

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 D2.1.2

Ontology and Multilingual Support Technical Specification

Workpackage WP2 – Knowledge and Languages
Task T2.1 – Ontology and multilingual support

Abstract

This document presents and describes the two software modules, contemporarily released as deliverable D.2.1.1, resulting from the design and development work carried out in task T2.1. These software modules provide the needed support (Editor) to the definition and management of ontologies and (Mapper) to the generation of transformation files assuring the translation of concepts between pairs of ontologies.

The Editor is conceived to define and manage ontologies made of three basic components, namely a data model, a taxonomy and a controlled vocabulary. These components are considered all necessary to realise the minimum set of easy, entry level services the SEAMLESS mediators will make available to their associated user companies.

The Mapper is conceived to related two ontologies by establishing correspondences between the respective concepts and terms. It is used both to annotate the local ontology of the single user company with the concepts of the mediator common ontology, and to map on each other the (few) global reference ontologies from which mediators will derive their own ontologies.

Finally, the multilingual support consists of the possibility to translate the terms of the local ontologies, normally expressed in the home language of the specific region, into/from the terms of the global ontologies, expressed in English as lingua franca.

Start date of project	Jan 1 st , 2006	Duration of project	30 months
Deliverable due date	Jan 21 st , 2007	Actual submission date	Jan 29 th , 2007
Dissemination level	PU	Revision status	Final
Responsible partner	U MODENA	Authors	T2.1 participants

Change record

Rev. N.	Description	Author	Date	Review
0	Early draft	S. Dondi (U MODENA) C. Reggiani (U MODENA) M. Barbi (KELYAN)	Dec 11, 2006	F. Bonfatti (U MODENA) S. Abels (TIE) S. Sancho (AITEX)
1	Full draft	S. Dondi (U MODENA) C. Reggiani (U MODENA) M. Barbi (KELYAN)	Jan 10, 2007	C. Lima (CSTB) S. Campbell (TIE) J.V. Vidagany (ANTARA)
2	Final version	S. Dondi (U MODENA) C. Reggiani (U MODENA) M. Barbi (KELYAN)	Jan 26, 2007	F. Bonfatti (U MODENA)



Table of contents

1	EXECUTIVE SUMMARY	5
2	ONTOLOGY MANAGEMENT AND USE	7
2.1	SEAMLESS ontologies.....	7
2.1.1	<i>The LOCL ontology</i>	9
2.1.2	<i>The COMM ontology</i>	10
2.1.3	<i>The GLOB ontology</i>	11
2.2	Reference ontology data model.....	11
2.2.1	<i>Company model</i>	12
2.2.2	<i>Product/service model</i>	13
2.2.3	<i>Negotiation model</i>	15
2.2.4	<i>Collaboration model</i>	16
2.2.5	<i>Vocabulary and taxonomy</i>	18
2.3	Ontology Management applications.....	20
2.4	LOCL Management application.....	20
2.4.1	<i>Use scenario</i>	21
2.4.2	<i>Application functionality</i>	21
2.4.3	<i>Application interfaces</i>	22
2.5	COMM Management application	23
2.5.1	<i>Use scenario</i>	23
2.5.2	<i>Application functionality</i>	24
2.5.3	<i>Application interfaces</i>	24
2.6	GLOB Management application	26
2.6.1	<i>Use scenario</i>	26
2.6.2	<i>Application functionality</i>	27
2.6.3	<i>Application interfaces</i>	27
3	ONTOLOGY EDITOR MODULE.....	29
3.1	State-of-the-art	29
3.1.1	<i>Apollo CH</i> (http://apollo.open.ac.uk/index.html)	30
3.1.2	<i>Protégé 2000</i> (http://protege.stanford.edu/index.html)	31
3.1.3	<i>Differential Ontology Editor DOE</i> (http://opales.ina.fr/public).....	32
3.1.4	<i>SWOOP</i> (http://www.mindswap.org/2004/SWOOP).....	34
3.1.5	<i>OntoTerm</i> (http://www.ontoterm.com)	35
3.1.6	<i>TopBraid Composer</i>	36
3.1.7	<i>Ontology editor comparison</i>	37
3.2	Ontology data structure.....	38
3.2.1	<i>Data model structure</i>	39
3.2.2	<i>Vocabulary structure</i>	45
3.2.3	<i>Taxonomy structure</i>	48
3.2.4	<i>SEAMLESS ontology and OWL</i>	50
3.3	Editor functionality	51
3.3.1	<i>Major design decisions</i>	51
3.3.2	<i>High level architecture</i>	52
3.3.3	<i>Editor core - action component</i>	54
3.3.4	<i>Editor core – form component</i>	54
3.3.5	<i>Editor core – model component</i>	55
3.3.6	<i>Editor core – wizard component</i>	57
3.3.7	<i>Editor core – view component</i>	59
3.3.8	<i>Editor core – util component</i>	59



3.3.9	<i>Editor core – guieditor component</i>	60
3.3.10	<i>Editor core – command component</i>	60
4	ONTOLOGY MAPPER MODULE	62
4.1	State-of-the-art	62
4.1.1	<i>MapForce 2007 (http://www.altova.com/products/mapforce)</i>	63
4.1.2	<i>Delta 4.0 (www.softshare.com)</i>	64
4.1.3	<i>Stylus Studio (http://www.stylusstudio.com)</i>	65
4.1.4	<i>Mapper comparison</i>	66
4.2	Transformation file data structure	67
4.2.1	<i>Structural file</i>	68
4.2.2	<i>Dictionary file</i>	70
4.3	Mapper functionality.....	71
4.3.1	<i>Major design decisions</i>	71
4.3.2	<i>High level architecture</i>	72
4.3.3	<i>Mapper core - action component</i>	73
4.3.4	<i>Mapper core – form component</i>	74
4.3.5	<i>Mapper core – model component</i>	75
4.3.6	<i>Mapper core – wizard component</i>	76
4.3.7	<i>Mapper core – view component</i>	77
4.3.8	<i>Mapper core – util component</i>	77
4.3.9	<i>Mapper core – mapper component</i>	78
4.3.10	<i>Mapper core – service component</i>	78
4.3.11	<i>Mapper core – operations component</i>	78
4.3.12	<i>Mapper core – transformations component</i>	79
5	FINAL REMARKS	81
	APPENDIX A: ONTOLOGY MGT USER INTERFACE	82
	APPENDIX B: ONTOLOGY EDITOR USER INTERFACE.....	84
	Ontology view	84
	Data model.....	84
	<i>Data model type</i>	85
	<i>Data model type attribute</i>	86
	Taxonomy	88
	<i>Taxonomy Sector → Viewpoint</i>	89
	Vocabulary.....	90
	<i>Vocabulary (local language)</i>	90
	<i>Vocabulary (local and lingua franca)</i>	93
	APPENDIX C: ONTOLOGY MAPPER USER INTERFACE.....	94
	Mapper form editor.....	94
	Structural mapping.....	95
	<i>Structural mapping operation</i>	95
	Taxonomy mapping	97
	Vocabulary mapping.....	98
	Mapper history view	99
	<i>Structural mapping history view</i>	99
	<i>Taxonomy mapping history view</i>	100

1 Executive Summary

This document presents the results of the work carried out in workpackage WP2 “Knowledge and Languages” and, specifically, in task T2.1 “Ontology and multilingual support”. In practice, it describes the functionality and provides the technical specifications of the two software modules that are delivered simultaneously as deliverable D2.1.1 “Ontology and multilingual support tool”, namely the ontology Editor and the ontology Mapper.

These software modules are developed starting from the analysis carried out in task T3.1 and documented in deliverable D3.1 “Overall Architecture Design”. They are intended to create and map onto each other the SEAMLESS global, common and local ontologies according to a “Local-As-View” (LAV) approach. The Editor provides the needed support to the definition and management of ontologies at the different levels of their hierarchy, while the Mapper generates the transformation files assuring the cross reference of concepts between pairs of ontologies.

The Editor is conceived to define and manage ontologies made of three basic components, namely a data model, a taxonomy and a controlled vocabulary. These components are considered all necessary to realise the minimum set of easy, entry level services the SEAMLESS mediators will make available to their associated user companies.

The Mapper is conceived to related two ontologies by establishing correspondences between the respective concepts and terms. It is used both to annotate the local ontology of the single user company with the concepts of the mediator common ontology, and to map on each other the (few) global reference ontologies from which mediators will derive their own ontologies.

Finally, the multilingual support consists of the possibility to translate the terms of the local and common ontologies, normally expressed in the home language of the specific region, into/from the terms of the global ontologies, expressed in English as *lingua franca*.

This document is subdivided into three main chapters and three appendices:

- SEAMLESS ontologies. The chapter recalls the reasons for using a three-levels hierarchy of ontologies and examines in detail the role of the ontologies in each level and the relations between ontology components at different levels. Then, the ontology management applications needed to handle the three hierarchy levels are described and the needs and requirements for an ontology Editor and Mapper are derived.
- Ontology Editor module. The chapter describes the Ontology Editor, the software module in charge of supporting the creation of an ontology composed by a data model, a taxonomy and a vocabulary. The Editor is general enough to be used in all the above-mentioned ontology management applications by adapting to the specific requirements of each of them. The chapter provides a state-of-the-art analysis and a detailed technical specification of the ontology data structure and the Editor software functionality.
- Ontology Mapper module. The chapter describes the Ontology Mapper, the software module in charge of supporting the mapping between concepts and terms of two ontologies. The Mapper is general enough to be used in the above-mentioned ontology management applications by adapting to the specific requirements of each of them. The chapter provides a state-of-the-art analysis and a detailed technical specification of the transformation file data structure and the Mapper software functionality.
- Appendices: user interfaces. The appendices collect the description of the user interfaces of the ontology management functions. In particular, it describes the user interfaces foreseen for the LOCL, COMM and GLOB management applications (to be developed in the next period) and those of the ontology Editor and Mapper modules (already realised).

