the reason why it was decided to design and develop the brand new tool described in this deliverable and released as software module in D2.2.1.

## 4 Semantic Translator module

The chapter provides the technical specification of the Semantic Translator, the web service in charge of performing the transformation of queries and documents from their source ontology to a given destination ontology.

**This Translator is new.** It is developed in the frame of task T2.2 with the aim of being general enough to transform queries, business documents and all other data types to be exchanged between partners, and to move from each of the SEAMLESS ontologies to the previous or the next one in the ontology hierarchy. Since this point on, the term **document** represents any XML file compliant with a given SEAMLESS ontology, therefore it includes business documents, companies and product profiles. On the other hand, the **query** is a file defining some search criteria on well-defined types within a SEAMLESS ontology.

### 4.1 Document transformation

The document transformation process defines the capability of the Semantic Translator of producing a XML document (destination document) whose contents are extracted by a source XML document. The source and destination documents can differ in terms of both data structure and language. Then, the translation process addresses two main tasks:

- **Structural translation.** The process by which the structure of the source document is converted into the structure of the destination document. Since both the documents are compliant with the XML language, structural translation means altering the source XML for obtaining a destination XML tree whose contents respect, as much as possible, the source ones.
- **Vocabulary translation.** This process is in charge of performing the linguistic translation of the document contents according to the destination ontology data types. Indeed, only a subset of the available data types is intended to be translated. Moreover, it is important to underline how the linguistic translation is required any time the source and destination ontology differs in terms of vocabulary or taxonomy.

#### 4.1.1 Structural translation

The structural translation process needs the following input information:

- **Structural file.** The outcome of the data model mapping providing the translation instructions for converting the source document data structure into the destination one. This file is not required if a COMM to GLOB or GLOB to COMM translation is carried out. In this case, the two documents shares the same structure except for the XML tag name. Hence, the dictionary file only is required.
- **Dictionary file.** The file containing the associations between the vocabulary and taxonomy terms of the two involved ontologies. The file can be produced by means of the Mapper module or Editor module (COMM Manager only)<sup>14</sup>. It is always adopted as base of knowledge for performing the linguistic translation. In case of COMM to GLOB or GLOB to COMM translation, it is used for structural translation as well.
- **Source document file.** The XML document to be translated. The document has to be a valid XML file according to the XSD schema defined in the source ontology<sup>15</sup>.
- **Destination document schema file.** The XSD schema defined in the destination ontology<sup>16</sup> providing the structure of the destination document. This file is adopted as base of knowledge for automatically editing the destination document according to the structural file indications.

The structural translation based on the dictionary file is simply carried out as a linguistic translation (if available) of the involved XML tags.

---

<sup>14</sup> See deliverable D2.1.2 for more details.

<sup>15</sup> See deliverable D2.1.2 for more details.

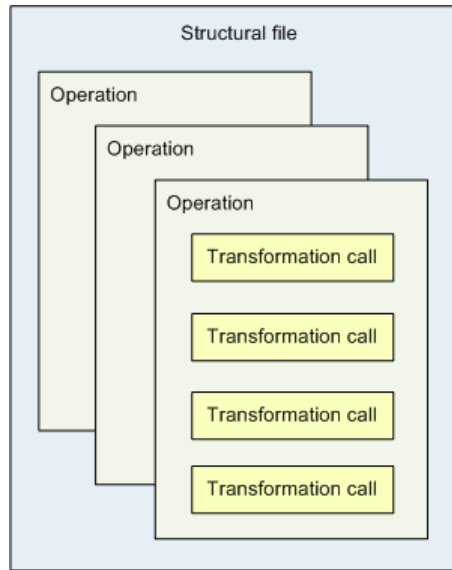
<sup>16</sup> See deliverable D2.1.2 for more details.

On the other hands, the translation based on the structural file depends on the instructions provided by the users throughout the mapping activity. According to the analysis produced in the section 2.3, a set of high level **operations** have been designed in order to provide the final users with a straightforward mechanism for issuing the mapping instructions. A brief recall on these operations follows:

<b>Concatenation</b>	Appends the content of different source nodes in a single destination node.
<b>Copy</b>	Copies the content of a single source node in a single destination node.
<b>Division</b>	Performs a division arithmetic operation among source nodes and places the result in a destination node.
<b>Multiplication</b>	Performs a multiplication arithmetic operation among source nodes and places the result in a destination node.
<b>Split by delimiter</b>	Splits the content of a single source node into many destination nodes according to the occurrences of the provided delimiter within the source content.
<b>Split by field length</b>	Splits the content of a single source node into many destination nodes according to the provided token lengths.
<b>Subtraction</b>	Performs a subtraction arithmetic operation among source nodes and places the result in a destination node.
<b>Sum</b>	Performs a sum arithmetic operation among source nodes and places the result in a destination node.
<b>Dictionary based structural translation</b>	Translates the node names of a source document according to the provided dictionary information.
<b>Linguistic content translation</b>	Translates the node contents of a source document according to the provided dictionary information.

On the Structural file point of view, the high level operations contains a set of low level **transformation calls**. The transformation call, defined within the structural file operation, invokes a basic elaboration unit that performs a simple, localised, manipulation of the source document. The outcome of each transformation call is then moved within the destination document.

To sum up, a single operation is carried out by an ordered sequence of transformations call while the overall structural translation process is the result of the application of many operations.



On the implementation point of view, the transformation calls actually invoke XSLT style sheet functions registered in the Semantic Translator. The currently available transformation calls along with their relevant required parameters are here enlisted (see Appendix: XSLT transformation functions for implementation details):

<b>add-element.:</b> appends a new tree node as a child of all the occurrences of an already existing nodes.	
<b>transformation_search-procedure</b>	Search
<b>transformation_stylesheet</b>	05_add-element.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path</b>	Xpath path referencing the existing tree node which plays the role of parent node.
<b>param : name_new-element</b>	Name of the new node
<b>param : text_default</b>	A text string by which the new node is populated.

<b>copy-element_many-to-many_RENAME:</b> copies all the occurrences of the specified tree node and their relevant sub-tree as children of another tree node.	
<b>transformation_search-procedure</b>	searchCommonPath
<b>transformation_stylesheet</b>	07_copy-element_many-to-many_RENAME.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path-1</b>	Xpath path representing the node to be copied
<b>param : path-2</b>	Xpath path representing the destination node
<b>param : name_new-element</b>	New name for the copied node

<b>modify-text-with-arithmetic-operation:</b> applies an arithmetic operation adopting the occurrences of a single specified node as operands. If the specified operand nodes cannot be converted in a numeric value, the transformation yields a NaN (Not-a-Number) value.	
---	--

<b>transformation_search-procedure</b>	Search
<b>transformation_stylesheet</b>	15_modify-text-with-arithmetic-operation.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path</b>	Xpath path of the node playing the role of operand and, at the same time, result.
<b>param : operator</b>	The arithmetic operator (+, -, *, /)
<b>param : decimals</b>	Decimal digits represented in the final result
<b>param : operand</b>	A constant value representing an additional operand.

<b>explosion-1-to-N-with-container-node_RENAME:</b> transforms a text node in N children nodes by separating the source text in N tokens according to the provided separator list.	
<b>transformation_search-procedure</b>	Search
<b>transformation_stylesheet</b>	17_explosion-1-to-N-with-container-node_RENAME.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path</b>	Xpath path of the source node
<b>param : name_new-element</b>	New name for the source node
<b>param : list_text-processing-parameters</b>	List of text strings providing the criteria for splitting text : number of characters or text separator string.
<b>param : list_node-names</b>	The list of names to be associated to the new N nodes.

<b>explosion-1-to-N-with-container-node_RENAME:</b> transforms a text node in N children nodes by separating the source text in N tokens according to the provided number.	
<b>transformation_search-procedure</b>	Search
<b>transformation_stylesheet</b>	17_bis_explosion-1-to-N-with-container-node_RENAME.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path</b>	Xpath path of the source node
<b>param : name_new-element</b>	New name for the source node
<b>param : list_text-processing-parameters</b>	List of text strings providing the criteria for splitting text : number of characters or text separator string.
<b>param : list_node-names</b>	The list of names to be associated to the new N nodes.

<b>fusion-N-to-1-with-container-node_separator_attribute-fusion:</b> merges all the children node contents of a specified tree node by means of the supplied separator	
<b>transformation_search-procedure</b>	Search
<b>transformation_stylesheet</b>	19_fusion-N-to-1-with-container-node_separator_attribute-fusion.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path</b>	Xpath path to the parent node which is intended to contain

	the final result
<b>param : name_new-element</b>	New name for the resulting node
<b>param : separator</b>	Text string adopted for separating the nodes contents

<b>fusion-2-to-1-without-container-node_arithmetic-operation_attribute-fusion:</b> applies an arithmetic operation adopting the single occurrences of the two specified nodes as operands. If the specified operand nodes cannot be converted in a numeric value, the transformation yields a NaN (Not-a-Number) value.	
<b>transformation_search-procedure</b>	searchCommonPath
<b>transformation_stylesheet</b>	20_fusion-2-to-1-without-container-node_arithmetic-operation_attribute-fusion.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path-1</b>	Xpath path referencing to the first operand.
<b>param : path-2</b>	Xpath path referencing to the second operand.
<b>param : operator</b>	The arithmetic operator (+, -, *, /)
<b>param : decimals</b>	Decimal digits represented in the final result
<b>param : name_new-element</b>	Name of the tree node in which the final result is placed. This tree node replaces the first operand node.

<b>fusion-2-to-1-without-container-node_separator_attribute-fusion:</b> merges all the occurrences of two sibling nodes in a single node, separating the nodes contents by the supplied separator value.	
<b>transformation_search-procedure</b>	searchCommonPath
<b>transformation_stylesheet</b>	20_fusion-2-to-1-without-container-node_separator_attribute-fusion.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path-1</b>	Xpath path for the first node
<b>param : path2</b>	Xpath path for the second node
<b>param : name_new-element</b>	Name for the resulting node
<b>param : separator</b>	Text string adopted for separating the nodes contents

<b>fusion-multiple-occurrences_summation-product_attribute-fusion:</b> applies an arithmetic operation adopting all the occurrences of the specified tree node as operands. If the specified operand nodes cannot be converted in a numeric value, the transformation yields a NaN (Not-a-Number) value.	
<b>transformation_search-procedure</b>	searchOccurrence
<b>transformation_stylesheet</b>	21_fusion-multiple-occurrences_summation-product_attribute-fusion.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path</b>	Xpath path referencing the operands nodes.
<b>param : choice</b>	Type of operation : summation or product
<b>param : decimals</b>	Decimal digits represented in the final result

<b>param : name_new-element</b>	Name of the tree node in which the final result is placed. This tree node replaces the operand node.
---------------------------------	--

<b>fusion-multiple-occurrences_separator_attribute-fusion:</b> merges the multiple occurrences of the specified node in a single node by appending the nodes contents.	
<b>transformation_search-procedure</b>	searchOccurrence
<b>transformation_stylesheet</b>	21_fusion-multiple-occurrences_separator_attribute-fusion.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path</b>	Xpath path of the sibling nodes to be merged.
<b>param : name_new-element</b>	Name of the node containing the merge result.
<b>param : separator</b>	Text string used as separator throughout the merging task.

<b>transform-root:</b> replaces the root node of the source document with the root node of the destination document, adding the missing namespaces.	
<b>transformation_search-procedure</b>	NONE
<b>transformation_stylesheet</b>	22_transform-root.xsl
<b>file_namespaces</b>	File containing the complete list of available namespaces
<b>param : file_destination</b>	The destination file
<b>param : file_source</b>	The source file
<b>param : file_source-namespaces</b>	File containing the list of available namespaces in the source document.
<b>param : file_destination-namespaces</b>	File containing the list of available namespaces in the destination document.

<b>transform-root_cleanup:</b> Removes "xmlns="" attributes from the destination document, introduced as side-effect of the translation operations.	
<b>transformation_search-procedure</b>	NONE
<b>transformation_stylesheet</b>	22_transform-root_cleanup.xsl

<b>rename-element:</b> Renames all the occurrences of a tree node with the supplied new name.	
<b>transformation_search-procedure</b>	Search
<b>transformation_stylesheet</b>	23_rename-element.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path</b>	Xpath path of the node to be renamed
<b>param : new-name</b>	The node new name.

<b>final-cleanup:</b> removes from the destination document the namespaces declarations imported by the source document and no longer used in the destination one.	
--	--

<b>transformation_search-procedure</b>	NONE
<b>transformation_stylesheet</b>	25_final-cleanup.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : file_source-namespaces</b>	File containing the list of available namespaces in the source document.
<b>param : file_destination-namespaces</b>	File containing the list of available namespaces in the destination document.

<b>remove-separator-from-text:</b> removes the last occurrence of the separator text from the specified node. This transformation is often used to clear the result of a merging transformation.	
<b>transformation_search-procedure</b>	Search
<b>transformation_stylesheet</b>	27_remove-separator-from-text.xsl
<b>param : file_namespaces</b>	File containing the complete list of available namespaces
<b>param : path</b>	Xpath path of the node on which the operation is carried out.
<b>param : separator</b>	The text string representing the separator to be removed.