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

- Technical partners in the SEAMLESS consortium. Deliverable D2.1.1 provides two software modules 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 modules and the data structures they generate.

- Mediator partners in the SEAMLESS consortium. The mediator partners will play a paramount role in validating the SEAMLESS technological infrastructure and deploying the developed applications and services. In order to do this they must understand as much as possible the role and potential of ontologies and use them in the most effective way. This deliverable can provide them with the needed knowledge and then constitute the basis for their work.
- Ontology experts in the SEAMLESS consortium. Following the indications contained in this deliverable the ontology experts can start developing one or more reference global ontologies (likely, one generic GLOB and two sector-specific GLOBs) to be used as reference during the rest of the project.
- The other projects of the DE cluster. The ontology management functions realised by the SEAMLESS project 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, this deliverable is a way to publish the description of two important and critical components of the SEAMLESS platform that can enable third parties (e.g. software houses, service providers) to undertake parallel development initiatives.



2 Ontology management and use

This chapter recalls first the reasons why SEAMLESS is using a three-levels hierarchy of ontologies and examines in detail the role of the ontologies in each level and the relations between ontology components at different levels. Then it analyses the functionality of the three management applications that are developed to deal with each of the ontology hierarchy levels, and derives from this the requirements of the ontology Editor and Mapper.

2.1 SEAMLESS ontologies

There are many difficulties to move a small or micro company towards a full participation in the electronic market, even though it is already used to collaborate with other companies and act in other countries. They include problems to divert technical personnel from core activities, lack of skills for process revision, awkwardness with different languages and regulations, and also limited money to invest in ICT tools and complexity of most ICT solutions.

The SEAMLESS project intends to contribute solving these problems by providing the target company with easy and cheap services made available through the mediators it is normally associated with. 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 platform¹. 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 company². 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 (we call GLOB this global ontology), based on English as *lingua franca*, is required to provide the bridge³ between different COMMs. 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 GLOBs⁴ (e.g. generic ontologies, sector-specific ontologies, etc.) that qualified organisations 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

¹ See requirements UC17, UC18, UC43, UC44 of deliverable D3.1.

² See requirements UC13, UC14 of deliverable D3.1.

³ See requirement UC16 of deliverable D3.1.

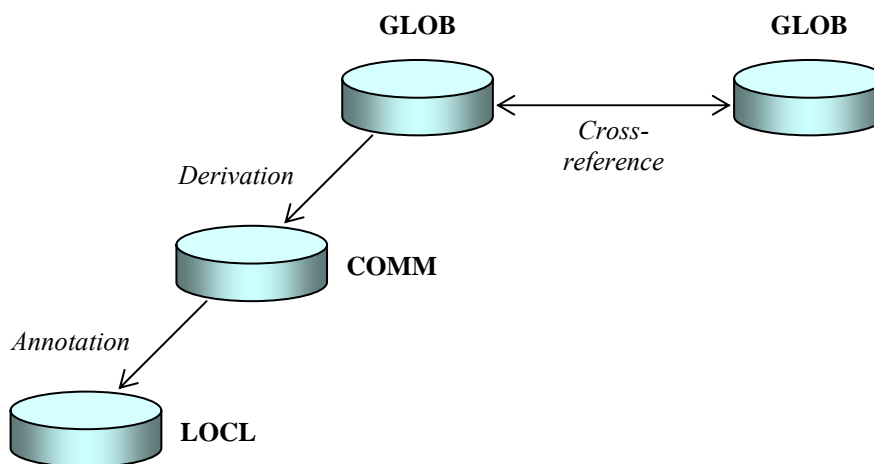
⁴ See requirements UC08, UC09 of deliverable D3.1.



relate homologous concepts and terms in the different GLOBs⁵. 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.

The consequence of these considerations is the ontology hierarchy depicted in the figure below. Note that the relations between ontologies have meanings that differ according to the hierarchy levels:

- **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 COMM⁶.
- **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 language⁷.
- **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 them⁸.



In the following we assume that a company is associated to only one mediator, and a mediator derives its COMM from just one GLOB. However, from the technical viewpoint more general specifications might be considered, for instance:

- A company can be associated to two or more mediators, meaning that it can annotate the data of its LOCL ontology with the concepts of different COMMs. This will enable that company to interface its information system with the applications and services made available by two or more platforms.
- A COMM can be derived from two or more GLOBs. Although not impossible, this is anyway quite complicated since it should likely call for generating the COMM by derivation from the

⁵ See requirements UC11, UC12 of deliverable D3.1.

⁶ See requirements UC17, UC18 of deliverable D3.1.

⁷ See requirement UC16 of deliverable D3.1.

⁸ See requirements UC11, UC11 of deliverable D3.1.

union of the reference GLOBs. It is not easy to find out practical cases where this solution might be useful, then its interest is actually negligible.

- A mediator can propose to its associated companies two or more COMMs derived from the same GLOB but in different languages (e.g. in Brussels there are French and Flemish speaking companies). The two versions might be conveniently maintained aligned with each other by a function to be studied for the purpose.
- A mediator can propose to its associated companies two or more COMMs each derived from a different GLOB. In this case it could be useful to consider the introduction of COMM-COMM relations as a shortcut to assure more efficiency.

These possibilities are however out of the SEAMLESS project scope, and as such they will not be considered in the development of this workpackage. They will only be taken into account for possible future developments after project completion.

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 defined as follows:

- **Data model.** A relevant part of the shared and exchanged knowledge consists of 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, in practice it is expected 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 means of payment. Although significantly depending on the single context, it is worth including them into the ontology as part of the 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.

In the next sections of this chapter the LOCL, COMM and GLB ontologies are examined in more detail so as to represent their roles, relations and differences.

2.1.1 The LOCL ontology

The LOCL ontology represents the concepts used by the single company when sharing or exchanging information with other companies. If the company is provided with an enterprise system some (not all) of the concepts of its data model are communicated to partners during the negotiation and collaboration phases: these are in fact the only concepts to be considered when building the company LOCL for interfacing its enterprise system with the mediator platform.

With respect to the above recalled structure of the SEAMLESS ontologies, the LOCL is characterised by the following simpler structure:



- Data model. This is the main LOCL component, reporting the concepts (tables, attributes, fields) representing the company and its demand/offer of products and services and those to be exchanged with partners in the sent/received business documents.
- Taxonomy. It is assumed that the single user company is not interested to propose its own general taxonomy of products and services, rather it will adopt in all its operations the taxonomy proposed by the mediator COMM.
- Vocabulary. The classifications used in the local data model (e.g. to code the features of its products and services or the payment means) are maintained by including them, and their respective options, into the vocabulary.

In order to establish a relation with the mediator platform, the LOCL concepts of the company must be mapped onto (annotated by) the COMM of its mediator. It is expected that the concepts of a LOCL are fully included into those of the reference COMM: if this does not occur, the LOCL can ask the COMM to extend its contents so as to find all the needed concepts. Experiences made by U MODENA in the supply chain and logistic fields show that this extension process is rapidly converging and very seldom the association to the mediator of a new company, with its own LOCL, requires a further addition in the COMM⁹.

2.1.2 The COMM ontology

The COMM ontology includes the concepts and terms the single SEAMLESS node considers necessary for its associated companies to collaborate and exchange data with their partners. Therefore it is defined by the mediator as a semantic (common) basis for its associated companies. It is expected that the COMM includes the concepts of all the LOCLs of the associated companies, which means it must be equal to or larger than their union.

The COMM is obtained by derivation from the reference GLOB. In principle, it is derived by selection (only useful concepts are taken) and language translation (from English to the mediator home language). However it is allowed to add concepts (not present in the GLOB) if they are useful to facilitate the collaboration among the companies associated to the single mediator.

With respect to the above recalled structure of the SEAMLESS ontologies, the COMM is characterised by the following features:

- Data model. This is a full data structure including and translating, at least, the mandatory concepts for company description, partner search, negotiation, and collaboration with partners. Generally speaking, the data model mirrors that of the reference GLOB meaning that it can be generic or sector-specific depending on the nature of the chosen GLOB.
- Taxonomy. The reference GLOB provides the COMM with a standard taxonomy whose terms (and relative descriptions) are simply translated from English into the local language. More precisely, the mediator selects the taxonomy terms to translate then the COMM includes both translated terms and terms with the only English description.
- Vocabulary. The COMM vocabulary is first derived and translated from that of the reference GLOB. However, contributions to its extension come from the LOCL vocabularies that are progressively annotated by this COMM. Every new term is introduced in both the local language and in English so as to be automatically forwarded to the GLOB.

The COMM concepts are reported and described in the language of the specific region. Every COMM is built up by deriving its concepts from one GLOB in order to establish a relation between the regional terms and those in the global ontology. It is expected that the general concepts of a COMM are fully included into those of the reference GLOB. When this does not occur, the COMM manager can ask the GLOB manager to extend its contents in order to find all the needed concepts¹⁰. Vice versa, whenever an addition is introduced into the GLOB, it is automatically communicated to all its derived COMMs.

⁹ See requirements UC05, UC07 of deliverable D3.1.

¹⁰ See requirements UC05, UC06, UC07 of deliverable D3.1.



2.1.3 The GLOB ontology

The GLOB ontology includes all the concepts that are considered necessary to collaborate and exchange information within the network, as such it can be seen as the reference knowledge of the system. Its concepts are reported and described in English intended as *lingua franca*. It is set up and managed by a qualified organisation assuring its maintenance and the access to its contents through proper web services.

With respect to the above recalled structure of the SEAMLESS ontologies, the GLOB is characterised by the following features:

- Data model. It includes all the concepts the SEAMLESS model considers mandatory for company description, partner search, negotiation and collaboration with partners, plus additional concepts that are introduced to fulfil the user requirements. Depending on whether it is a generic or sector-specific ontology, the data model can be taken, e.g., from the UBL standard or, say, from the TEXWEAVE standard for the textile sector.
- Taxonomy. Similarly, the GLOB adopts a standard taxonomy for products and services that could be generic or sector-specific. In order to simplify the task of mediators in deriving the respective COMMs, the standard taxonomy can be proposed as it is or already reduced to a minimum set of useful terms.
- Vocabulary. In principle, the GLOB could be generated with only the terms coming from the taxonomy. In fact its vocabulary is mainly developed thanks to the contributions coming from the subscribing mediators which, in turn, receive most of them from the annotated LOCLs of the associated companies.

The SEAMLESS environment must include at least one GLOB, but different organisations can propose GLOBs in competitions with each other. In order to realise a semantic roaming condition, the GLOB manager should map the elements of its GLOB, namely data model, taxonomy and vocabulary against the homologous ones in the other GLOBs¹¹.

2.2 Reference ontology data model

The SEAMLESS project will provide a mechanism by which companies can actively collaborate even if the managed information is defined in different formats and languages. This means that, in line of principle, SEAMLESS ensures the communication among companies which manage information contents according to different ontologies.

However, the effectiveness of the SEAMLESS translation facilities considerably depends on the level of overlapping of the involved ontology contents. Therefore, this section introduces those concepts that all the SEAMLESS Global ontologies should, at least, implement in order to enable the basic SEAMLESS communication processes. They represent just **the skeleton** of every SEAMLESS-compliant data model.

In other words, all the SEAMLESS global ontologies are asked to define their specific data models by including and expanding this reference data structure. The assumptions and guidelines outlined below have been followed throughout the design process:

- The data model concepts are arranged in four separate sub-models, namely company, product/service, negotiation and collaboration, and an additional sub-model to represent vocabulary and taxonomy.
- Many ideas have been inspired by the Unified Business Language (UBL) specifications, thus providing a reference ontology compliant with the nowadays standards in terms of business communications.
- Since this is a reference ontology, its attributes should be considered mandatory for all the SEAMLESS ontologies. Of course, any single ontology will put “meat” around this skeleton by introducing additional concepts, which can be mandatory or optional in turn for the derived ontologies (if any).

¹¹ See requirements UC11, UC12 of deliverable D3.1.

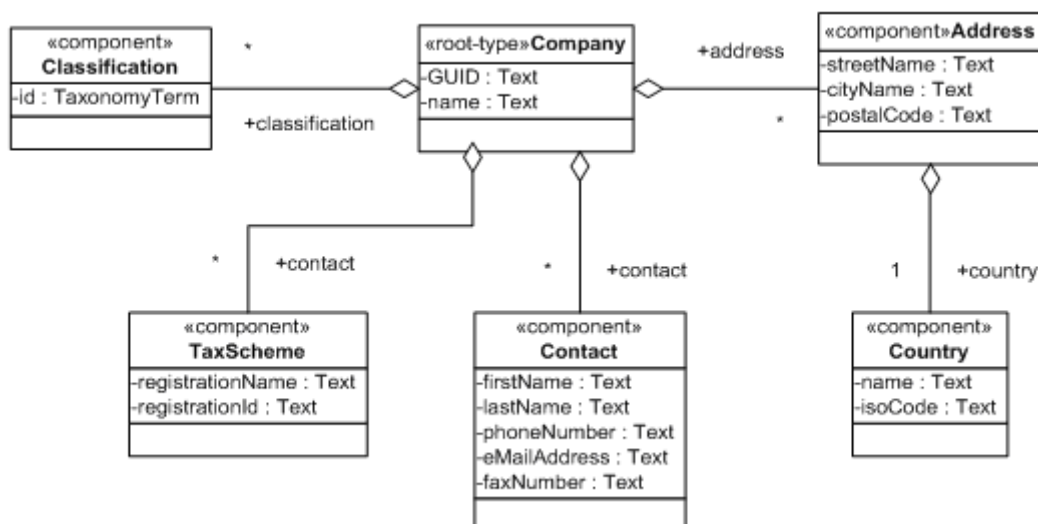


Additionally, the diagrams defined in the sections above follow the design practices supplied by the SEAMLESS Editor Module application. In particular:

- Classes can be defined as root-type or component. A root-type is a container for other root-type classes or component while a component is a reusable class that can only exist within another component or root-type.
- The composition among components and root-types is the only allowed form of association among classes.
- The adopted data types (a subset of the Editor predefined set) are:
 - **Text** (a free text not subjected to translation);
 - **DateTime** (information on date and time);
 - **Date** (information on date);
 - **Term** (a text representing a term defined in the ontology vocabulary and, then, subjected to translation);
 - **Decimal** (a decimal number);
 - **TaxonomyTerm** (a text representing a term defined in the ontology taxonomy and, then, subjected to translation);
 - **Binary** (a data structure represented by a link to an Internet resource along with its relevant MIME type);
 - **NumParam** (a data structure represented by a NumTerm, i.e. a vocabulary term expressing a numeric measure along with its measure of unit, and a text representing the users choice as numeric value);
 - **EnumParam** (a data structure represented by a EnumTerm, i.e. a vocabulary term expressing a discrete physic feature along with the set of possible values, and a subset of these values representing the users choice).

2.2.1 Company model

This schema defines the basic structure of a company.



Company root-type:

- *GUID*: Global unique identifier of the company.
- *name*: The name of the company.
- *classification*: The set of taxonomy terms identifiers described in the Classification component (see below).
- *taxScheme*: It refers to the TaxScheme component information.
- *contact*: The personal data, first name and last name, described in the Contact component.
- *address*: The address information provided by the Address component.

Classification component:

- *id*: The unique identifier for a specific taxonomy term¹².

TaxScheme component:

- *registrationName*: The registration information necessary to identify a company with the tax commercial code (i.e. vat number).
- *registrationId*: The identifier registration value of the *company*.

Contact component¹³:

- *firstName*: The first name of the contact person in the company.
- *lastName*: The last name of the contact person in the company.
- *phoneNumber*: Phone number of the contact person in the company.
- *eMailAddress*: The e-mail address of the contact person in the company.
- *faxNumber*: The fax number of the contact person in the company.

Address component:

- *streetName*: the name of the street as part of the address
- *cityName*: the name of the city, town, village, or a built up area and used as part of an address.
- *postalCode*: a string of characters for one or more properties according to the postal service of that country; a group of letters and/or numbers added to the postal address to assist in the sorting of mail.

Country component:

- *name*: The ISO-compliant country name (related to an existing isoCode).
- *isoCode*: A code defined by the standard ISO identifying a country.

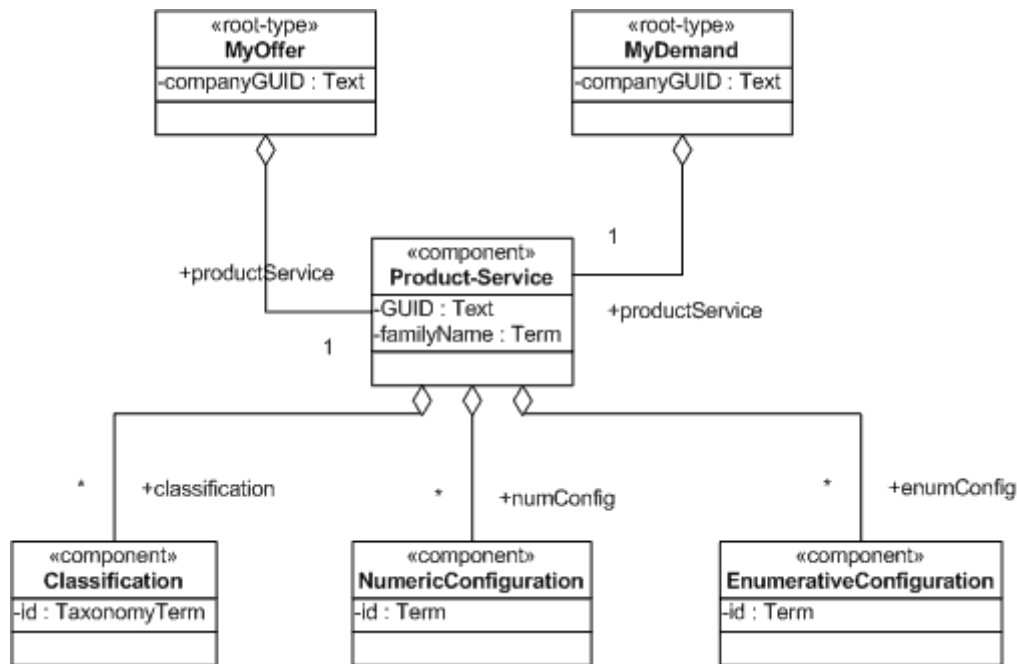
2.2.2 Product/service model

This is the data structure to support the definition of products and services offered (MyOffer) and demanded (MyDemand) by a company.