Once defined the operations and the transformation, the **operation-transformation relationship** can be defined, i.e. the sequence of transformations executed by the Semantic Translator in order to perform any single operation:

<b>High level operation</b>	<b>transformation call sequence</b>
<b>Concatenation</b>	<ul style="list-style-type: none"> <li>• transform-root</li> <li>• transform-root_cleanup</li> <li>• add-element</li> <li>• copy-element_many-to-many_RENAME</li> <li>• fusion-multiple-occurrences_separator_attribute-fusion</li> <li>• fusion-2-to-1-without-container-node_separator_attribute-fusion</li> <li>• remove-separator-from-text</li> <li>• final-cleanup</li> </ul>
<b>Copy</b>	<ul style="list-style-type: none"> <li>• transform-root</li> <li>• transform-root_cleanup</li> <li>• final-cleanup</li> </ul>
<b>Division</b>	<ul style="list-style-type: none"> <li>• transform-root</li> <li>• transform-root_cleanup</li> <li>• add-element</li> <li>• copy-element_many-to-many_RENAME</li> <li>• fusion-multiple-occurrences_summation-product_attribute-fusion</li> <li>• fusion-2-to-1-without-container-node_arithmetic-operation_attribute-fusion</li> </ul>

	<ul style="list-style-type: none"> <li>• final-cleanup</li> </ul>
<b>Multiplication</b>	<ul style="list-style-type: none"> <li>• transform-root</li> <li>• transform-root_cleanup</li> <li>• add-element</li> <li>• copy-element_many-to-many_RENAME</li> <li>• fusion-multiple-occurrences_summation-product_attribute-fusion</li> <li>• fusion-2-to-1-without-container-node_arithmetic-operation_attribute-fusion</li> <li>• modify-text-with-arithmetic-operation</li> <li>• final-cleanup</li> </ul>
<b>Split by delimiters</b>	<ul style="list-style-type: none"> <li>• transform-root</li> <li>• transform-root_cleanup</li> <li>• explosion-1-to-N-with-container-node_RENAME</li> <li>• final-cleanup</li> </ul>
<b>Split by field length</b>	<ul style="list-style-type: none"> <li>• transform-root</li> <li>• transform-root_cleanup</li> <li>• bis_explosion-1-to-N-with-container-node_RENAME</li> <li>• final-cleanup</li> </ul>
<b>Subtraction</b>	<ul style="list-style-type: none"> <li>• transform-root</li> <li>• transform-root_cleanup</li> <li>• add-element</li> <li>• copy-element_many-to-many_RENAME</li> <li>• fusion-multiple-occurrences_summation-product_attribute-fusion</li> <li>• fusion-2-to-1-without-container-node_arithmetic-operation_attribute-fusion</li> <li>• final-cleanup</li> </ul>
<b>Sum</b>	<ul style="list-style-type: none"> <li>• transform-root</li> <li>• transform-root_cleanup</li> <li>• add-element</li> <li>• copy-element_many-to-many_RENAME</li> <li>• fusion-multiple-occurrences_summation-product_attribute-fusion</li> <li>• fusion-2-to-1-without-container-node_arithmetic-operation_attribute-fusion</li> <li>• modify-text-with-arithmetic-operation</li> <li>• final-cleanup</li> </ul>

#### 4.1.2 Vocabulary translation

The vocabulary translation process relies on the following input information:

- Dictionary file. The file containing the associations between the vocabulary and taxonomy terms of the two involved ontologies. The file can be produced by means of the Mapper module or

Editor module (COMM Manager only)<sup>17</sup>. It is always adopted as base of knowledge for performing the linguistic translation. In case of COMM to GLOB or GLOB to COMM translation, it is used for structural translation as well.

- Source document file. The XML document to be translated. The document has to be a valid XML file according to the XSD schema defined in the source ontology<sup>18</sup>.
- Source vocabulary file. The vocabulary file defined in the source ontology<sup>19</sup>. This file is used as word reference for looking-up synonyms of the source terms that cannot be translated. Indeed, the vocabulary translation keep trace of the not translated terms and, for each of them, provides the list of available synonyms in order to make available a more comprehensive logging information.
- Destination document schema file. The XSD schema defined in the destination ontology<sup>20</sup> providing the structure of the destination document. This file is adopted as base of knowledge for detecting which contents of the destination document has to be translated. Indeed, the translation process has to be applied only to those contents related to specific data types, namely term attribute, numeric parameter attribute, enumerative parameter attribute and taxonomy term.
- Destination vocabulary file. The vocabulary file defined in the destination ontology<sup>21</sup>. It provides the vocabulary translation process with the information for deciding if a source term has to be translated (no correspondence in the destination vocabulary found).

In case of structural translation based on dictionary file, the vocabulary translation process is firstly in charge of translating the tag node names of the source document file. The translation is carried out by applying the dictionary file association rule to each tag of the source document. This produces a new document whose tag are named according to the destination ontology vocabulary.

Regardless of the applied structural translation process, the vocabulary translation works on an intermediate version of the destination document. This version presents the correct data structure, i.e. the destination one, but the contents are still written according to the source vocabulary. The following steps are then performed:

- The destination document schema file is accessed to look-up which node contents has to be translated according to their relevant data types.
- For each of the nodes yielded at the previous step, the current term is read. Then, the process look-up the destination vocabulary file in order to check if the same term is already available in that vocabulary. If so, no additional operation is executed since it means the node content is correct for the destination vocabulary.
- Since no match has been found in the destination vocabulary, the dictionary file is accessed in order to find out a possible translation. If the translation exists, the destination term replaces the source one. If it does not exist, no replacement occurs and the not translated term along with its available synonym is logged.

The result of this process represents the actual destination document.

## 4.2 Query transformation

The aim of this section is to introduce and justify the design choice adopted in the SEAMLESS project in order to manage the input/output query structure. In this scenario, the solution that allows automatic query translations assumes XQuery as input/output query language. This is the solution adopted in the present version of the query transformation for the reasons described below. In particular, it should be able, at any SEAMLESS ontology level, to guarantee the query translation and to allow partner search and so on.

---

<sup>17</sup> See deliverable D2.1.2 for more details.

<sup>18</sup> See deliverable D2.1.2 for more details.

<sup>19</sup> See deliverable D2.1.2 for more details.

<sup>20</sup> See deliverable D2.1.2 for more details.

<sup>21</sup> See deliverable D2.1.2 for more details.

The choice of XQuery as query language has been adopted in virtue of its high level of application, because it is side-effect free and useful for XML-to-XML transformation and for information retrieval. Moreover, it adopts a SQL-like syntax, surely easy to be read.

The choice of XQuery has been taken after a precise comparison with other XML-based query language:

- SQL/XML. It is a standard for representing SQL queries by means of a XML-like structure. SQL/XML queries can only work on relational data storages. Since it does not treat heterogeneous sources (XML), it has not been considered appropriate according to the SEAMLESS requirements.
- XQueryX . It is a standard for representing a generic query language by means of a XML-like structure. Since it is mainly a query representation language, it is not adequate for actually performing queries against a given data store.
- SPARQL. It is a XML query language working on RDF-based data stores. This feature could be limiting according to the overall design choice of representing SEAMLESS information as simple XML documents.

The possibility to transform queries expressed in other languages is however maintained. As we shall see in section 4.2.2, before being submitted to the Semantic Translator the query undergoes a pre-processing phase to extract its predicates and possible return fields, and represent them in a “neutral” XML form that is independent of the query language. These predicates and return fields are the only parts of the query processed by the Translator, and the result is finally post-processed to re-build the query.

The pre- and post-processing functions that are developed in the present Semantic Translation release are those parsing and composing queries in the XQuery language. Should the e.g. SPARQL language be preferred in the future or in a different context, this will just call for developing the relative pre- and post-processing functions (according to the technical specifications provided below) while the transformation process remains unchanged.

In conclusion, it is important to point out how the choice of XQuery as input/output query language is just an implementation choice due to the motivations asserted above. Therefore, the overall architecture of the Semantic Translator is designed in order to accept other possible query languages.

#### 4.2.1 SEAMLESS requirements on XQuery structure

The XQuery language is very rich: it allows the use of XPath expression, XML elements construction, it supports any classic programming constructs, it makes available more than hundred data function and it permits some new functions to be defined.

As a matter of fact a query can be represented in various ways then, as a design choice, these ways have to be limited by introducing specific requirements that all the SEAMLESS query should follow.

The solution proposed for the SEAMLESS project focuses on the most versatile construct of this language: the “*flwors expression*.”. The “*flwors expression*” stands for for-let-where-order-by-return expression that represents the standard blocks that a XQuery statement can contain. The block semantics recalls the “select-from-where” clauses of the SQL language. Hence, the first requirement on the SEAMLESS query is that it has to be a strict “*flwors*” XQuery statement.

An example of a XQuery query follows:

```
xquery version "1.0" encoding "UTF-8";

let $x:=doc("companies.xml")

for   $company in $x/*
      $emailAddress in $company/emailAddress,
      $name in $company/name,
      $cityName in $company/address/cityName,
      $iso in $company/address/country/isoCode

where $emailAddress eq "tony@virgilio.it" and
      (($name eq "Tony") or ($name eq "Anto"))and
```

```

(($cityName eq "Modena")or($cityName eq "Bologna"))and
$iso = 678

return $company/phoneNumber, $company/name

```

In detail:

- **let:** it is non-mandatory. If it is present, it must contain the reference to the XML document to query.
- **for:** it contains all the fields that are considered in the where clause, each of them expressed by the full XPath path assumed in the XML document tree (the “\$company” alias represents any child element of “\$x”. In this example, the root node is a company list container, then “\$company” is associated to any actual company instance).
- **where:** it includes the searching conditions collected together by means of “and” and “or” operators. Each condition cares about the relation between a tree field and a value by means one of these “XQuery comparisons”: “=”, “!=”, “>”, “<”, “=>”, “<=”, “eq”, “ne”, “lt”, “le”, “gt”, “ge”.
- **return:** it refers to all the fields requested as query result, expressed by a full-path. It is possible to return all the fields of the XML tree document.

Considerations:

- It is supposed that a query is allowed to contain no more than one “let” and one “for” clause, although it would be allowed by the language.
- Ordination activities on the content are here not taken into account since meaningless for the SEAMLESS project.
- There are no constraints in order to limit the use of nested parenthesis, tabs and white spaces.

Additionally, the following constructs have to be considered prohibited:

XPath conditions: for \$r in \$x/address[cityName="Modena"] which should expressed as:

```

for $c in $x/address/cityName,
    where $c eq "Modena"

```

- XML or HTML tags

Moreover, XQuery does not take into account the disjunctive queries, as for the most XML query languages. This behaviour can be obtained by means of a union operation of conjunctive queries.

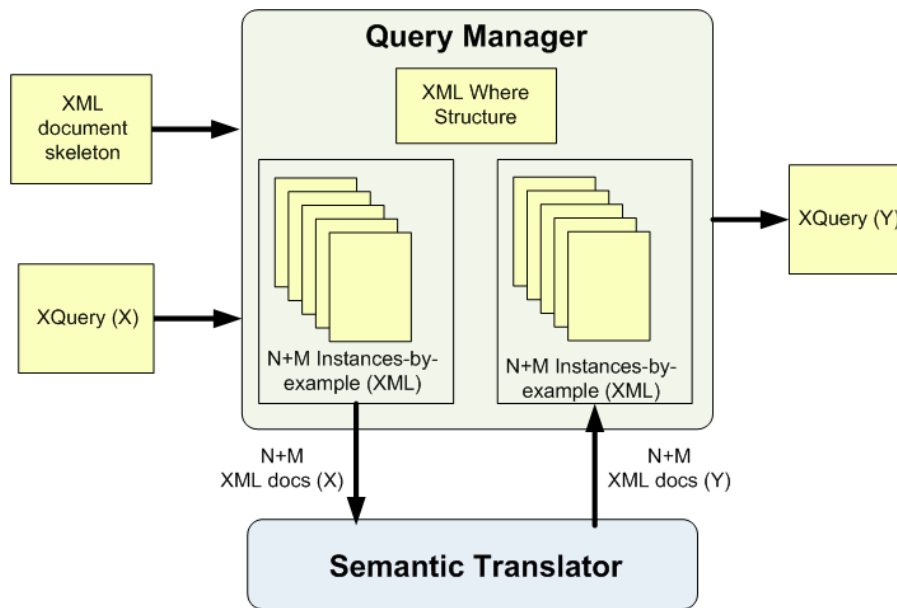
## 4.2.2 Query translation solution

The query translation process that works at every SEAMLESS ontology level is performed by the same Semantic Translator service provided it is wrapped by another service called the Query Manager (see the figure in the next page).

Besides assuring the possibility to use the Semantic Translator for both documents and queries, there is another reason to introduce the Query Manager: it allows to decouple the transformation process from the used query language. Indeed, the only parts of the query that shall undergo the transformation process are the predicate and the possible return fields, and they can be easily extracted from the query and expressed in a “neutral” XML format (Query Manager pre-processing). The transformation produces a new XML version that is used to re-build the query in the original language (Query Manager post-processing).

The Query Manager expected inputs are:

- Source XQuery query. The Source query written according to the ontology X concepts and the XQuery requirements explained in the section above.
- XML document skeleton. An XML document compliant with the ontology X definition of the concept on which the query is intended to be performed. The document simply provides an empty skeleton, i.e. the tree node contents are not valued.



The Query Manager analyses the incoming query and then produces two different types of document: Query-by-example instances and the Where Structure.

The **Query-by-example instances**. Each instance is a XML document obtained as a copy of the provided skeleton. For any where-condition found in the input query, a document is instantiated and only the field referencing the original “where” condition is set to the value present in the source query (in the example, the number of these documents is N).

In the same way, the Query Manager generates instances referring to the “return” fields (in the example, the number of these documents is M); if ALL the fields are requested (such as in the SQL SELECT \*) it is  $M=0$ . “N+M XML docs (X)” in the figure above indicates all the generated XML document instances. N+M stays for the total number of these instance-document, where N refers to the number of XQuery “where” conditions, M refers to the number of XQuery “return” fields. N+M is the sum of these two quantities.