¹² A Company should be described by means of at least one taxonomy term (e.g. 'Roof covering', 'Wood window', etc...). These taxonomy terms are used in order to classify a company in terms of products/services and to permit the same company to introduce itself. The 'id' attribute is the identifier associated to a taxonomy term.

¹³ The attributes *phoneNumber*, *eMailAddress* and *faxNumber* are not all strictly mandatory. However, it is recommended that at least one of these attributes is defined.





MyOffer root-type:

- *companyGUID*: The global unique identifier of the company offering a product-service.
- *productService*: The information about a given product/service provided by the Product-Service component.

MyDemand root-type:

- *companyGUID*: The global unique identifier of the company offering a product-service.
- *productService*: The product/service information provided by the Product-Service component.

Product-Service component:

- *GUID*: The global unique identifier of the product-service.
- *familyName*: The descriptive product family name.
- *classification*: The set of taxonomy terms identifiers described in the Classification component (see below).
- *numConfig*: The identifier of a numeric information used for the product/service configuration provided by the NumericConfiguration component (see below).
- *enumConfig*: The identifier of an enumerative information used for the product/service configuration provided by the EnumerativeConfiguration component.

Classification component:

- *id*: The unique identifier of a specific taxonomy term¹⁴.

NumericConfiguration component:

- *id*: The numeric identifier¹⁵.

¹⁴ These classification attributes are in charge of describing products/services offered and demanded by every company. The 'id' attribute is the identifier associated to each taxonomy term.

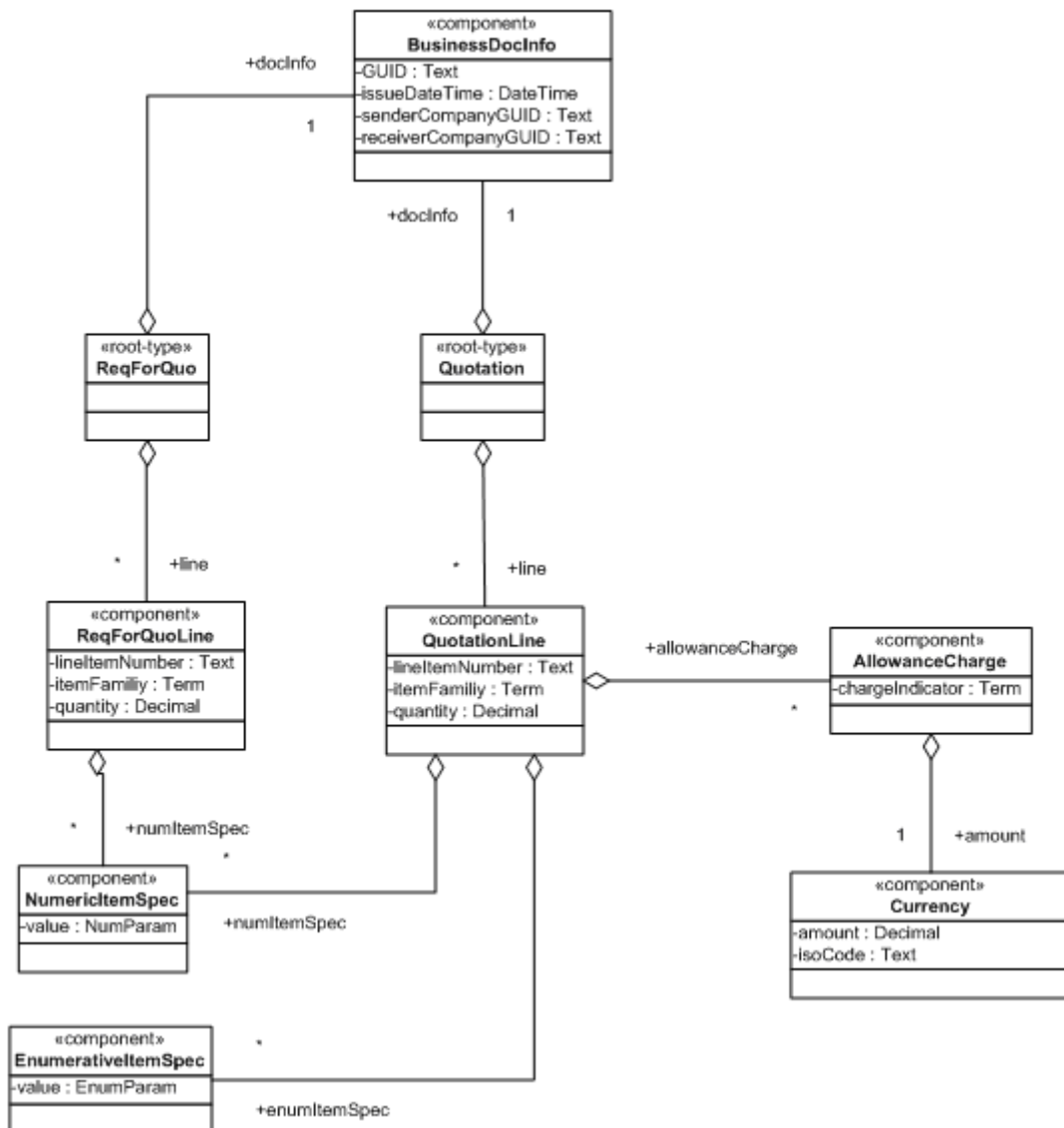
¹⁵ 'NumericConfiguration' components are terms defined in the SEAMLESS Vocabulary used to better define each product/service. E.g. a product/service 'House external door' could be described by the

EnumerativeConfiguration component:

- *id*: The enumerative value identifier¹⁶.

2.2.3 Negotiation model

This schema defines the structure of those documents on which the negotiation between companies will rely. The negotiation is considered the communication process by which companies define an agreement on the demanded and offered conditions. This stage anticipates the exchange of business documents.



numeric configuration term 'Height' that indicates the vertical height of the door. The 'id' attribute is the identifier associated to each NumericConfiguration term (e.g. 'Height').

¹⁶ 'EnumerativeConfiguration' components are terms defined in the SEAMLESS Vocabulary used to better define each product/service. E.g. a product 'House external door' could be described by the enumerative configuration term 'Colour'. Obviously, through the SEAMLESS Editor UI, user is allowed to hook (add) the list of SEAMLESS Vocabulary terms defining the domain of the 'Colour' term: e.g. 'Red', 'Yellow', 'Green', etc... The 'id' attribute is the identifier associated to each EnumerativeConfiguration term (e.g. 'Colour').

BusinessDocInfo component:

- *GUID*: The Global unique identifier of the request for quotation document.
- *issueDateTime*: The date time to issue the business document.
- *senderCompanyGUID*: The reference to the sender company described in the Company model.
- *receiverCompanyGUID*: Reference to the receiver company described in the Company model.

ReqForQuo root-type:

- *docInfo*: The business document basic information provided by the BusinessDocInfo component.
- *line*: The request for quotation line information provided by the ReqForQuoLine component.

Quotation root-type:

- *docInfo*: The business document basic information provided by the BusinessDocInfo component.
- *line*: The quotation line information provided by the QuotationLine component.

ReqForQuoLine component:

- *lineItemNumber*: The information necessary to identify a line of a request for quotation.
- *itemFamily*: The descriptive product family name.
- *quantity*: The number of the products.
- *enumItemSpec*: An enumerative value to better define the product of the line¹⁷.
- *numItemSpec*: A numeric value to better define the product of the line¹⁸.

QuotationLine component:

- *lineItemNumber*: The information necessary to identify a line of a quotation.
- *itemFamily*: The descriptive product family name.
- *quantity*: The number of the products.
- *enumItemSpec*: Enumerative value to describe the product in the line (see the footnote above).
- *numItemSpec*: A numeric value to better define the product in the line (see the footnote above).
- *allowanceCharge*: It reports details about a component of pricing, such as a service, promotion, allowance, or charge, applied to an associated quotation line or the whole transaction.

AllowanceCharge component:

- *chargeIndicator*: The allowance or charge information for a particular quotation line.
- *amount*: The price information provided by the Currency component.

Currency component:

- *amount*: The calculated value to represent the payment price.
- *isoCode*: A code defined by the standard ISO identifying a currency.

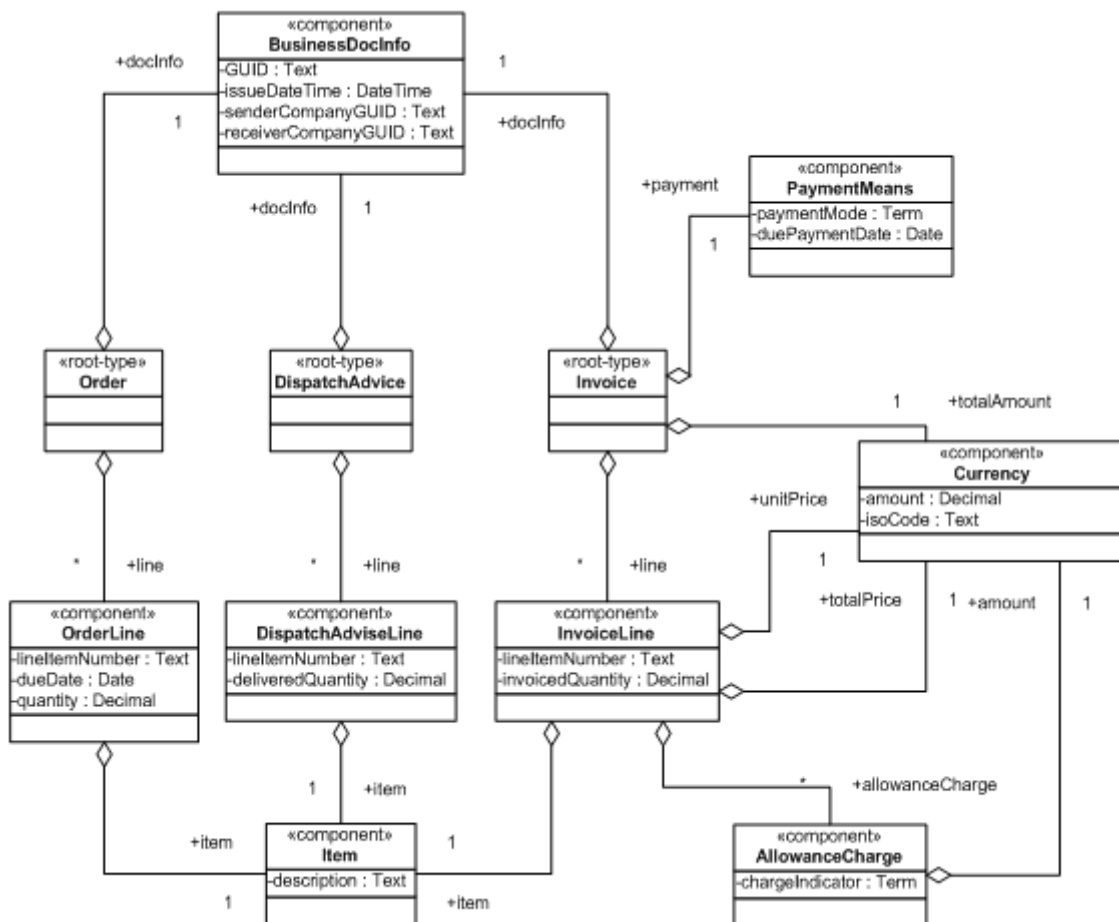
2.2.4 Collaboration model

This schema defines the basic structure of the business documents generated by the collaboration activity between companies, namely order, dispatch advice and invoice.

¹⁷ For example, if the document's line handles a 'Door' product, an 'enumItemSpec' could be the enumerative configuration term 'Colour'.

¹⁸ For example, if the document's line handles a 'Door' product, a 'numItemSpec' could be the numeric configuration term 'Height'.





Order root-type:

- *docInfo*: The business document basic information provided by the BusinessDocInfo component.
- *line*: The order lines information provided by the OrderLine component.

DispatchAdvice root-type:

- *docInfo*: The business document basic information provided by the BusinessDocInfo component.
- *line*: The advice lines information provided by the DispatchAdviceLine component.

Invoice root-type:

- *docInfo*: The business document basic information provided by the BusinessDocInfo component.
- *line*: The invoice lines information provided by the InvoiceLine component.
- *payment*: It refers to the information described in the PaymentMeans component. It depicts the information directly related to the means of payment.
- *totalAmount*: The total invoice price information provided by the Currency component.

BusinessDocInfo component:

- *GUID*: Global unique identifier of the business document.
- *issueDateTime*: The date time to issue the document.
- *senderCompanyGUID*: Global unique identifier of the sender company.
- *receiverCompanyGUID*: Global unique identifier of the receiver company.

OrderLine component:

- *lineItemNumber*: The information necessary to identify a line of an order.
- *dueDate*: The expiration date of the line .
- *quantity*: The amount of pieces of a particular requested good.
- *Item*: A description of the product contained in the Item component.

DispatchAdviceLine component:

- *lineItemNumber*: The information necessary to identify a line of an advice.
- *deliveredQuantity*: The delivered quantity of a given product.
- *Item*: A description of the product contained in the Item component.

InvoiceLine component:

- *lineItemNumber*: The information necessary to identify a line of an invoice.
- *invoiceQuantity*: The delivered quantity of a given product.
- *Item*: A description of the product contained in the Item component.
- *unitPrice*: The unit price provided by the Currency component. It is the cost of one good among a set of 'invoiceQuantity' goods.
- *totalPrice*: The total cost provided by the Currency component. It is the multiplication between the 'unitPrice' and the 'invoiceQuantity' of a given product.
- *allowanceCharge*: It reports details about a component of pricing, such as a service, promotion, allowance, or charge, applied to an associated invoice line or the whole transaction.

PaymentMeans component:

- *paymentMode*: It identifies the way through with the payment is processed. For example the SEAMLESS Vocabulary term 'Credit Card' could represent an example of payment mode.
- *duePaymentDate*: The point in time in which the payment is to be made.

Currency component:

- *amount*: The calculated value to represent the payment price.
- *isoCode*: The currency code of the payment (ISO-compliant).

AllowanceCharge component:

- *chargeIndicator*: The allowance or charge information for a particular invoice line.
- *amount*: The quantity information provided by the Currency component.

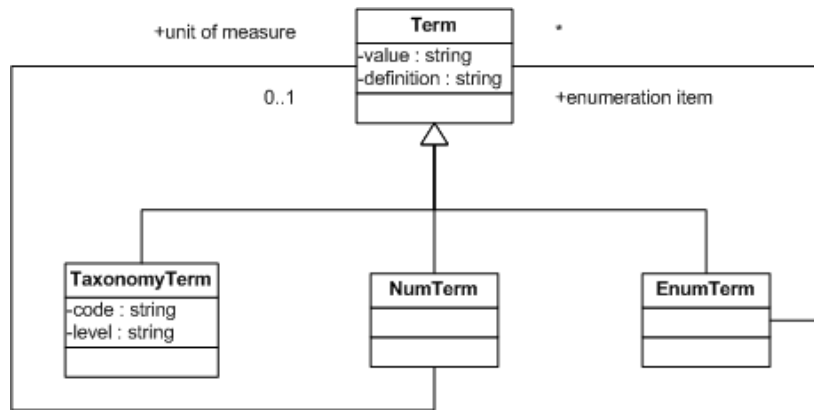
2.2.5 Vocabulary and taxonomy

The reference ontology, as well as any SEAMLESS ontology, relies on the concepts of vocabulary and taxonomy. The vocabulary contains all the terms available within an ontology while the taxonomy is a subset of vocabulary terms properly arranged as hierarchical tree structure.

The reference ontology defines the data structure of both the vocabulary and the taxonomy in order to provide a single architecture shared among all the SEAMLESS ontologies. It is important to point out that, differently from the data sub-models previously introduced, the vocabulary and taxonomy structure cannot be either altered or enriched by the single ontology. What ontologies are asked to do is to provide their relevant vocabularies and taxonomies by proper contents, respecting the designed architecture.

The figure below shows the schema of the vocabulary data structure:





Term class. A simple term:

- *value*: Term value (e.g. glass).
- *definition*: Term definition providing a way for recognising the term among other terms with the same value (e.g. tool for drinking)

TaxonomyTerm class. A term within a taxonomy structure (see below for taxonomy details):

- *code*: User provided code for the term in the taxonomy.
- *level*: Dotted notation (e.g. 1.2.4) representing the term level within the taxonomy.

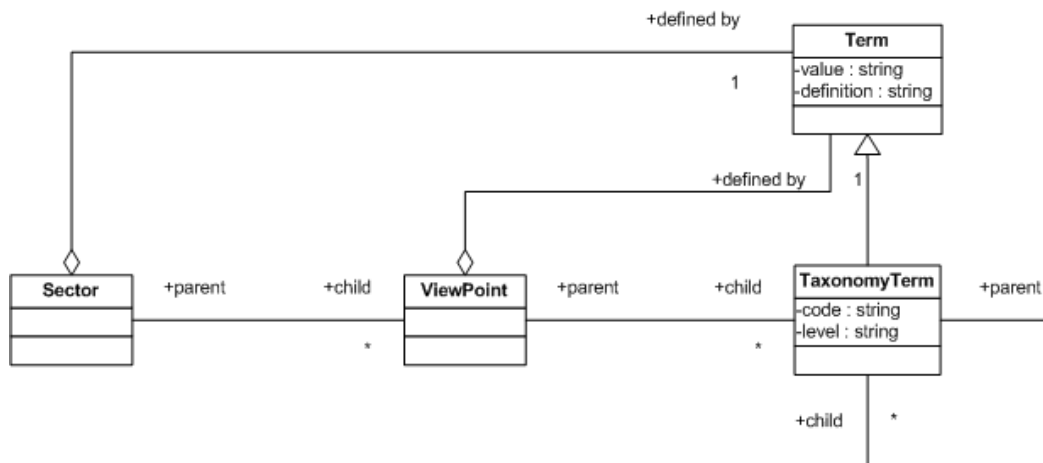
NumTerm class. A term representing a numeric measure (e.g. length, surface, height, etc.):

- *unit of measure*: Optional term defining the unit of measure.