In order to provide an example, let assume an Italian company looking for an English company which produces chairs. Since the query expresses two conditions, namely country equal to England and product equal to chair, the number of Company instances due to the “where” condition (N) is 2. One should look like:

```
...
<nazione>Inghilterra</nazione>
...
```

and the other:

```
...
<prodotto>sedia</prodotto>
...
```

Since the query is issued by an Italian company, these instances are compliant with the Italian ontology. The missing part of the Company instances marked as “...” are meaningless since they provide empty contents.

As result of the Semantic Translator invocation, the two instances are converted according to the target ontology:

```
...  
<country>England</country>  
...
```

and:

```
...  
<product>chair</product>  
...
```

The final task is setting-up a new XQuery statement based upon the two new instances.

To do this, the query-by-example instances are not enough to translate the source query in the destination query because they do not manage any information about the comparison and logical operators used in the “where” predicate. To complete the query translation the “where structure” is necessary.

The **Where Structure** is an XML file containing some information of the structure of the source query “where” conditions. The Query Manager generates this XML file by analysing the where conditions operator and relationships. Then, this XML file contains information of the operator and references the query-by-example instances for representing the query operands.

Once the query-by-example instances and the where structure are set, the Query Manager invokes the regular translation methods provided by the Document Transformation Service in order to translate all the produced instances. This can be accomplished since all the instances are actual SEAMLESS documents.

Once the Query Manager received the translated instances, then it reconstructs the query on the basis of the destination schema by means of the Where Structure file that never changes in content and structure until the process is concluded.

## 4.3 Translator architecture design and implementation

This SECTION goes through the Semantic Translator design by firstly outlining the overall architecture and, then, covering the implementation details.

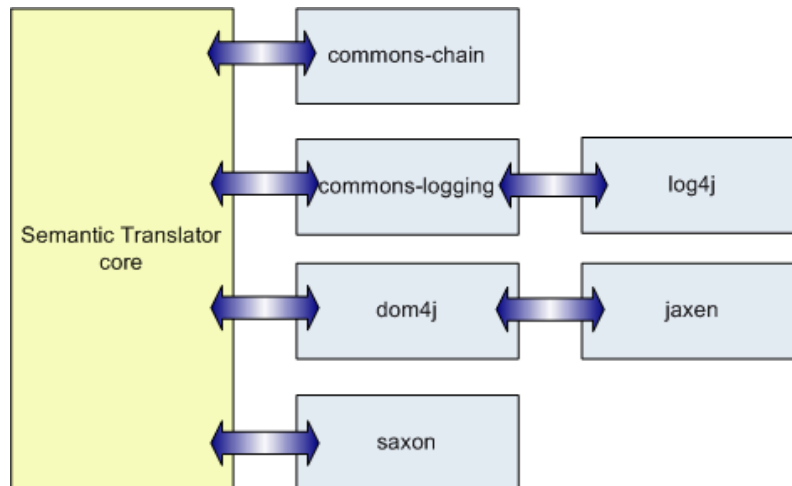
### 4.3.1 Major design decisions

The overall Semantic Translator design process has been inspired by the following guidelines:

- Open source components. Adoption, as far as possible, of open source technologies in order to produce an expandable solution, not limited by commercial license restrictions.
- Component based approach. Leveraging on well-documented and stable third party components in order to exploit their services. The inclusion of external libraries bring relevant benefits to the whole development process: (i) inclusion of already tested and good quality code, (ii) time and cost saving, (iii) introduction of de-facto standard such as frameworks, patterns, etc. that should improve the software modularisation and maintainability.
- Interoperability. A fundamental aspect of the development of the Semantic Translator module is represented by the capability of coping within a Service Oriented Architecture. This is an essential key-point taking into account the overall idea of the SEAMLESS network.

### 4.3.2 High level architecture

This figure introduces the high level architecture of the Semantic Translator module, where the yellow block represents the Semantic Translator core, the light blue the external libraries and the blue arrows the communication among different modules:



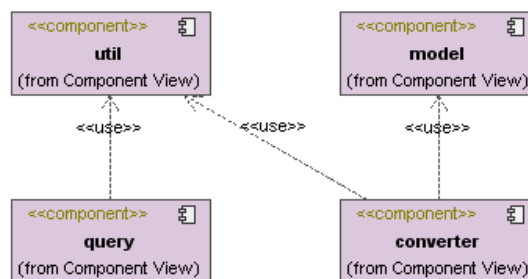
The **commons-chain** module (<http://jakarta.apache.org/commons/chain/>) implements a popular technique for organising the execution of complex processing flows according to the "Chain of Responsibility" pattern, as described (among many other places) in the classic "Gang of Four" design patterns book. Although the fundamental API contracts required to implement this design pattern are extremely simple, it is useful to have a base API that facilitates using the pattern, and (more importantly) encouraging composition of command implementations from multiple diverse sources. In particular, this is the case of the Semantic Translator module since it includes several execution flows.

The **commons-logging**<sup>22</sup> and **log4j**<sup>23</sup> libraries provide services for logging management. Since the Semantic Translator is conceived as a service without interactive user interface, the use of logging facilities is necessary.

**Dom4j** module (<http://www.dom4j.org>) is a flexible Java API for either parsing or creating XML contents in a programmatic fashion. The parsed XML files can be simple browsed or modified and then being saved to a persistent support again. Within the Editor, dom4j is used as XML parser for the ontology contents such as dictionary, vocabulary and taxonomy.

**Saxon** (<http://saxon.sourceforge.net/>) is the library that implements the XSLT processor adopted by the Semantic Translator. This open source implementation offers a reliable and easy-to-use API for executing XSLT style sheets.

The **Semantic Translator core** module has been developed in order to address the explicit SEAMLESS requirements. The component diagram realising the Semantic Translator core module is depicted in the figure below.



Generated by UModel

www.altova.com

<sup>22</sup> <http://jakarta.apache.org/commons/logging/>

<sup>23</sup> <http://logging.apache.org/log4j/docs/index.html>

The following sections provide a more detailed view on each component.

### 4.3.3 Semantic Translator core – converter component

This component includes the classes in charge of implementing the document transformation process.

Interface Summary	
<b>IConverterInput</b>	Provides input information to the converter.
<b>IConverterProperties</b>	Provides properties to the converter.

Class Summary	
<b>CommandLineStart</b>	Launch the converter for testing.
<b>Converter</b>	Main class performing the conversion task.
<b>ConverterContext</b>	Context shared among the command implementations.
<b>ConverterInit</b>	Static initializer for the Converter.
<b>ElementNameConverter</b>	Applies the element name conversion.
<b>ExecuteTransformationCmd</b>	Executes the transformation.
<b>OutputTransformationCmd</b>	Output the document.
<b>PrepareExecutionCmd</b>	Load the transformation stylesheet and input document.
<b>StructureConverter</b>	Applies the structure conversion.
<b>TermConverter</b>	Applies the term based conversion.
<b>UntranslatedTerm</b>	Not translated term information.

### 4.3.4 Semantic Translator core – query component

This component includes the classes in charge of implementing the query transformation process.

Interface Summary	
<b>Generation</b>	Interface for the creation of the temporary files (implemented by DocGenerator).
<b>Translation</b>	Interface for the translation of the input query (implemented by TranslatedQuery).

Class Summary	
<b>Clause</b>	It contains a condition-clause of the query's where (ie. a field-comparator-value triple).

<b>DirList</b>	It contains the utilities needed to explore a directory.
<b>DocGenerator</b>	It contains the methods used to create the temporary files needed for the query translation (instance-by-example files and where-structure file).
<b>GenerateReturn</b>	Utilities needed to translate the return fields of the query (saved in the file "return.txt").
<b>IBE</b>	It contains some utilities to create and use instances by example.
<b>QueryManager</b>	It includes the QueryManager's functions (creation of IBE and where-structure files and translation of the input query).
<b>QueryParser</b>	It contains methods to break up the input query into its parts (let, for, where,...) and save the where and return conditions into a txt file.
<b>TranslatedQuery</b>	It contains the utilities needed to reconstruct the final translated query.
<b>Validation</b>	Contains the utilities for the validation of the WhereStructure xml document.
<b>WhereScheme</b>	It contains the utilities to create the scheme of the where-statement's structure.
<b>WhereStatement</b>	It contains a list of all the local-datascheme-fields to query and all the methods about it (i.e. a list of "Clause" object).
<b>XmlUtil</b>	It contains general utilities that can be used with XML documents.
<b>XSDCreation</b>	It contains the utilities to create the XSD of the where-statement's scheme.

#### 4.3.5 Semantic Translator core – util component

Component providing utility classes.

Class Summary	
<b>CustomLogger</b>	Utility class for logging functions.
<b>Folder2Document</b>	Converts a folder-structured document into a dom4j document.

#### 4.3.6 Semantic Translator core – model component

This component contains model classes providing an abstraction upon a XSD schema in order to obtain SEAMLESS compliant information.

Interface Summary	
<b>IDataModelNode</b>	Generic node representing a data model part.

Class Summary	
<b>DataModelFolder</b>	Container for IDataModelNode.

<b>DataModelReader</b>	Reads data model and produces relevant information
<b>DefaultDataModelNode</b>	Default implementation of IDataModelNode.
<b>IdGenerator</b>	Unique id generator class.

## 4.4 Translator WS interface

This section defines the web service interface exposed by the Semantic Translator module. This interface is designed taken into account these points:

- The translation service is intended to be invoked frequently, any time a communication between different ontologies occurs. Then, performance issues should be taken into account.
- The input information required for a single translation process may include ontology vocabulary and other files whose dimension could significantly impact on the invocation response time. Because of this, a proper caching mechanism should be introduced in order to re-use the same ontology files for many transformations.

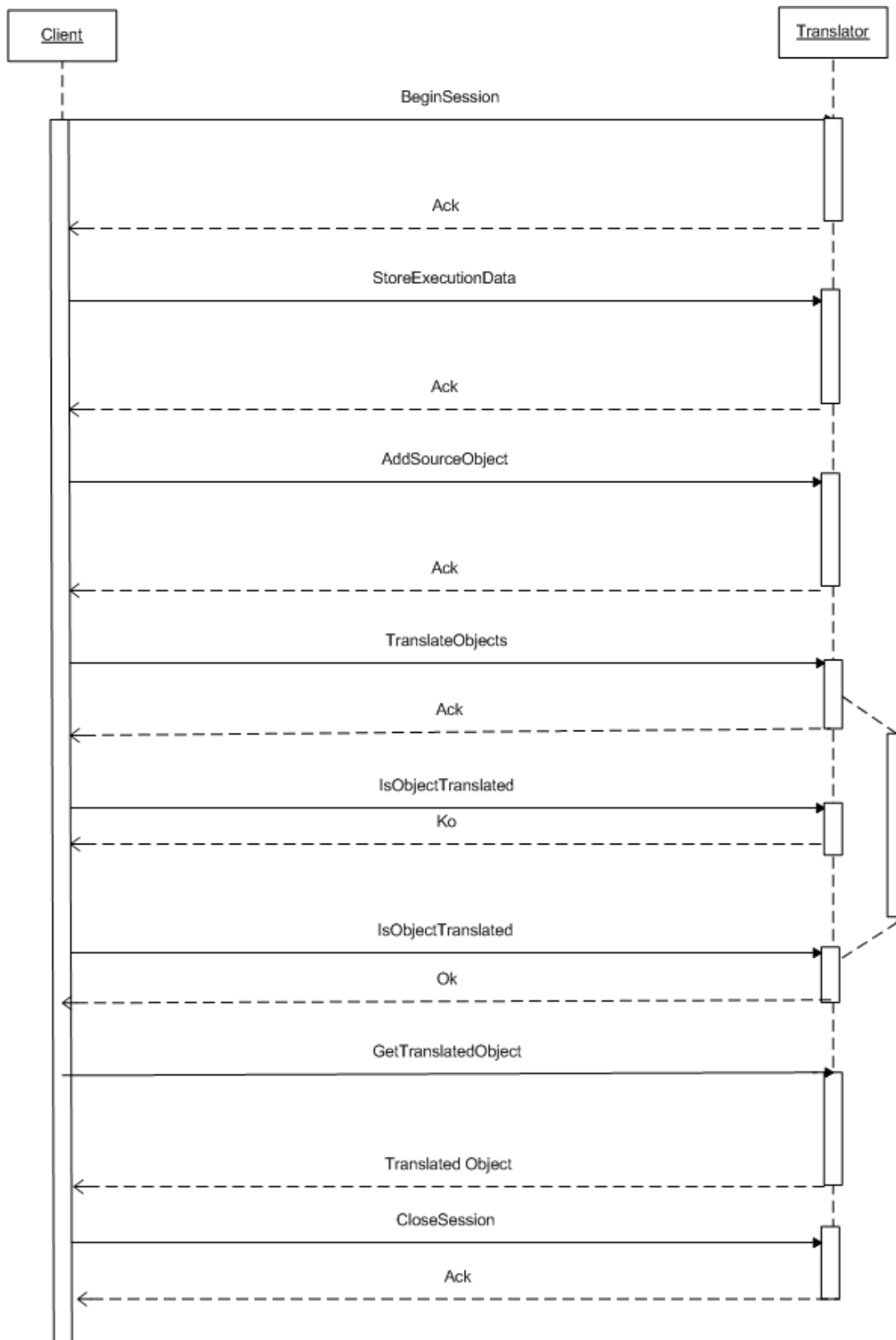
The diagram in the following page depicts the interface invocation sequence:

At the beginning, the client invokes the **BeginSession** method to allocate the service resources (ontology files) and specify which kind of translation is required. The service response is an acknowledgment (Ack) stating whether the requested resources are already available or, on the contrary, which resources are needed.

The **StoreExecutionData** is then called in order to add the possible missing resources. An acknowledgment is returned stating if the resources has been correctly loaded. The **AddSourceObject** let the clients to upload the source files, either documents or query, to be translated.

The **TranslateObjects** method starts the translation process. This service actually delegates the execution of the translation process to another process in order to immediately return an acknowledgment stating whether the translation process has been correctly started.