EnumTerm class. A term representing a discrete set of options (e.g. colour associated to red, blue, etc):

- *enumeration item*: Term representing an enumeration item.

The schema representing the taxonomy structure follows:



Term class. A simple term:

- *value*: Term value (e.g. glass).
- *definition*: Definition of the term in natural language (e.g. tool for drinking), in order to uniquely identify a term among other terms with the same value.

TaxonomyTerm class. A term within a taxonomy structure:



- *code*: The code of the term in the taxonomy.
- *level*: Dotted notation (e.g. 1.2.4) representing the term level within the taxonomy.
- *parent*: The parent node of the taxonomy term. It could be represented by either another TaxonomyTerm or a ViewPoint.
- *child*: The list of TaxonomyTerm playing the role of children nodes.

ViewPoint class. A container for a TaxonomyTerm hierarchy (e.g. process, product, service, etc.):

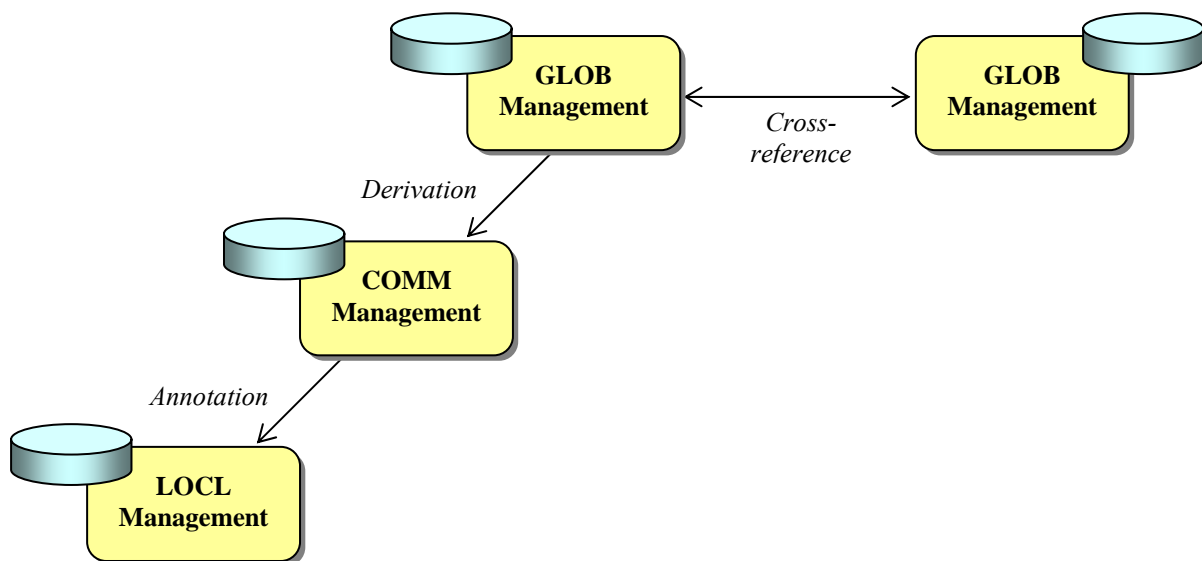
- *child*: The list of TaxonomyTerm playing the role of children nodes.
- *defined by*: The term representing the actual content of the ViewPoint.
- *parent*: The Sector to which the ViewPoint belongs.

Sector class. A container for taxonomy ViewPoints. It allows users to manage taxonomies related to different sectors in the same ontology:

- *child*: The list of ViewPoints playing the role of children nodes.
- *defined by*: The term representing the actual content of the Sector.

2.3 Ontology Management applications

Editing, annotating, deriving, cross-referencing ontologies calls for specific applications able to provide the needed support to the involved ontology experts. We can now enrich the figure of section 2.1 by showing the Ontology Management applications the SEAMLESS project is developing to this purpose.



In the following sections the three management applications for LOCL, COMM and GLOB ontologies are described in detail with the purpose to show their interactions and their main components and, specifically, to emphasise the critical role played by the Ontology Editor and Ontology Mapper modules that will be technically documented in the next chapter.

2.4 LOCL Management application

This section describes the functionality of the SEAMLESS application addressed to manage the data structure (LOCL ontology) of the information system at the single user company (if any) and create its mapping onto (annotation by) the COMM ontology of the supporting mediator.



2.4.1 Use scenario

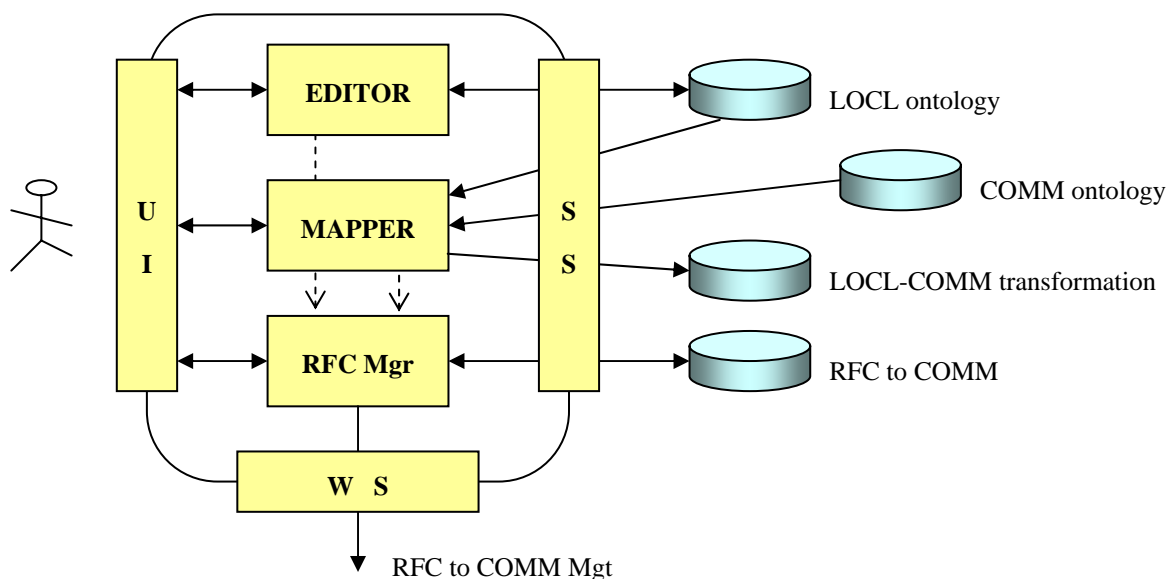
The LOCL Management application is intended to support the single company having its own enterprise information system (LOCL managing organisation) in performing the required activities about LOCL construction and mapping onto the COMM. The following (preliminary) list proposes the activities that will be, in most cases, executed with the support of the mediator experts:

- Select the concepts of interest. The user company selects the concepts (tables, attributes, fields) of its information system that are considered necessary, e.g., for defining the company profile and those typically contained in the business documents to exchange with the partners.
- Define the LOCL ontology. The user company uses the Editor of the LOCL Management application to define its LOCL, that is, the structure of the concepts that have been selected to exchange data and business documents with the mediator platform¹⁹.
- Generate the LOCL-COMM transformation files. The single company finally uses Mapper of the LOCL Management application to annotate the LOCL data with the COMM concepts. The result takes the form of cross-reference (mapping) files between this LOCL and the COMM²⁰.

If carried out with the help of mediator experts this task requires few hours, including the annotation phase, for every LOCL.

2.4.2 Application functionality

The LOCL Management application is expected to provide the single company with the functionality depicted in the following figure (UI means user interface, SS means storage system, and WS means web service interface):



- EDITOR. It is the LOCL ontology Editor, operated by the LOCL ontology responsible or a mediator expert working on behalf of it, to define and possibly update the LOCL ontology.
- MAPPER. It is the LOCL ontology Mapper that uses the COMM concepts to annotate the data of the LOCL ontology. The outcome are the LOCL-COMM transformation files to be processed by the translation service (see deliverable D2.2.2).

¹⁹ LOCL ontology editing has no explicit reference to D3.1 use cases; nevertheless it can be hooked to UC43 imagining that this import phase can refer to LOCL ontology editing too.

²⁰ See requirements UC17, UC18, UC43, UC44 of deliverable D3.1.

- RFC Mgr. It is the component to enter and manage the requests for change identified during the editing and mapping sessions. These requests include possible improvements of the COMM data model and new vocabulary terms to be added to the COMM vocabulary²¹.

2.4.3 Application interfaces

The storage system (SS) provides the needed storage and retrieval facilities for the identified data types, namely LOCL and COMM ontologies, LOCL-COMM transformation files, and RFC to COMM. Also, subscriptions of user companies to the COMM and the reverse notifications of changes to the COMM are not shown since it is assumed they are managed by the same storage system.