Once the translation is started, the clients are asked to invoke the **IsObjectTranslated** method to check whether the translation has been carried out. Finally, the **GetTranslatedObject** and **CloseSession** methods are called to, respectively, return the translated object and close the opened session.



The detailed list of the exposed methods follows.

Before examining them in detail it should be noted that the first two methods mention “metadata” as call parameters. These metadata have no relations with the metadata used to identify and characterise the objects stored by the SRRN. These just represent the information that is needed to know which file are currently available in the cache and then if any of them must be replaced before starting the translation process.

**int StoreExecutionData(Object data, string metadata)**

Stores ontology information or translation data in the Semantic Translator module along with proper metadata describing the provided data. This method feeds the Semantic Translator internal cache.

**Parameters:**

- *data*. The file to be stored.
- *metadata*. Information describing the provided file.

**Returns:**

0 if the operation succeeded, otherwise a negative integer value representing an error code.

**string BeginSession(string username, string password, string src\_ontology\_meta, string dest\_ontology\_meta, int execution\_mode, int data\_model\_type, string root\_type\_name)**

Invokes the Semantic Translator for initialising the resources for a new translation session. The returned value is a XML string containing the session information.

**Parameters:**

- *username*. Username associated to the new session.
- *password*. Password associated to the new session.
- *src\_ontology\_meta*. Metadata identifying the source ontology.
- *dest\_ontology\_meta*. Metadata identifying the destination ontology.
- *execution\_mode*. 0 – document translation; 1 – query translation
- *data\_model\_type*. Identifies the ontology data model type. 0 – collaboration; 1- company; 2 – negotiation; 3 – product and service
- *root\_type\_name*. Identifies the data model root type within the provided data\_model\_type.

**Returns:**

A XML string containing either the session identifier in case all the required execution data are available in the Semantic Translator cache, or the list of data that are missing. If the latter, the StoreExecutionData should be called for storing the missing information and, then, the BeginSession again.

**string AddSourceObject(string username, string password, string session\_id, Object obj)**

Adds a document or a query to be translated to the provided session. The object structure has to be conform to the parameters provided to the BeginSession method. Many documents/queries can be translated within the same session.

**Parameters:**

- *username*. Username associated to the session.
- *password*. Password associated to the session.
- *session\_id*. Session identifier.
- *obj*. Object to be translated (document or query).

**Returns:**

An object identifier if the operation succeeds, otherwise an execution errors. The methods validates the request by the username, password and session\_id values.

**int TranslateObjects(string username, string password, string session\_id)**

Launches the translation process on the provided objects. This method is intended to return immediately after the process is started.

**Parameters:**

- *username*. Username associated to the session.
- *password*. Password associated to the session.
- *session\_id*. Session identifier.

**Returns:**

0 if the operation succeeded, otherwise a negative integer value representing an error code.

**int IsObjectTranslated(string username, string password, string session\_id, string obj\_id)**

Returns the translation execution state for the given object within the given session.

**Parameters:**

- *username*. Username associated to the session.
- *password*. Password associated to the session.
- *session\_id*. Session identifier.
- *obj\_id*. Object identifier.

**Returns:**

0 if the translation has successfully completed, 1 if the translation is still running, otherwise a negative integer value representing an error code.

**Object GetTranslatedObject(string username, string password, string session\_id, string obj\_id)**

Returns the translated object according the provided parameters.

**Parameters:**

- *username*. Username associated to the session.
- *password*. Password associated to the session.
- *session\_id*. Session identifier.
- *obj\_id*. Object identifier.

**Returns:**

The translated object if the process has been successfully executed, otherwise null.

**int CloseSession(string username, string password, string session\_id)**

Closes the session, removing all the allocated information.

**Parameters:**

- *username*. Username associated to the session.
- *password*. Password associated to the session.
- *session\_id*. Session identifier.
- 

**Returns:**

0 if the operation succeeded, otherwise a negative integer value representing an error code.

## 5 Final remarks

The current version of the Semantic Translator is to be intended as release 1.0. It can translate documents and queries according to what depicted in the previous chapters but some issues still exist that somehow limit the potentials of this solution and, then, they should be faced in order to improve the level of the application resilience.

The most relevant limitation of the current version of the Semantic Translator comes from the reduced set of information delivered by the transformation file. In particular, the translation operations are only based on mapping instructions linking the leaf nodes of the two (source and destination) data structures. This solution, effective in many situations, fails whenever the source and destination data structures differ in terms of hierarchy levels and, at translation time, the involved components occur more than once. Let us see an example:

```
<company>
  <site>
    <street/>
    <city/>
    <country/>
  </site>
</company>

<company>
  <site>
    <address>
      <street/>
      <city/>
      <country/>
    </address>
  </site>
</company>
```

This definition introduces two simple data structures (source on the left, destination on the right) delivering the same concept: a company can be defined by many addresses and each address contains three data, namely street, city and country. However, the two schemes differ in terms of hierarchy levels: the source schema introduces the component “site” while, the destination, “site” and “address”.

On the mapping point of view, the most natural mapping instructions in this case are linking the attributes “street”, “city” and “country” of the source schema with the homologous ones in the destination schema. Unfortunately, this information is not enough for instructing the Semantic Translator. Let us see why:

```
<company>
  <site>
    <street>Oxford street, 21</street>
    <city>London</city>
    <country>England</country>
  </site>
  <site>
    <street>Liverpool drive, 3</street>
    <city>London</city>
    <country>England</country>
  </site>
</company>
```

The definition above outlines a possible, valid source document.

The first transformation operation says something like “copy the /company/site/street content into /company/site/address/street node on the destination side”. In this case, the result after the first operation will be:

```
<company>
  <site>
    <address>
      <street>Oxford street, 21</street>
      <street>Liverpool drive, 3</street>
      <city/>
      <country/>
    </address>
  </site>
</company>
```

It is immediately clear how the obtained result is not the expected one, i.e. a company with two sites, the first site an address with one street (“Oxford street, 21”) and the second site an address with one street (“Liverpool drive, 3”). The limitation here is evident.

This wrong behaviour is due to the lack of information within the structural file. The missing information is how the component cardinalities should be linked in order to obtain a proper result. In the example above, the “street-to-street” mapping instruction has to be associated to the information that the site cardinality in the source document has to be the same of the site cardinality in the destination document. Definitely, this information can be provided only by the users who performs the mapping since it is strictly semantic-related and, then, it cannot be automatically drawn by the Semantic Translator.

In order to face such a problem, the present Semantic Translator 1.0 will be improved in order to properly manage the component cardinality constraints and, consequently, the Mapper module will provide the users with the capability to express this information.

On the overall architecture point of view, the communication interface between the Ontology Mapper and the Semantic Translator should be improved as well in order to better isolate the implementation issues of the Semantic Translator.

Finally, it is important to point out how the overall architecture leaves room for implementing the expected improvements without sensibly affecting other modules. In particular, the capability of performing data type conversion functions, the facility for managing the destination document multiplicity and the possibility of linking source and destination components imply:

- Upgrading the Ontology Mapper user interface in order to let the users insert the additional required information.
- Altering the structural file schema for holding the additional required information.
- Implementing the new algorithms in the Semantic Translator module.

# Appendix: XSLT transformation functions

This Appendix collects the style sheets written in XSLT language that implement the elementary (basic) transformation function used by the SEAMLESS Semantic Translator.

```
/* 05_add-element.xsl */
```

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <!--insert filename of namespace file-->
  <xsl:param name="file_namespaces"/>
  <!--insert XPath expression of pointed container element-->
  <xsl:param name="path"/>
  <!--insert name of element to create-->
  <xsl:param name="name_new-element"/>
  <!--insert default text of element to create-->
  <!--default value = UNSPECIFIED-->
  <xsl:param name="text_default">UNSPECIFIED</xsl:param>
  <xsl:include href="utility_copy-descendants.xsl"/>
  <!--<xsl:strip-space elements="*" />-->
  <xsl:template match="/">
    <xsl:element name="seamless-node">
      <xsl:apply-templates select="PLACEHOLDER"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*" />
      <xsl:call-template name="copy-descendants" />
      <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
      <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
      <xsl:element name="{ $name_new-element }" namespace="{ $ns }">
        <xsl:value-of select="$text_default" />
      </xsl:element>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

```
/* 07_copy-element_many-to-many_RENAME.xsl */
```

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <!--insert filename of namespace file-->
  <xsl:param name="file_namespaces"/>
  <!--insert XPath expression of pointed element to copy-->
  <xsl:param name="path-1"/>
```

```

<!--insert XPath expression of pointed container element-->
<xsl:param name="path-2"/>
<!--insert name of element to create-->
<xsl:param name="name_new-element"/>
<xsl:include href="utility_get-name.xsl"/>
<xsl:include href="utility_copy-descendants.xsl"/>
<xsl:include href="utility_get-common-path.xsl"/>
<xsl:include href="utility_search-A.xsl"/>
<xsl:include href="utility_search-B.xsl"/>
<xsl:template match="/">
  <xsl:element name="seamless-node">
    <xsl:apply-templates select="PLACEHOLDER"/>
  </xsl:element>
</xsl:template>
<xsl:template match="*">
  <xsl:copy>
    <xsl:copy-of select="@*" />
    <xsl:call-template name="processing_action" />
  </xsl:copy>
</xsl:template>
<xsl:template name="processing_action">
  <xsl:variable name="path_common">
    <xsl:call-template name="get-common-path">
      <xsl:with-param name="path-1" select="$path-1" />
      <xsl:with-param name="path-2" select="$path-2" />
    </xsl:call-template>
  </xsl:variable>
  <xsl:choose>
    <!--in case of XPath expression of destination element being different from common path-->
    <xsl:when test="string-length(substring-after($path-2, $path_common)) != 0">
      <!--search node corresponding to destination element-->
      <xsl:call-template name="search_navigation-A">
        <xsl:with-param name="path" select="substring-after($path-2, $path_common)" />
      </xsl:call-template>
    </xsl:when>
    <!--in case of XPath expression of destination element being equal to common path-->
    <xsl:otherwise>
      <xsl:call-template name="copy-descendants" />
      <xsl:call-template name="processing_reverted-to-common-path" />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
<xsl:template name="processing_action-A">
  <xsl:variable name="path_common">
    <xsl:call-template name="get-common-path">
      <xsl:with-param name="path-1" select="$path-1" />
      <xsl:with-param name="path-2" select="$path-2" />
    </xsl:call-template>
  </xsl:variable>
  <xsl:copy>
    <xsl:copy-of select="@*" />
    <xsl:call-template name="copy-descendants" />
    <xsl:call-template name="search_revert-to-common-path">

```

```

                <xsl:with-param name="path" select="substring-after($path-2, $path_common)"/>
            </xsl:call-template>
        </xsl:copy>
    </xsl:template>
    <!--revert current node to node corresponding to common path and apply template processing_reverted-to-common-path-->
    <xsl:template name="search_revert-to-common-path">
        <xsl:param name="path"/>
        <xsl:variable name="token" select="substring-after(substring($path, 2), '/')"/>
        <xsl:choose>
            <xsl:when test="string-length($token) != 0">
                <xsl:for-each select="..">
                    <xsl:call-template name="search_revert-to-common-path">
                        <xsl:with-param name="path" select="concat('/', $token)"/>
                    </xsl:call-template>
                </xsl:for-each>
            </xsl:when>
            <xsl:otherwise>
                <xsl:for-each select="..">
                    <xsl:call-template name="processing_reverted-to-common-path"/>
                </xsl:for-each>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
    <xsl:template name="processing_reverted-to-common-path">
        <xsl:variable name="path_common">
            <xsl:call-template name="get-common-path">
                <xsl:with-param name="path-1" select="$path-1"/>
                <xsl:with-param name="path-2" select="$path-2"/>
            </xsl:call-template>
        </xsl:variable>
        <xsl:choose>
            <!--in case of XPath expression of source element being different from common path-->
            <xsl:when test="string-length(substring-after($path-1, $path_common)) != 0">
                <!--search node corresponding to source element-->
                <xsl:call-template name="search_navigation-B">
                    <xsl:with-param name="path_to-copy" select="substring-after($path-1, $path_common)"/>
                </xsl:call-template>
            </xsl:when>
            <!--in case of XPath expression of source element being equal to common path-->
            <xsl:otherwise>
                <xsl:call-template name="processing_action-B"/>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
    <xsl:template name="processing_action-B">
        <!--substitute pointed element with new transformed element-->
        <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
        <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
        <xsl:element name="{ $name_new-element}" namespace="{ $ns}">
            <xsl:copy-of select="@*" />
            <xsl:call-template name="copy-descendants" />
        </xsl:element>
    </xsl:template>

```

```

</xsl:stylesheet>

/* 15_modify-text-with-arithmetic-operation.xsl */

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <!--insert XPath expression of pointed element-->
  <xsl:param name="path"/>
  <!--insert symbol of arithmetic operation to execute on text of pointed element-->
  <xsl:param name="operator"/>
  <!--insert numerical value of operand of arithmetic operation to execute on text of pointed element-->
  <!--ATTENTION: you must use a point as decimal separator-->
  <xsl:param name="operand"/>
  <!--insert number of decimals of result to display-->
  <!--default value = 2-->
  <xsl:param name="decimals">2</xsl:param>
  <xsl:include href="utility_calculate-expression.xsl"/>
  <xsl:include href="utility_add-decimals.xsl"/>
  <!--<xsl:strip-space elements="*" />-->
  <xsl:template match="/">
    <xsl:element name="seamless-node">
      <xsl:apply-templates select="PLACEHOLDER"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*" />
      <xsl:variable name="result">
        <xsl:call-template name="calculate-expression">
          <xsl:with-param name="op1" select="text()" />
          <xsl:with-param name="op2" select="$operand" />
        </xsl:call-template>
      </xsl:variable>
      <!--set conventional format to display result-->
      <xsl:variable name="format-string">
        <xsl:text>#</xsl:text>
        <xsl:if test="$decimals != 0">
          <xsl:text>.</xsl:text>
          <xsl:call-template name="add-decimals">
            <xsl:with-param name="counter" select="$decimals" />
          </xsl:call-template>
        </xsl:if>
      </xsl:variable>
      <xsl:value-of select="format-number($result, $format-string)" />
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>

/* 17_explosion-1-to-N-with-container-node_RENAME.xsl */

<?xml version="1.0" encoding="UTF-8"?>

```

```

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <!--insert filename of namespace file-->
  <xsl:param name="file_namespaces"/>
  <!--insert XPath expression of pointed container element-->
  <xsl:param name="path"/>
  <!--insert name of element to create-->
  <xsl:param name="name_new-element"/>
  <!--insert list of names of elements to create, each one followed by character | as a separator (nb: also the last one)-->
  <xsl:param name="list_node-names"/>
  <!--insert list of values for parameter of chosen text processing function, each one followed by character | as a separator (nb: also the last one)-->
  <xsl:param name="list_text-processing-parameters"/>
  <xsl:include href="placeholder.xml"/>
  <!--<xsl:strip-space elements="*" />-->
  <xsl:template match="/">
    <xsl:element name="seamless-node">
      <xsl:apply-templates select="PLACEHOLDER"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="*">
    <!--transform pointed container element-->
    <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
    <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
    <xsl:element name="{ $name_new-element}" namespace="{ $ns}">
      <xsl:copy-of select="@*" />
      <!--apply explosion action based on text node of pointed element-->
      <xsl:call-template name="processing_recursive-action">
        <xsl:with-param name="list_node-names" select="$list_node-names"/>
        <xsl:with-param name="list_text-processing-parameters" select="$list_text-processing-parameters"/>
        <xsl:with-param name="text" select="text()" />
      </xsl:call-template>
    </xsl:element>
  </xsl:template>
  <xsl:template name="processing_recursive-action">
    <xsl:param name="list_node-names"/>
    <xsl:param name="list_text-processing-parameters"/>
    <xsl:param name="text"/>
    <!--define first token of current list of node names as current node name-->
    <xsl:variable name="name_new-element" select="substring-before($list_node-names, '|')"/>
    <!--define first token of current list of parameter values as current parameter value-->
    <xsl:variable name="parameter" select="substring-before($list_text-processing-parameters, '|')"/>
    <!--create new element inside node corresponding to pointed element-->
    <!--with current node name as a name and with result of text processing function as text-->
    <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
    <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
    <xsl:element name="{ $name_new-element}" namespace="{ $ns}">
      <xsl:call-template name="text-processing">
        <xsl:with-param name="text" select="$text"/>
        <xsl:with-param name="parameter" select="$parameter"/>
        <xsl:with-param name="choice" select="'before'"/>
      </xsl:call-template>
    </xsl:element>
  </xsl:template>

```

```

<!--if there are more node names to process in current list of node names, go on with recursive action-->
<xsl:if test="string-length(substring-after($list_node-names, '|')) != 0">
  <xsl:variable name="remaining-text">
    <xsl:call-template name="text-processing">
      <xsl:with-param name="text" select="$text"/>
      <xsl:with-param name="parameter" select="$parameter"/>
      <xsl:with-param name="choice" select="'after'"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:call-template name="processing_recursive-action">
    <xsl:with-param name="list_node-names" select="substring-after($list_node-names, '|')"/>
    <xsl:with-param name="list_text-processing-parameters" select="substring-after($list_text-processing-parameters, '|')"/>
    <xsl:with-param name="text" select="$remaining-text"/>
  </xsl:call-template>
</xsl:if>
</xsl:template>
</xsl:stylesheet>

```

**/\* 17\_bis\_explosion-1-to-N-with-container-node\_RENAME.xsl \*/**

```

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.1">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <!--insert filename of namespace file-->
  <xsl:param name="file_namespaces"/>
  <!--insert XPath expression of pointed container element-->
  <xsl:param name="path"/>
  <!--insert name of element to create-->
  <xsl:param name="name_new-element"/>
  <!--insert list of names of elements to create, each one followed by character | as a separator (nb: also the last one)-->
  <xsl:param name="list_node-names"/>
  <!--insert list of values for parameter of chosen text processing function, each one followed by character | as a separator (nb: also the last one)-->
  <xsl:param name="list_text-processing-parameters"/>
  <xsl:include href="file:/D:/Progetti/Seamless/svil/it.unimore.seamless.converter/xslt/utility_text-processing_number-of-characters.xsl"/>
  <!--<xsl:strip-space elements="*" /-->
  <xsl:template match="/">
    <xsl:element name="seamless-node">
      <xsl:apply-templates select="PLACEHOLDER"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="*">
    <!--transform pointed container element-->
    <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
    <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
    <xsl:element name="{ $name_new-element }" namespace="{ $ns }">
      <xsl:copy-of select="@*" />
      <!--apply explosion action based on text node of pointed element-->
      <xsl:call-template name="processing_recursive-action">
        <xsl:with-param name="list_node-names" select="$list_node-names"/>
        <xsl:with-param name="list_text-processing-parameters" select="$list_text-processing-parameters"/>
        <xsl:with-param name="text" select="text()"/>
      </xsl:call-template>
    </xsl:element>
  </xsl:template>

```

```

</xsl:template>
<xsl:template name="processing_recursive-action">
  <xsl:param name="list_node-names"/>
  <xsl:param name="list_text-processing-parameters"/>
  <xsl:param name="text"/>
  <!--define first token of current list of node names as current node name-->
  <xsl:variable name="name_new-element" select="substring-before($list_node-names, '|')"/>
  <!--define first token of current list of parameter values as current parameter value-->
  <xsl:variable name="parameter" select="substring-before($list_text-processing-parameters, '|')"/>
  <!--create new element inside node corresponding to pointed element-->
  <!--with current node name as a name and with result of text processing function as text-->
  <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
  <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
  <xsl:element name="{ $name_new-element }" namespace="{ $ns }">
    <xsl:call-template name="text-processing">
      <xsl:with-param name="text" select="$text"/>
      <xsl:with-param name="parameter" select="$parameter"/>
      <xsl:with-param name="choice" select="'before'"/>
    </xsl:call-template>
  </xsl:element>
  <!--if there are more node names to process in current list of node names, go on with recursive action-->
  <xsl:if test="string-length(substring-after($list_node-names, '|')) != 0">
    <xsl:variable name="remaining-text">
      <xsl:call-template name="text-processing">
        <xsl:with-param name="text" select="$text"/>
        <xsl:with-param name="parameter" select="$parameter"/>
        <xsl:with-param name="choice" select="'after'"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:call-template name="processing_recursive-action">
      <xsl:with-param name="list_node-names" select="substring-after($list_node-names, '|')"/>
      <xsl:with-param name="list_text-processing-parameters" select="substring-after($list_text-processing-parameters, '|')"/>
      <xsl:with-param name="text" select="$remaining-text"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>
</xsl:stylesheet>

/* 19_fusion-N-to-1-with-container-node_separator_attribute-fusion.xml */

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <!--insert filename of namespace file-->
  <xsl:param name="file_namespaces"/>
  <!--insert XPath expression of pointed container element-->
  <xsl:param name="path"/>
  <!--insert new name for pointed container element-->
  <xsl:param name="name_new-element"/>
  <!--insert value of string of characters separating unified texts and unified values of attributes-->
  <xsl:param name="separator"/>
  <xsl:include href="utility_drop-repetitions.xml"/>

```

```

<!--<xsl:strip-space elements="*" /-->
<xsl:template match="/">
  <xsl:element name="seamless-node">
    <xsl:apply-templates select="PLACEHOLDER" />
  </xsl:element>
</xsl:template>
<xsl:template match="*">
  <!--substitute pointed element with new transformed element-->
  <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
  <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
  <xsl:element name="{ $name_new-element}" namespace="{ $ns}">
    <!--copy attributes of container element-->
    <xsl:for-each select="@*">
      <xsl:variable name="name_current-attribute" select="name()" />
      <!--fuse values of current attribute and of attributes with same name belonging to each contained child element-->
      <xsl:variable name="content">
        <xsl:value-of select="."/>
        <xsl:value-of select="$separator" />
        <xsl:for-each select=".*" *">
          <xsl:for-each select="@*[name() = $name_current-attribute]">
            <xsl:value-of select="."/>
            <xsl:value-of select="$separator" />
          </xsl:for-each>
        </xsl:for-each>
      </xsl:variable>
      <xsl:variable name="stripped-content">
        <xsl:call-template name="drop-repetitions">
          <xsl:with-param name="piece-before" />
          <xsl:with-param name="piece-after" select="$content" />
          <xsl:with-param name="separator" select="$separator" />
        </xsl:call-template>
      </xsl:variable>
      <xsl:variable name="pre2" select="substring-before($name_current-attribute, ':')"/>
      <xsl:choose>
        <!--in case of attribute name associated with namespace prefix-->
        <xsl:when test="string-length($pre2) != 0">
          <xsl:variable name="ns2" select="document($file_namespaces)//namespace[prefix = $pre2]/uri"/>
          <xsl:attribute name="{ $name_current-attribute}" namespace="{ $ns2}">
            <xsl:value-of select="$stripped-content" />
          </xsl:attribute>
        </xsl:when>
        <!--in case of attribute name not associated with namespace prefix-->
        <xsl:otherwise>
          <xsl:attribute name="{ $name_current-attribute}">
            <xsl:value-of select="$stripped-content" />
          </xsl:attribute>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
    <!--copy attributes of each contained child element-->
    <xsl:for-each select="*" *">
      <xsl:for-each select="@*" *">
        <xsl:variable name="name_current-attribute2" select="name()" />

```

```

<!--current attribute can be processed only if it has not been processed by parent node or by preceding siblings-->
<xsl:variable name="flag">
  <xsl:for-each select="../preceding-sibling::*">
    <xsl:for-each select="*[name() = $name_current-attribute2]">
      <xsl:text>1</xsl:text>
    </xsl:for-each>
  </xsl:for-each>
</xsl:variable>
<xsl:if test="not(contains($flag, '1'))">
  <!--fuse values of each occurrence of current attribute inside each contained child element-->
  <xsl:variable name="content">
    <xsl:value-of select="."/>
    <xsl:value-of select="$separator"/>
    <xsl:for-each select="../following-sibling::*">
      <xsl:for-each select="*[name() = $name_current-attribute2]">
        <xsl:value-of select="."/>
        <xsl:value-of select="$separator"/>
      </xsl:for-each>
    </xsl:for-each>
  </xsl:variable>
  <xsl:variable name="stripped-content">
    <xsl:call-template name="drop-repetitions">
      <xsl:with-param name="piece-before"/>
      <xsl:with-param name="piece-after" select="$content"/>
      <xsl:with-param name="separator" select="$separator"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:variable name="pre3" select="substring-before($name_current-attribute2, ':')"/>
  <xsl:choose>
    <!--in case of attribute name associated with namespace prefix-->
    <xsl:when test="string-length($pre3) != 0">
      <xsl:variable name="ns3" select="document($file_namespaces)//namespace[prefix = $pre3]/uri"/>
      <xsl:attribute name="{ $name_current-attribute2 }" namespace="{ $ns3 }">
        <xsl:value-of select="$stripped-content"/>
      </xsl:attribute>
    </xsl:when>
    <!--in case of attribute name not associated with namespace prefix-->
    <xsl:otherwise>
      <xsl:attribute name="{ $name_current-attribute2 }">
        <xsl:value-of select="$stripped-content"/>
      </xsl:attribute>
    </xsl:otherwise>
  </xsl:choose>
</xsl:if>
</xsl:for-each>
<!--node corresponding to pointed element does not contain a text node because it is not a "leaf"-->
<!--copy value of text nodes of each contained child element-->

```

```

        <xsl:for-each select="*">
            <xsl:value-of select="text()"/>
            <xsl:value-of select="$separator"/>
        </xsl:for-each>
    </xsl:element>
</xsl:template>
</xsl:stylesheet>

/* 20_fusion-2-to-1-without-container-node_arithmetic-operation_attribute-fusion.xsl */

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="xml" omit-xml-declaration="yes"/>
    <!--insert filename of namespace file-->
    <xsl:param name="file_namespaces"/>
    <!--insert XPath expression of first pointed element to process-->
    <!--ATTENTION: pointed element must have single occurrence-->
    <xsl:param name="path-1"/>
    <!--insert XPath expression of second pointed element to process-->
    <!--ATTENTION: pointed element must have single occurrence-->
    <xsl:param name="path-2"/>
    <!--insert name of element to create-->
    <xsl:param name="name_new-element"/>
    <!--insert symbol of arithmetic operation to execute on texts of pointed elements-->
    <xsl:param name="operator"/>
    <!--insert number of decimals of result to display-->
    <!--default value = 2-->
    <xsl:param name="decimals">2</xsl:param>
    <!--insert value of string of characters separating unified texts and unified values of attributes-->
    <xsl:param name="separator"/>
    <xsl:include href="utility_get-name.xsl"/>
    <xsl:include href="utility_copy-descendants.xsl"/>
    <xsl:include href="utility_get-common-path.xsl"/>
    <xsl:include href="utility_get-last-token.xsl"/>
    <xsl:include href="utility_search-A_version-2.xsl"/>
    <xsl:include href="utility_search-B.xsl"/>
    <xsl:include href="utility_calculate-expression.xsl"/>
    <xsl:include href="utility_add-decimals.xsl"/>
    <xsl:include href="utility_fuse-attributes.xsl"/>
    <xsl:include href="utility_drop-repetitions.xsl"/>
    <!--<xsl:strip-space elements="*" />-->
    <xsl:template match="/">
        <xsl:element name="seamless-node">
            <xsl:apply-templates select="PLACEHOLDER"/>
        </xsl:element>
    </xsl:template>
    <xsl:template match="*">
        <xsl:copy>
            <xsl:copy-of select="@*" />
            <xsl:call-template name="processing_action" />
        </xsl:copy>
    </xsl:template>

```

```

<xsl:template name="processing_action">
  <xsl:variable name="path_common">
    <xsl:call-template name="get-common-path">
      <xsl:with-param name="path-1" select="$path-1"/>
      <xsl:with-param name="path-2" select="$path-2"/>
    </xsl:call-template>
  </xsl:variable>
  <!--search node corresponding to destination element-->
  <xsl:call-template name="search_navigation-A">
    <xsl:with-param name="path-1" select="substring-after($path-1, $path_common)"/>
    <xsl:with-param name="path-2" select="substring-after($path-2, $path_common)"/>
  </xsl:call-template>
  <!--if destination element does not exist, fuse only occurrences of the other element-->
  <xsl:variable name="name_current-element-1">
    <xsl:call-template name="get-last-token">
      <xsl:with-param name="path" select="$path-1"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:variable name="name_current-element-2">
    <xsl:call-template name="get-last-token">
      <xsl:with-param name="path" select="$path-2"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:if test="not(boolean(*[name() = $name_current-element-1]))">
    <xsl:for-each select="*[name() = $name_current-element-2]">
      <!--substitute pointed element with new transformed element-->
      <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
      <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
      <xsl:element name="{ $name_new-element }" namespace="{ $ns }">
        <xsl:copy-of select="@*" />
        <!--set conventional format to display result-->
        <xsl:variable name="format-string">
          <xsl:text>#</xsl:text>
          <xsl:if test="$decimals != 0">
            <xsl:text>.</xsl:text>
            <xsl:call-template name="add-decimals">
              <xsl:with-param name="counter" select="$decimals"/>
            </xsl:call-template>
          </xsl:if>
        </xsl:variable>
        <xsl:value-of select="format-number(text(), $format-string)"/>
      </xsl:element>
    </xsl:for-each>
  </xsl:if>
</xsl:template>
<xsl:template name="processing_action-A">
  <xsl:variable name="path_common">
    <xsl:call-template name="get-common-path">
      <xsl:with-param name="path-1" select="$path-1"/>
      <xsl:with-param name="path-2" select="$path-2"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:call-template name="processing_action-B"/>

```

```

</xsl:template>
<xsl:template name="processing_action-B">
  <xsl:variable name="name_current-element-1">
    <xsl:call-template name="get-last-token">
      <xsl:with-param name="path" select="$path-1"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:variable name="name_current-element-2">
    <xsl:call-template name="get-last-token">
      <xsl:with-param name="path" select="$path-2"/>
    </xsl:call-template>
  </xsl:variable>
  <!--substitute first pointed element with new transformed element-->
  <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
  <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
  <xsl:element name="{ $name_new-element}" namespace="{ $ns}">
    <!--copy attributes of both pointed elements-->
    <xsl:for-each select="@*">
      <xsl:variable name="name_current-attribute" select="name()"/>
      <xsl:variable name="content">
        <xsl:value-of select="."/>
        <xsl:value-of select="$separator"/>
        <xsl:for-each select="../*[name() = $name_current-element-2]">
          <xsl:for-each select="@*[name() = $name_current-attribute]">
            <xsl:value-of select="."/>
            <xsl:value-of select="$separator"/>
          </xsl:for-each>
        </xsl:for-each>
      </xsl:variable>
      <xsl:variable name="stripped-content">
        <xsl:call-template name="drop-repetitions">
          <xsl:with-param name="piece-before"/>
          <xsl:with-param name="piece-after" select="$content"/>
          <xsl:with-param name="separator" select="$separator"/>
        </xsl:call-template>
      </xsl:variable>
      <xsl:variable name="pre2" select="substring-before($name_current-attribute, ':')"/>
      <xsl:choose>
        <!--in case of attribute name associated with namespace prefix-->
        <xsl:when test="string-length($pre2) != 0">
          <xsl:variable name="ns2" select="document($file_namespaces)//namespace[prefix = $pre2]/uri"/>
          <xsl:attribute name="{ $name_current-attribute}" namespace="{ $ns2}">
            <xsl:value-of select="$stripped-content"/>
          </xsl:attribute>
        </xsl:when>
        <!--in case of attribute name not associated with namespace prefix-->
        <xsl:otherwise>
          <xsl:attribute name="{ $name_current-attribute}">
            <xsl:value-of select="$stripped-content"/>
          </xsl:attribute>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </xsl:element>

```

```

<xsl:for-each select="../*[name() = $name_current-element-2]">
  <xsl:for-each select="@*">
    <xsl:variable name="name_current-attribute2" select="name()"/>
    <xsl:if test="not(boolean(../*[name() = $name_current-element-1]/@*[name() = $name_current-attribute2]))">
      <xsl:variable name="content2">
        <xsl:value-of select="."/>
        <xsl:value-of select="$separator"/>
      </xsl:variable>
      <xsl:variable name="pre3" select="substring-before($name_current-attribute2, ':')"/>
      <xsl:choose>
        <!--in case of attribute name associated with namespace prefix-->
        <xsl:when test="string-length($pre3) != 0">
          <xsl:variable name="ns3" select="document($file_namespaces)//namespace[prefix = $pre3]/uri"/>
          <xsl:attribute name="{ $name_current-attribute2 }" namespace="{ $ns3 }">
            <xsl:value-of select="$content2"/>
          </xsl:attribute>
        </xsl:when>
        <!--in case of attribute name not associated with namespace prefix-->
        <xsl:otherwise>
          <xsl:attribute name="{ $name_current-attribute2 }">
            <xsl:value-of select="$content2"/>
          </xsl:attribute>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:if>
  </xsl:for-each>
</xsl:for-each>
<!--process value of text nodes of both pointed elements by means of chosen operation-->
<xsl:variable name="current-value-2">
  <xsl:value-of select="../*[name() = $name_current-element-2]/text()"/>
</xsl:variable>
<xsl:variable name="result">
  <xsl:call-template name="calculate-expression">
    <xsl:with-param name="op1" select="text()"/>
    <xsl:with-param name="op2" select="$current-value-2"/>
  </xsl:call-template>
</xsl:variable>
<!--set conventional format to display result-->
<xsl:variable name="format-string">
  <xsl:text>#</xsl:text>
  <xsl:if test="$decimals != 0">
    <xsl:text>.</xsl:text>
    <xsl:call-template name="add-decimals">
      <xsl:with-param name="counter" select="$decimals"/>
    </xsl:call-template>
  </xsl:if>
</xsl:variable>
<xsl:value-of select="format-number($result, $format-string)"/>
</xsl:element>
</xsl:template>
</xsl:stylesheet>

/* 20_fusion-2-to-1-without-container-node_separator_attribute-fusion.xsl */

```

```

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <!--insert filename of namespace file-->
  <xsl:param name="file_namespaces"/>
  <!--insert XPath expression of first pointed element to process-->
  <xsl:param name="path-1"/>
  <!--insert XPath expression of second pointed element to process-->
  <xsl:param name="path-2"/>
  <!--insert name of element to create-->
  <xsl:param name="name_new-element"/>
  <!--insert value of string of characters separating unified texts and unified values of attributes-->
  <xsl:param name="separator"/>
  <xsl:include href="utility_get-name.xsl"/>
  <xsl:include href="utility_copy-descendants.xsl"/>
  <xsl:include href="utility_get-common-path.xsl"/>
  <xsl:include href="utility_get-last-token.xsl"/>
  <xsl:include href="utility_search-A_version-2.xsl"/>
  <xsl:include href="utility_search-B.xsl"/>
  <xsl:include href="utility_fuse-attributes.xsl"/>
  <xsl:include href="utility_drop-repetitions.xsl"/>
  <!--<xsl:strip-space elements="*" /-->
  <xsl:template match="/">
    <xsl:element name="seamless-node">
      <xsl:apply-templates select="PLACEHOLDER"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="*">
    <xsl:copy>
      <xsl:copy-of select="@*" />
      <xsl:call-template name="processing_action" />
    </xsl:copy>
  </xsl:template>
  <xsl:template name="processing_action">
    <xsl:variable name="path_common">
      <xsl:call-template name="get-common-path">
        <xsl:with-param name="path-1" select="$path-1"/>
        <xsl:with-param name="path-2" select="$path-2"/>
      </xsl:call-template>
    </xsl:variable>
    <!--search node corresponding to destination element-->
    <xsl:call-template name="search_navigation-A">
      <xsl:with-param name="path-1" select="substring-after($path-1, $path_common)"/>
      <xsl:with-param name="path-2" select="substring-after($path-2, $path_common)"/>
    </xsl:call-template>
    <!--if destination element does not exist, fuse only occurrences of the other element-->
    <xsl:variable name="name_current-element-1">
      <xsl:call-template name="get-last-token">
        <xsl:with-param name="path" select="$path-1"/>
      </xsl:call-template>
    </xsl:variable>
  </xsl:template>

```

```

<xsl:variable name="name_current-element-2">
  <xsl:call-template name="get-last-token">
    <xsl:with-param name="path" select="$path-2"/>
  </xsl:call-template>
</xsl:variable>
<xsl:if test="not(boolean(*[name() = $name_current-element-1]))">
  <xsl:for-each select="*[name() = $name_current-element-2][position() = 1]">
    <!-- substitute pointed element with new transformed element-->
    <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
    <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
    <xsl:element name="{ $name_new-element }" namespace="{ $ns }">
      <!-- copy attributes of each occurrence of pointed element-->
      <xsl:for-each select="@*">
        <xsl:call-template name="fuse-attributes">
          <xsl:with-param name="name_current-element" select="$name_current-element-2"/>
          <xsl:with-param name="separator" select="$separator"/>
        </xsl:call-template>
      </xsl:for-each>
      <xsl:for-each select="following-sibling::*[name() = $name_current-element-2]">
        <xsl:for-each select="@*">
          <xsl:variable name="name_current-attribute" select="name()"/>
          <xsl:variable name="flag">
            <xsl:for-each select="..preceding-sibling::*[name() = $name_current-element-2]">
              <xsl:for-each select="@*[name() = $name_current-attribute]">
                <xsl:text>1</xsl:text>
              </xsl:for-each>
            </xsl:for-each>
          </xsl:variable>
          <xsl:if test="not(contains($flag, '1'))">
            <xsl:call-template name="fuse-attributes">
              <xsl:with-param name="name_current-element" select="$name_current-element-2"/>
              <xsl:with-param name="separator" select="$separator"/>
            </xsl:call-template>
          </xsl:if>
        </xsl:for-each>
      </xsl:for-each>
      <!-- copy value of text nodes of each occurrence of pointed element-->
      <xsl:value-of select="text()"/>
      <xsl:value-of select="$separator"/>
      <xsl:for-each select="following-sibling::*[name() = $name_current-element-2]">
        <xsl:value-of select="text()"/>
        <xsl:value-of select="$separator"/>
      </xsl:for-each>
    </xsl:element>
  </xsl:for-each>
</xsl:if>
</xsl:template>
<xsl:template name="processing_action-A">
  <xsl:variable name="path_common">
    <xsl:call-template name="get-common-path">
      <xsl:with-param name="path-1" select="$path-1"/>
      <xsl:with-param name="path-2" select="$path-2"/>
    </xsl:call-template>
  </xsl:variable>

```

```

</xsl:variable>
<!--processing starts after selecting the first sibling of the set of pointed container elements-->
<xsl:variable name="name_last-token">
  <xsl:call-template name="get-last-token">
    <xsl:with-param name="path" select="$path-1"/>
  </xsl:call-template>
</xsl:variable>
<xsl:if test="not(boolean(preceding-sibling::*[name() = $name_last-token]))">
  <xsl:call-template name="processing_action-B"/>
</xsl:if>
</xsl:template>
<xsl:template name="processing_action-B">
  <xsl:variable name="name_current-element-1">
    <xsl:call-template name="get-last-token">
      <xsl:with-param name="path" select="$path-1"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:variable name="name_current-element-2">
    <xsl:call-template name="get-last-token">
      <xsl:with-param name="path" select="$path-2"/>
    </xsl:call-template>
  </xsl:variable>
  <!--substitute first pointed element with new transformed element-->
  <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
  <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
  <xsl:element name="{ $name_new-element}" namespace="{ $ns}">
    <!--copy attributes of each occurrence of both pointed elements-->
    <!--deal with attributes of current element-->
    <xsl:for-each select="@*">
      <xsl:variable name="name_current-attribute" select="name()"/>
      <xsl:variable name="content">
        <xsl:value-of select="."/>
        <xsl:value-of select="$separator"/>
        <xsl:for-each select="./following-sibling::*[name() = $name_current-element-1]">
          <xsl:for-each select="@*[name() = $name_current-attribute]">
            <xsl:value-of select="."/>
            <xsl:value-of select="$separator"/>
          </xsl:for-each>
        </xsl:for-each>
        <xsl:for-each select="./../*[name() = $name_current-element-2]">
          <xsl:for-each select="@*[name() = $name_current-attribute]">
            <xsl:value-of select="."/>
            <xsl:value-of select="$separator"/>
          </xsl:for-each>
        </xsl:for-each>
      </xsl:variable>
      <xsl:variable name="stripped-content">
        <xsl:call-template name="drop-repetitions">
          <xsl:with-param name="piece-before"/>
          <xsl:with-param name="piece-after" select="$content"/>
          <xsl:with-param name="separator" select="$separator"/>
        </xsl:call-template>
      </xsl:variable>

```

```

<xsl:variable name="pre2" select="substring-before($name_current-attribute, ':')"/>
<xsl:choose>
  <!--in case of attribute name associated with namespace prefix-->
  <xsl:when test="string-length($pre2) != 0">
    <xsl:variable name="ns2" select="document($file_namespaces)//namespace[prefix = $pre2]/uri"/>
    <xsl:attribute name="{ $name_current-attribute}" namespace="{ $ns2}">
      <xsl:value-of select="$stripped-content"/>
    </xsl:attribute>
  </xsl:when>
  <!--in case of attribute name not associated with namespace prefix-->
  <xsl:otherwise>
    <xsl:attribute name="{ $name_current-attribute}">
      <xsl:value-of select="$stripped-content"/>
    </xsl:attribute>
  </xsl:otherwise>
</xsl:choose>
</xsl:for-each>
<!--deal with attributes of other occurrences of current element-->
<xsl:for-each select="following-sibling::*[name() = $name_current-element-1]">
  <xsl:for-each select="@*">
    <xsl:variable name="name_current-attribute2" select="name()"/>
    <!--current attribute can be processed only if it has not been processed by preceding siblings of pointed elements-->
    <xsl:variable name="flag">
      <xsl:for-each select="..preceding-sibling::*[name() = $name_current-element-1]">
        <xsl:for-each select="@*[name() = $name_current-attribute2]">
          <xsl:text>1</xsl:text>
        </xsl:for-each>
      </xsl:for-each>
    </xsl:variable>
    <xsl:if test="not(contains($flag, 1))">
      <xsl:variable name="content2">
        <xsl:value-of select="."/>
        <xsl:value-of select="$separator"/>
        <xsl:for-each select="..following-sibling::*[name() = $name_current-element-1]">
          <xsl:for-each select="@*[name() = $name_current-attribute2]">
            <xsl:value-of select="."/>
            <xsl:value-of select="$separator"/>
          </xsl:for-each>
        </xsl:for-each>
        <xsl:for-each select="..../*[name() = $name_current-element-2]">
          <xsl:for-each select="@*[name() = $name_current-attribute2]">
            <xsl:value-of select="."/>
            <xsl:value-of select="$separator"/>
          </xsl:for-each>
        </xsl:for-each>
      </xsl:variable>
      <xsl:variable name="stripped-content2">
        <xsl:call-template name="drop-repetitions">
          <xsl:with-param name="piece-before"/>
          <xsl:with-param name="piece-after" select="$content2"/>
          <xsl:with-param name="separator" select="$separator"/>
        </xsl:call-template>
      </xsl:variable>

```

```

<xsl:variable name="pre3" select="substring-before($name_current-attribute2, ':')"/>
<xsl:choose>
  <!--in case of attribute name associated with namespace prefix-->
  <xsl:when test="string-length($pre3) != 0">
    <xsl:variable name="ns3" select="document($file_namespaces)//namespace[prefix = $pre3]/uri"/>
    <xsl:attribute name="{ $name_current-attribute2}" namespace="{ $ns3}">
      <xsl:value-of select="$stripped-content2"/>
    </xsl:attribute>
  </xsl:when>
  <!--in case of attribute name not associated with namespace prefix-->
  <xsl:otherwise>
    <xsl:attribute name="{ $name_current-attribute2}">
      <xsl:value-of select="$stripped-content2"/>
    </xsl:attribute>
  </xsl:otherwise>
</xsl:choose>
</xsl:if>
</xsl:for-each>
</xsl:for-each>
<!--deal with attributes of occurrences of other element-->
<xsl:for-each select="../*[name() = $name_current-element-2]">
  <xsl:for-each select="@*">
    <xsl:variable name="name_current-attribute3" select="name()"/>
    <!--current attribute can be processed only if it has not been processed by preceding siblings of pointed elements-->
    <xsl:variable name="flag2">
      <xsl:for-each select="../*[name() = $name_current-element-1]">
        <xsl:for-each select="@*[name() = $name_current-attribute3]">
          <xsl:text>1</xsl:text>
        </xsl:for-each>
      </xsl:for-each>
      <xsl:for-each select="..preceding-sibling::*[name() = $name_current-element-2]">
        <xsl:for-each select="@*[name() = $name_current-attribute3]">
          <xsl:text>1</xsl:text>
        </xsl:for-each>
      </xsl:for-each>
    </xsl:variable>
    <xsl:if test="not(contains($flag2, 1))">
      <xsl:variable name="content3">
        <xsl:value-of select="."/>
        <xsl:value-of select="$separator"/>
        <xsl:for-each select="..following-sibling::*[name() = $name_current-element-2]">
          <xsl:for-each select="@*[name() = $name_current-attribute3]">
            <xsl:value-of select="."/>
            <xsl:value-of select="$separator"/>
          </xsl:for-each>
        </xsl:for-each>
      </xsl:variable>
      <xsl:variable name="stripped-content3">
        <xsl:call-template name="drop-repetitions">
          <xsl:with-param name="piece-before"/>
          <xsl:with-param name="piece-after" select="$content3"/>
          <xsl:with-param name="separator" select="$separator"/>
        </xsl:call-template>

```

```

</xsl:variable>
<xsl:variable name="pre4" select="substring-before($name_current-attribute3, ':')"/>
<xsl:choose>
  <!--in case of attribute name associated with namespace prefix-->
  <xsl:when test="string-length($pre4) != 0">
    <xsl:variable name="ns4" select="document($file_namespaces)//namespace[prefix = $pre4]/uri"/>
    <xsl:attribute name="{ $name_current-attribute3}" namespace="{ $ns4}">
      <xsl:value-of select="$stripped-content3"/>
    </xsl:attribute>
  </xsl:when>
  <!--in case of attribute name not associated with namespace prefix-->
  <xsl:otherwise>
    <xsl:attribute name="{ $name_current-attribute3}">
      <xsl:value-of select="$stripped-content3"/>
    </xsl:attribute>
  </xsl:otherwise>
</xsl:choose>
</xsl:if>
</xsl:for-each>
</xsl:for-each>
<!--copy value of text nodes of each occurrence of both pointed elements-->
<xsl:for-each select="../*">
  <xsl:choose>
    <xsl:when test="name() = $name_current-element-1">
      <xsl:value-of select="text()"/>
      <xsl:value-of select="$separator"/>
    </xsl:when>
    <xsl:when test="name() = $name_current-element-2">
      <xsl:value-of select="text()"/>
      <xsl:value-of select="$separator"/>
    </xsl:when>
  </xsl:choose>
</xsl:for-each>
</xsl:element>
</xsl:template>
</xsl:stylesheet>

```

**/\* 21\_fusion-multiple-occurrences\_summation-product\_attribute-fusion.xsl \*/**

<?xml version="1.0" encoding="UTF-8"?>

```

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <!--insert filename of namespace file-->
  <xsl:param name="file_namespaces"/>
  <!--insert XPath expression of pointed element with multiple occurrences-->
  <xsl:param name="path"/>
  <!--insert name of element to create-->
  <xsl:param name="name_new-element"/>
  <!--insert name of operation to execute on text of multiple occurrences of pointed element-->
  <!--choice is between "summation" and "product"-->
  <xsl:param name="choice"/>
  <!--insert number of decimals of result to display-->

```

```

<!--default value = 2-->
<xsl:param name="decimals">2</xsl:param>
<!--insert value of string of characters separating attribute values-->
<xsl:param name="separator"/>
<xsl:include href="utility_get-name.xsl"/>
<xsl:include href="utility_get-last-token.xsl"/>
<xsl:include href="utility_add-decimals.xsl"/>
<xsl:include href="utility_drop-repetitions.xsl"/>
<xsl:include href="utility_fuse-attributes.xsl"/>
<!--<xsl:strip-space elements="*" /-->
<xsl:template match="/">
  <xsl:element name="seamless-node">
    <xsl:apply-templates select="PLACEHOLDER"/>
  </xsl:element>
</xsl:template>
<xsl:template match="*">
  <!--processing starts after selecting the first sibling of the set of pointed elements-->
  <xsl:variable name="name_last-token">
    <xsl:call-template name="get-last-token">
      <xsl:with-param name="path" select="$path"/>
    </xsl:call-template>
  </xsl:variable>
  <xsl:if test="not(boolean(preceding-sibling::*[name() = $name_last-token]))">
    <xsl:call-template name="processing_action"/>
  </xsl:if>
</xsl:template>
<xsl:template name="processing_action">
  <xsl:variable name="name_current-element">
    <xsl:call-template name="get-last-token">
      <xsl:with-param name="path" select="$path"/>
    </xsl:call-template>
  </xsl:variable>
  <!--substitute pointed element with new transformed element-->
  <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
  <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
  <xsl:element name="{ $name_new-element }" namespace="{ $ns }">
    <!--copy attributes of each occurrence of pointed element-->
    <xsl:for-each select="@*">
      <xsl:call-template name="fuse-attributes">
        <xsl:with-param name="name_current-element" select="$name_current-element"/>
        <xsl:with-param name="separator" select="$separator"/>
      </xsl:call-template>
    </xsl:for-each>
    <xsl:for-each select="following-sibling::*[name() = $name_current-element]">
      <xsl:for-each select="@*">
        <xsl:variable name="name_current-attribute1" select="name()"/>
        <xsl:variable name="flag">
          <xsl:for-each select="..preceding-sibling::*[name() = $name_current-element]">
            <xsl:for-each select="@*[name() = $name_current-attribute1]">
              <xsl:text>1</xsl:text>
            </xsl:for-each>
          </xsl:for-each>
        </xsl:variable>
      </xsl:for-each>
    </xsl:element>
  </xsl:variable>

```

```

                <xsl:if test="not(contains($flag, '1'))">
                    <xsl:call-template name="fuse-attributes">
                        <xsl:with-param name="name_current-element" select="$name_current-element"/>
                        <xsl:with-param name="separator" select="$separator"/>
                    </xsl:call-template>
                </xsl:if>
            </xsl:for-each>
        </xsl:for-each>
        <!--process value of text nodes of each occurrence of pointed element by means of chosen operation-->
        <xsl:variable name="result_final">
            <xsl:choose>
                <!--in case of multiple occurrences of pointed element-->
                <xsl:when test="following-sibling::*[name() = $name_current-element]">
                    <xsl:call-template name="processing_recursive-action">
                        <xsl:with-param name="nome" select="$name_current-element"/>
                        <xsl:with-param name="operand_1" select="text()"/>
                    </xsl:call-template>
                </xsl:when>
                <!--in case of only one occurrence of pointed element-->
                <xsl:otherwise>
                    <xsl:value-of select="text()"/>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:variable>
        <!--set conventional format to display result-->
        <xsl:variable name="format-string">
            <xsl:text>#</xsl:text>
            <xsl:if test="$decimals != 0">
                <xsl:text>.</xsl:text>
                <xsl:call-template name="add-decimals">
                    <xsl:with-param name="counter" select="$decimals"/>
                </xsl:call-template>
            </xsl:if>
        </xsl:variable>
        <xsl:value-of select="format-number($result_final, $format-string)"/>
    </xsl:element>
</xsl:template>
<xsl:template name="processing_recursive-action">
    <xsl:param name="nome"/>
    <xsl:param name="operand_1"/>
    <!--move to next sibling of the set of pointed elements-->
    <xsl:for-each select="following-sibling::*[name() = $nome][position() = 1]">
        <xsl:variable name="operand_2" select="text()"/>
        <xsl:variable name="result_intermediate">
            <xsl:choose>
                <xsl:when test="$choice = 'summation'">
                    <xsl:value-of select="$operand_1 + $operand_2"/>
                </xsl:when>
                <xsl:when test="$choice = 'product'">
                    <xsl:value-of select="$operand_1 * $operand_2"/>
                </xsl:when>
            </xsl:choose>
        </xsl:variable>
    </xsl:for-each>

```

```

        <xsl:choose>
          <!--if there are more node siblings of the set of pointed elements to be processed, go on with recursive action-->
          <xsl:when test="following-sibling::*[name() = $nome]">
            <xsl:call-template name="processing_recursive-action">
              <xsl:with-param name="nome" select="$nome"/>
              <xsl:with-param name="operand_1" select="$result_intermediate"/>
            </xsl:call-template>
          </xsl:when>
          <xsl:otherwise>
            <xsl:value-of select="$result_intermediate"/>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each>
    </xsl:template>
  </xsl:stylesheet>

```

**/\* 21\_fusion-multiple-occurrences\_separator\_attribute-fusion.xsl \*/**

```

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <!--insert filename of namespace file-->
  <xsl:param name="file_namespaces"/>
  <!--insert XPath expression of pointed element with multiple occurrences-->
  <xsl:param name="path"/>
  <!--insert name of element to create-->
  <xsl:param name="name_new-element"/>
  <!--insert value of string of characters separating unified texts and attribute values-->
  <xsl:param name="separator"/>
  <xsl:include href="utility_get-name.xsl"/>
  <xsl:include href="utility_get-last-token.xsl"/>
  <xsl:include href="utility_drop-repetitions.xsl"/>
  <xsl:include href="utility_fuse-attributes.xsl"/>
  <!--<xsl:strip-space elements="*" />-->
  <xsl:template match="/">
    <xsl:element name="seamless-node">
      <xsl:apply-templates select="PLACEHOLDER"/>
    </xsl:element>
  </xsl:template>
  <xsl:template match="*">
    <!--processing starts after selecting the first sibling of the set of pointed elements-->
    <xsl:variable name="name_last-token">
      <xsl:call-template name="get-last-token">
        <xsl:with-param name="path" select="$path"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:if test="not(boolean(preceding-sibling::*[name() = $name_last-token]))">
      <xsl:call-template name="processing_action"/>
    </xsl:if>
  </xsl:template>
  <xsl:template name="processing_action">
    <xsl:variable name="name_current-element">

```

```

        <xsl:call-template name="get-last-token">
            <xsl:with-param name="path" select="$path"/>
        </xsl:call-template>
    </xsl:variable>
    <!--substitute pointed element with new transformed element-->
    <xsl:variable name="pre" select="substring-before($name_new-element, ':')"/>
    <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
    <xsl:element name="{ $name_new-element}" namespace="{ $ns}">
        <!--copy attributes of each occurrence of pointed element-->
        <xsl:for-each select="@*">
            <xsl:call-template name="fuse-attributes">
                <xsl:with-param name="name_current-element" select="$name_current-element"/>
                <xsl:with-param name="separator" select="$separator"/>
            </xsl:call-template>
        </xsl:for-each>
        <xsl:for-each select="following-sibling::*[name() = $name_current-element]">
            <xsl:for-each select="@*">
                <xsl:variable name="name_current-attribute1" select="name()"/>
                <xsl:variable name="flag">
                    <xsl:for-each select="..preceding-sibling::*[name() = $name_current-element]">
                        <xsl:for-each select="@*[name() = $name_current-attribute1]">
                            <xsl:text>1</xsl:text>
                        </xsl:for-each>
                    </xsl:for-each>
                </xsl:variable>
                <xsl:if test="not(contains($flag, '1'))">
                    <xsl:call-template name="fuse-attributes">
                        <xsl:with-param name="name_current-element" select="$name_current-element"/>
                        <xsl:with-param name="separator" select="$separator"/>
                    </xsl:call-template>
                </xsl:if>
            </xsl:for-each>
        </xsl:for-each>
        <!--copy value of text nodes of each occurrence of pointed element-->
        <xsl:value-of select="text()"/>
        <xsl:value-of select="$separator"/>
        <xsl:for-each select="following-sibling::*[name() = $name_current-element]">
            <xsl:value-of select="text()"/>
            <xsl:value-of select="$separator"/>
        </xsl:for-each>
    </xsl:element>
</xsl:template>
</xsl:stylesheet>

/* 22_transform-root.xsl */

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <!--insert filename of destination file containing root element to copy-->
    <xsl:param name="file_destination"/>
    <!--insert filename of source file containing current document-->
    <!--default value = CURRENT (in order to correctly interact with translator)-->

```

```

<xsl:param name="file_source">CURRENT</xsl:param>
<!--insert filename of source namespace file-->
<xsl:param name="file_source-namespaces"/>
<!--insert filename of destination namespace file-->
<xsl:param name="file_destination-namespaces"/>
<!--<xsl:strip-space elements="*" /-->
<!--list each source namespace different from each destination namespace-->
<xsl:variable name="list_namespaces">
  <xsl:for-each select="document($file_source-namespaces)//namespace/prefix">
    <xsl:variable name="source-namespace" select="."/>
    <xsl:variable name="flag">
      <xsl:for-each select="document($file_destination-namespaces)//namespace[prefix = $source-namespace]">
        <xsl:text>1</xsl:text>
      </xsl:for-each>
    </xsl:variable>
    <xsl:if test="$flag != '1'">
      <xsl:value-of select="$source-namespace"/>
      <xsl:text>|</xsl:text>
    </xsl:if>
  </xsl:for-each>
</xsl:variable>
<xsl:variable name="destination-root" select="document($file_destination)"/>
<xsl:variable name="source-root" select="document($file_source)"/>
<xsl:template match="/">
  <!--copy root element of document corresponding to destination file-->
  <xsl:for-each select="$destination-root/*">
    <xsl:copy>
      <!--add namespaces of document corresponding to source file not present in document corresponding to destination file-->
      <xsl:call-template name="copy-namespaces">
        <xsl:with-param name="list_namespaces" select="$list_namespaces"/>
      </xsl:call-template>
      <xsl:copy-of select="$destination-root/*/*"/>
      <!--append each node (different from root) of document corresponding to source file-->
      <xsl:apply-templates select="$source-root/*/*"/>
    </xsl:copy>
  </xsl:for-each>
</xsl:template>
<xsl:template match="node()">
  <xsl:copy>
    <xsl:copy-of select="@*"/>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
<xsl:template name="copy-namespaces">
  <xsl:param name="list_namespaces"/>
  <xsl:variable name="token" select="substring-before($list_namespaces, '|')"/>
  <xsl:if test="string-length($token) != 0">
    <xsl:variable name="ns-to-copy" select="document($file_source-namespaces)//namespace[prefix = $token]/uri"/>
    <xsl:for-each select="$source-root/*/namespace:*">
      <xsl:variable name="content" select="."/>
      <xsl:if test="$content = $ns-to-copy">
        <xsl:copy-of select="."/>
      </xsl:if>
    </xsl:for-each>
  </xsl:if>
</xsl:template>

```

```

        </xsl:for-each>
        <xsl:if test="string-length(substring-after($list_namespaces, '|')) != 0">
          <xsl:call-template name="copy-namespaces">
            <xsl:with-param name="list_namespaces" select="substring-after($list_namespaces, '|')"/>
          </xsl:call-template>
        </xsl:if>
      </xsl:if>
    </xsl:template>
  </xsl:stylesheet>

```

**/\* 22\_transform-root\_cleanup.xsl \*/**

```

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <!--<xsl:strip-space elements="*" /-->
  <xsl:template match="*">
    <!--process each element of current document-->
    <xsl:element name="{name()}">
      <!--remove each reference to empty namespaces-->
      <!--ATTENTION: copy of destination namespaces inside root element of this stylesheet is required-->
      <xsl:for-each select="namespace::*">
        <xsl:variable name="current-namespace" select="."/>
        <xsl:if test="$current-namespace != ''">
          <xsl:copy-of select="."/>
        </xsl:if>
      </xsl:for-each>
      <!--copy current element structure-->
      <xsl:copy-of select="@*" />
      <xsl:copy-of select="text()" />
      <xsl:apply-templates select="*" />
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>

```

**/\* 23\_rename-element.xsl \*/**

```

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="xml" omit-xml-declaration="yes" />
  <!--insert filename of namespace file-->
  <xsl:param name="file_namespaces" />
  <!--insert XPath expression of pointed element-->
  <xsl:param name="path" />
  <!--insert new name of pointed element-->
  <xsl:param name="new-name" />
  <xsl:include href="utility_copy-descendants.xsl" />
  <!--<xsl:strip-space elements="*" /-->
  <xsl:template match="/">
    <xsl:element name="seamless-node">
      <xsl:apply-templates select="PLACEHOLDER" />
    </xsl:element>
  </xsl:template>

```

```

</xsl:template>
<xsl:template match="*">
  <xsl:variable name="pre" select="substring-before($new-name, ':')"/>
  <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
  <xsl:element name="{ $new-name}" namespace="{ $ns}">
    <xsl:copy-of select="@*" />
    <xsl:call-template name="copy-descendants" />
  </xsl:element>
</xsl:template>
</xsl:stylesheet>

/* 25_final-cleanup.xsl */

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <!--insert filename of source namespace file-->
  <xsl:param name="file_source-namespaces"/>
  <!--insert filename of destination namespace file-->
  <xsl:param name="file_destination-namespaces"/>
  <!--<xsl:strip-space elements="*" />-->
  <!--global variable to list namespaces whose references must be removed from current document-->
  <xsl:variable name="list_namespaces">
    <!--each reference to a source namespace different from each destination namespace must be removed from current document-->
    <xsl:for-each select="document($file_source-namespaces)//namespace/prefix">
      <xsl:variable name="source-namespace" select="."/>
      <xsl:variable name="flag">
        <xsl:for-each select="document($file_destination-namespaces)//namespace[prefix = $source-namespace]">
          <xsl:text>1</xsl:text>
        </xsl:for-each>
      </xsl:variable>
      <!--if the flag is false, each reference to current source namespace must be removed from current document-->
      <xsl:if test="$flag != '1'">
        <xsl:value-of select="$source-namespace"/>
        <xsl:text>|</xsl:text>
      </xsl:if>
    </xsl:for-each>
    <!--deal with presence of different default namespaces in both namespace files-->
    <xsl:if test="document($file_source-namespaces)//namespace[prefix = '']">
      <xsl:if test="document($file_destination-namespaces)//namespace[prefix = '']">
        <xsl:text>|</xsl:text>
      </xsl:if>
    </xsl:if>
  </xsl:variable>
  <xsl:template match="*">
    <!--process each element of current document-->
    <xsl:element name="{name()}">
      <!--remove each reference to a source namespace-->
      <!--ATTENTION: copy of destination namespaces inside root element of this stylesheet is required-->
      <xsl:for-each select="namespace::*">
        <xsl:variable name="current-namespace" select="."/>
        <xsl:variable name="flag">
          <xsl:call-template name="compare-namespaces">

```

```

                <xsl:with-param name="list_to-compare" select="$list_namespaces"/>
                <xsl:with-param name="comparison-object" select="$current-namespace"/>
            </xsl:call-template>
        </xsl:variable>
        <!--if the flag is false, copy current reference to a namespace-->
        <!--if the flag is true, current reference is lost-->
        <xsl:if test="$flag != 1">
            <xsl:copy-of select="."/>
        </xsl:if>
    </xsl:for-each>
    <!--copy current element structure-->
    <xsl:copy-of select="@*"/>
    <xsl:copy-of select="text()"/>
    <xsl:apply-templates select="*" />
</xsl:element>
</xsl:template>
<xsl:template name="compare-namespaces">
    <xsl:param name="list_to-compare"/>
    <xsl:param name="comparison-object"/>
    <!--define first token of current list of namespaces as current comparison token-->
    <xsl:variable name="token" select="substring-before($list_to-compare, '|')"/>
    <xsl:variable name="comparison-token" select="document($file_source-namespaces)//namespace[prefix = $token]/uri"/>
    <!--if current namespace (see calling template) is included in list of namespaces (see global variable), it must be removed from current element-->
    <xsl:if test="$comparison-object = $comparison-token">
        <xsl:text>1</xsl:text>
    </xsl:if>
    <!--if there are more namespaces to process in current list of namespaces, go on with recursive action-->
    <xsl:if test="string-length(substring-after($list_to-compare, '|')) != 0">
        <xsl:call-template name="compare-namespaces">
            <xsl:with-param name="list_to-compare" select="substring-after($list_to-compare, '|')"/>
            <xsl:with-param name="comparison-object" select="$comparison-object"/>
        </xsl:call-template>
    </xsl:if>
</xsl:template>
</xsl:stylesheet>

```

**/\* 27\_remove-separator.xsl \*/**

```

<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <!--insert filename of namespace file-->
    <xsl:param name="file_namespaces"/>
    <!--insert value of string of characters separating different terms inside attribute or text node-->
    <xsl:param name="separator"/>
    <!--<xsl:strip-space elements="*" />-->
    <xsl:template match="*">
        <!--process each element of current document-->
        <xsl:copy>
            <!--process each attribute of current element-->
            <xsl:for-each select="@*">
                <xsl:variable name="attribute_current-value" select="."/>
                <xsl:choose>

```

```

<!--in case of attribute value containing occurrences of separator-->
<xsl:when test="contains($attribute_current-value, $separator)">
  <xsl:variable name="attribute_current-name" select="name()"/>
  <xsl:variable name="pre" select="substring-before($attribute_current-name, ':')"/>
  <xsl:choose>
    <!--in case of attribute name associated with namespace prefix-->
    <xsl:when test="string-length($pre) != 0">
      <xsl:variable name="ns" select="document($file_namespaces)//namespace[prefix = $pre]/uri"/>
      <xsl:attribute name="{ $attribute_current-name}" namespace="{ $ns}">
        <xsl:variable name="new-content">
          <xsl:call-template name="process-content">
            <xsl:with-param name="content" select="$attribute_current-value"/>
            <xsl:with-param name="separator" select="$separator"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:value-of select="$new-content"/>
      </xsl:attribute>
    </xsl:when>
    <!--in case of attribute name not associated with namespace prefix-->
    <xsl:otherwise>
      <xsl:attribute name="{ $attribute_current-name}">
        <xsl:variable name="new-content">
          <xsl:call-template name="process-content">
            <xsl:with-param name="content" select="$attribute_current-value"/>
            <xsl:with-param name="separator" select="$separator"/>
          </xsl:call-template>
        </xsl:variable>
        <xsl:value-of select="$new-content"/>
      </xsl:attribute>
    </xsl:otherwise>
  </xsl:choose>
</xsl:when>
<!--in case of attribute value not containing occurrences of separator-->
<xsl:otherwise>
  <xsl:copy-of select="."/>
</xsl:otherwise>
</xsl:choose>
</xsl:for-each>
<!--process value of text node of current element-->
<xsl:choose>
  <!--in case of text node containing occurrences of separator-->
  <xsl:when test="contains(text(), $separator)">
    <xsl:variable name="new-content">
      <xsl:call-template name="process-content">
        <xsl:with-param name="content" select="text()"/>
        <xsl:with-param name="separator" select="$separator"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:value-of select="$new-content"/>
  </xsl:when>
  <!--in case of text node not containing occurrences of separator-->
  <xsl:otherwise>
    <xsl:copy-of select="text()"/>
  </xsl:otherwise>
</xsl:choose>

```

```
                </xsl:otherwise>
            </xsl:choose>
            <xsl:apply-templates select="*" />
        </xsl:copy>
    </xsl:template>
    <xsl:template name="process-content">
        <xsl:param name="content" />
        <xsl:param name="separator" />
        <xsl:choose>
            <!--in case of list of terms-->
            <xsl:when test="string-length(substring-after($content, $separator)) != 0">
                <xsl:value-of select="$content" />
            </xsl:when>
            <!--in case of single term-->
            <xsl:otherwise>
                <xsl:value-of select="substring-before($content, $separator)" />
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
</xsl:stylesheet>
```