The SEAMLESS project intends to realise the storage system by adapting and integrating the widely mentioned SRRN platform developed in the frame of the SEEMseed project (see task T3.3). Looking forwards to detailing and using the interface with the SRRN the present version of Editor and Mapper stores the generated objects on the local file system.

The user interface (UI) is proposed in Appendix A.

The web service interface (WS) provides the following RPC invocations:

string GetReceiverList(string username, string password)	
Returns the list of possible receivers to whom the request can be sent according to the registered subscriptions.	
Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the LOCL manager. • <i>password</i>. Password of the LOCL manager.
Returns:	A XML file containing the list of potential receivers according to the provided parameters and available subscriptions. The receiver list only contains the receiver unique identifier.

int SendRequest(string username, string password, string receiver_id, string subject, string body, int priority, Object attachment)	
Sends a new request for change to selected receiver (COMM Manager in this specific case). The request is automatically associated to a unique identifier.	
Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the LOCL manager sending the request. • <i>password</i>. Password of the LOCL manager sending the request. • <i>receiver_id</i>. Identifies the request receiver by means of the value provided by the GetReceiverList method. • <i>subject</i>. Free text describing the subject of the request. • <i>body</i>. Free text explaining the details of the request. • <i>priority</i>. Priority level of the request (0 – low, 1 – medium, 2 – high) • <i>attachment</i>. (optional) A Raw file by which detailing the request (e.g. a data model schema, a vocabulary screenshot, etc.)
Returns:	0 if the operation succeeds, otherwise a negative integer value as an error code.

string GetSentRequestList(string username, string password)	
Returns the list of the requests sent by the identified SEAMLESS user.	
Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the LOCL manager.

²¹ See requirements UC05, UC07 of deliverable D3.1.

	<ul style="list-style-type: none"> • <i>password</i>. Password of the LOCL manager.
Returns:	A XML file containing the list of sent requests according to the provided user identification. The request list only contains the request unique identifier.

string GetRequestDetails(string username, string password, string request_id)	
Returns all the information related to the request for change identified by the provided identifier.	
Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the LOCL manager. • <i>password</i>. Password of the LOCL manager. • <i>request_id</i>. The request identifier obtained by the GetSentRequestList method.
Returns:	A XML file containing the information of the identified request. The system is in charge of validating the invocation according to the provided user identification parameters. The request for change information is actually yielded if the request_id is valid and the relevant request has been either sent or addressed to the identified user.

2.5 COMM Management application

This section describes the functionality of the SEAMLESS application addressed to manage the COMM ontology, which represents the knowledge shared by the all the user companies associated to a certain mediator and the bridge to collaborate with other companies in the network.

2.5.1 Use scenario

The COMM Management application is intended to support the ontology experts at the mediator (COMM managing organisation) in performing the required activities about COMM construction and update. A (preliminary) list of activities is here proposed:

- Choose the reference GLOB. The mediator willing to behave as SEAMLESS node with respect to its associated companies has first to choose the global ontology to take as reference for its COMM. The selection criteria can be numerous, such as the cultural sectoral or regional proximity as well as the qualification of the GLOB holder and the subscription cost.
- Define the COMM ontology. Once chosen the reference GLOB and subscribed to it, the mediator downloads the ontology into its COMM Management application²². After that it uses the COMM ontology Editor to select the GLOB concepts and terms of interest, and to translate them into the local language. Moreover, the mediator can add local concepts and terms in order to facilitate the communication among the companies associated to it.
- Generate the COMM-GLOB transformation files. The Editor of the COMM Management application produces the COMM ontology²³ and generates, at the same time, the cross reference²⁴ (mapping) files between this COMM and the reference GLOB, and possible requests for change to the GLOB itself²⁵.
- Make the COMM available to interested companies. Once defined the COMM, the mediator proposes it to the associated companies and discusses with conditions (and cost) of its use. The interested companies that decide to adopt it have simply to subscribe to that COMM and use it to annotate the data of the respective LOCLs.
- Propose changes to the GLOB. While creating or updating the COMM, or supporting the associated companies in mapping their LOCLs, the mediator can find limits in the reference

²² See requirements UC01, UC02, UC04 of deliverable D3.1.

²³ See requirements UC13, UC14 of deliverable D3.1.

²⁴ See requirement UC16 of deliverable D3.1.

²⁵ See requirements UC05, UC06, UC07 of deliverable D3.1.

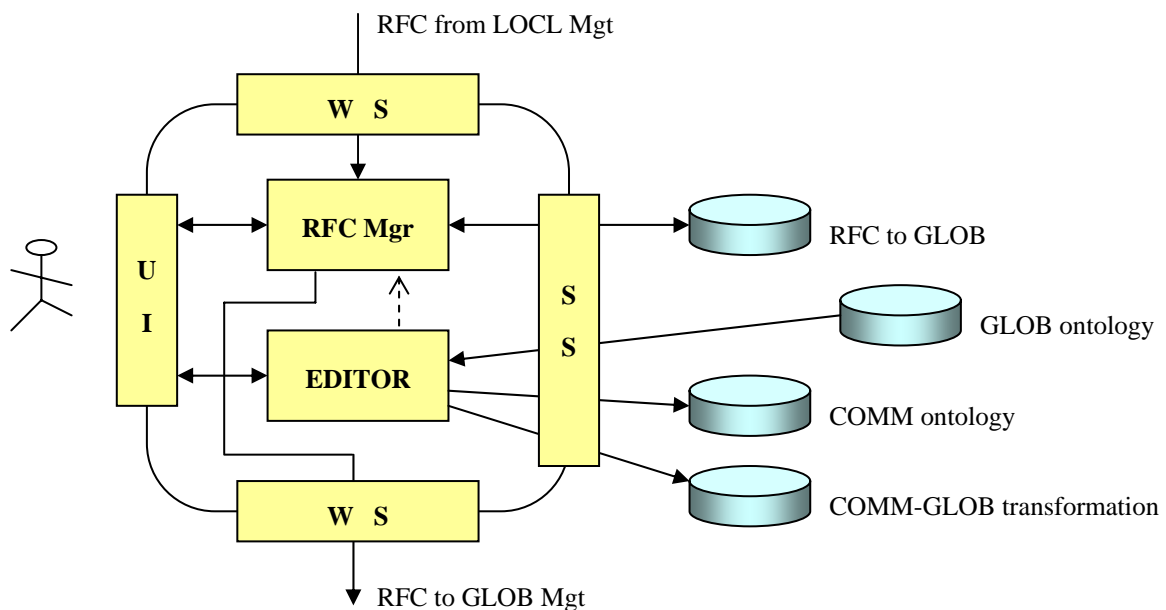


GLOB that should be overcome. Then it can use the specific function of the COMM Management application to issue requests for change or for addition of new vocabulary terms against the GLOB application.

- Periodically update the COMM. Whenever changes occur in the reference GLOB, the mediator is notified and then undertakes a revision process on the COMM in order to align it to the last GLOB version.

2.5.2 Application functionality

The COMM Management application is expected to provide the mediator ontology experts with the functionality depicted in the following figure (UI means user interface, SS means storage system, and WS means web service interface):



- RFC Mgr.** It is the component to manage the requests for change generated from the LOCL Mgt application and to enter those identified during the COMM generation session. These requests include possible improvements of the GLOB data model and new terms to be added to the GLOB vocabulary.
- EDITOR.** It is the COMM ontology Editor, operated by the COMM ontology responsible at the mediator organisation, to define and possibly update the COMM ontology. The outcome includes the COMM-GLOB transformation files to be processed by the translation service (see deliverable D2.2.2).

2.5.3 Application interfaces

The storage system (SS) provides the needed storage and retrieval facilities for the identified data types, namely COMM and GLOB ontologies, COMM-GLOB transformation files, and RFC to GLOB. Also, subscriptions of mediators to the GLOB and the reverse notifications of changes to the GLOB are not shown since it is assumed they are managed by the same storage system.

The SEAMLESS project intends to realise the storage system by adapting and integrating the widely mentioned SRRN platform developed in the frame of the SEEMseed project (see task T3.3). Looking forwards to detailing and using the interface with the SRRN the present version of Editor and Mapper stores the generated objects on the local file system.

The user interface (UI) is proposed in Appendix A.



The web service interface (WS) provides the following RPC invocations:

string GetReceiverList(string username, string password)	
Returns the list of possible receivers to whom the request can be sent according to the registered subscriptions.	
Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the COMM manager. • <i>password</i>. Password of the COMM manager.
Returns:	A XML file containing the list of potential receivers according to the provided parameters and available subscriptions. The receiver list only contains the receiver unique identifier.

int SendRequest(string username, string password, string receiver_id, string subject, string body, int priority, Object attachment)	
Sends a new request for change to selected receiver (GLOB Manager in this specific case). The request is automatically associated to a unique identifier.	
Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the COMM manager sending the request. • <i>password</i>. Password of the COMM manager sending the request. • <i>receiver_id</i>. Identifies the request receiver by means of the value provided by the GetReceiverList method. • <i>subject</i>. Free text describing the subject of the request. • <i>body</i>. Free text explaining the details of the request. • <i>priority</i>. Priority level of the request (0 – low, 1 – medium, 2 – high) • <i>attachment</i>. (optional) A Raw file by which detailing the request (e.g. a data model schema, a vocabulary screenshot, etc.)
Returns:	0 if the operation succeeded, otherwise a negative integer value as an error code.

string GetSentRequestList(string username, string password)	
Returns the list of the requests sent by the identified SEAMLESS user.	
Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the COMM manager. • <i>password</i>. Password of the COMM manager.
Returns:	A XML file containing the list of sent requests according to the provided user identification. The request list only contains the request unique identifier.

string GetReceivedRequestList(string username, string password)	
Returns the list of the requests received by the identified SEAMLESS user.	
Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the COMM manager. • <i>password</i>. Password of the COMM manager.
Returns:	A XML file containing the list of received requests according to the provided user identification. The request list only contains the request unique identifier.

string GetRequestDetails(string username, string password, string request_id)	
Returns all the information related to the request for change identified by the provided identifier.	



Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the COMM manager. • <i>password</i>. Password of the COMM manager. • <i>request_id</i>. The request identifier obtained by the GetSentRequestList method.
Returns:	A XML file containing the information of the identified request. The system is in charge of validating the invocation according to the provided user identification parameters. The request for change information is actually yielded if the <i>request_id</i> is valid and the relevant request has been either sent or addressed to the identified user.

2.6 GLOB Management application

The section describes the functionality of the SEAMLESS application to manage the GLOB ontology to which one or more mediators can refer for the creation of their COMM ontologies. Besides creating a GLOB, the application includes the possibility of cross-referencing other GLOBs for semantic roaming purposes.

2.6.1 Use scenario

An important outcome of the SEAMLESS project is the definition of a “generic” global ontology and two “specialised” global ontologies, respectively for the Building & Construction and Textile sectors. The main activities that the organisations responsible for GLOBs will perform with the support of the GLOB Management application are the following:

- Define the generic GLOB ontology, meaning its data model + taxonomy + vocabulary. The generic GLOB comes first to indicate the mandatory concepts that every GLOB must include in order to assure a satisfactory overlapping with the other GLOBs and then an effective semantic roaming between them.
- Define the specialised GLOBs. The B&C and TEX ontology managers prepare in turn their specialised GLOBs by defining their own data models, taxonomies and vocabularies. An important help comes to them from taking the generic GLOB as basis and inspiration.
- Define further GLOBs. Interested organisations can afterwards decide to set up other GLOBs likely in competition with the existing ones. Even in this case they can take inspiration from the first generic and specialised GLOBs defined within the frame of this project.
- Edit the new GLOB. Every organisation managing a GLOB uses the GLOB application Editor to code its new defined global ontology²⁶. This implies at least describing the data model, coding the taxonomy structure and possibly defining vocabulary terms, while other terms will come later from the subscribing mediators.
- Make the GLOB available to interested mediators. Once defined the GLOB, the organisation managing it makes the GLOB visible to everybody through an open user interface of the GLOB Management application. An interested mediator can decide to adopt it: in that case it has simply to discuss with the organisation responsible for the given GLOB the conditions (and cost) for its use, subscribe to that GLOB, then download the GLOB and use it for generating the relative COMM²⁷.
- Map the GLOB onto other GLOBs. The organisation managing that GLOB is informed about other GLOBs proposed by homologous organisations to the SEAMLESS mediators. It agrees with each of them on the necessity to cross-reference the respective GLOBs, and uses the GLOB application Mapper to relate the concepts of the two GLOBs and generate the relative transformation files²⁸.

²⁶ See requirements UC08, UC09 of deliverable D3.1.

²⁷ See requirements UC01, UC02, UC04 of deliverable D3.1.

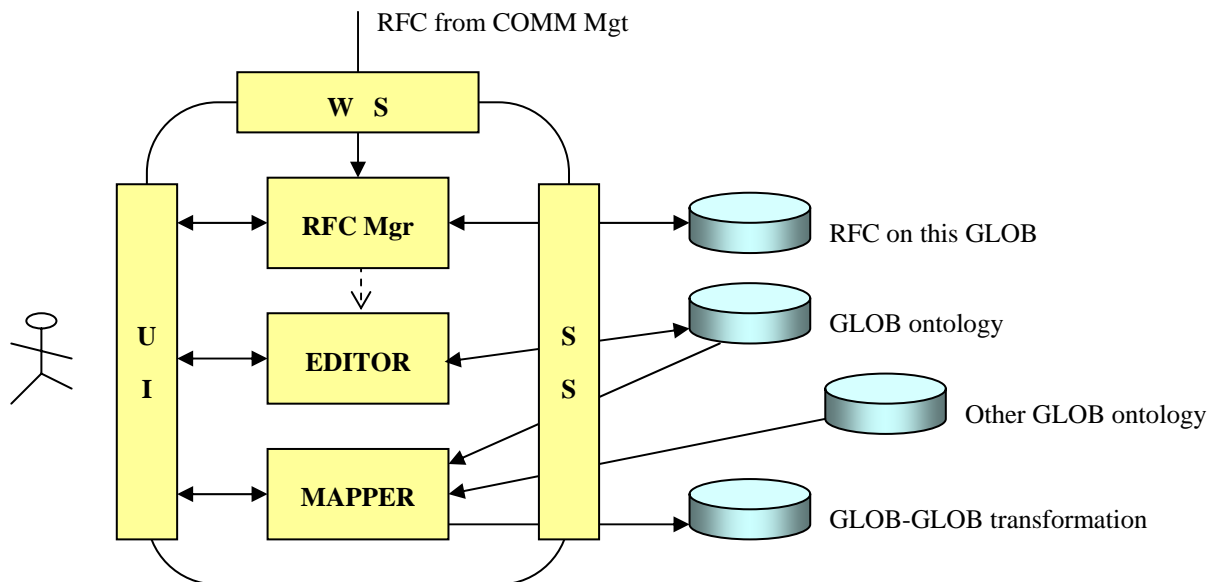
²⁸ See requirements UC11, UC12 of deliverable D3.1.



- Update the GLOB. The organisation managing the GLOB receives, through a GLOB application service, the requests for change from the subscribing mediators. One or twice a year the managing organisation undertakes a regular GLOB revision process taking also into account these requests, delivers a new version and notifies the subscribers²⁹. The GLOB revision, including the analysis of the requests for change, is supported by the GLOB application.
- Control the vocabulary. The organisation managing the GLOB knows that its vocabulary is growing based on the contributions that continuously come from the subscribing mediators. In order to remove garbage and possibly improve or merge terms and definitions, the managing organisation periodically undertakes a vocabulary control procedure by means of the GLOB application, and the result is notified to subscribers.

2.6.2 Application functionality

The COMM Management application is expected to provide the mediator ontology experts with the functionality depicted in the following figure (UI means user interface, SS means storage system, and WS means web service interface):



- RFC Mgr. It is the component to collect and manage the requests for change generated by the COMM Management applications of the subscribing mediators. These requests include possible improvements to the GLOB data model and new terms to be added to the GLOB vocabulary.
- EDITOR. It is the GLOB ontology Editor, operated by the ontology responsible at the GLOB holding organisation, to define and possibly update the GLOB ontology. The editing process is periodical and takes into account the requests for change that have come from the subscribing mediators during the last period.

2.6.3 Application interfaces

The storage system (SS) provides the needed storage and retrieval facilities for the identified data types, namely GLOB ontologies, GLOB-GLOB transformation files, and RFC on this GLOB. Also, subscriptions of other GLOBs to this GLOB and the reverse notifications of changes to this GLOB are not shown since it is assumed they are managed by the same storage system.

²⁹ See requirements UC03, UC05, UC06, UC07 of deliverable D3.1. It is wise to remember that a GLOB Manager cannot propose a change to another GLOB manager.

The SEAMLESS project intends to realise the storage system by adapting and integrating the widely mentioned SRRN platform developed in the frame of the SEEMseed project (see task T3.3). Looking forwards to detailing and using the interface with the SRRN the present version of Editor and Mapper stores the generated objects on the local file system.

The user interface (UI) is proposed in Appendix A.

The web service interface (WS) provides the following RPC invocations:

string GetReceivedRequestList(string username, string password)	
Returns the list of the requests received by the identified SEAMLESS user.	
Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the GLOB manager. • <i>password</i>. Password of the GLOB manager.
Returns:	A XML file containing the list of received requests according to the provided user identification. The request list only contains the request unique identifier.

string GetRequestDetails(string username, string password, string request_id)	
Returns all the information related to the request for change identified by the provided identifier.	
Parameters:	<ul style="list-style-type: none"> • <i>username</i>. Username of the GLOB manager. • <i>password</i>. Password of the GLOB manager. • <i>request_id</i>. The request identifier obtained by the GetSentRequestList method.
Returns:	A XML file containing the information of the identified request. The system is in charge of validating the invocation according to the provided user identification parameters. The request for change information is actually yielded if the <i>request_id</i> is valid and the relevant request has been either sent or addressed to the identified user.



3 Ontology Editor module

This chapter describes the SEAMLESS Ontology Editor, the software module in charge of supporting the creation of an ontology composed by a data model, a taxonomy and a vocabulary.

This Editor is new. It is developed within task T2.1 to fully meet the specific requirements of the SEAMLESS project. It is general enough to be used in all the above-mentioned ontology management applications by adapting to the specific requirements of each of them.

The chapter provides a state-of-the-art analysis and a detailed technical specification of the ontology data structure and the Editor software functionality.

3.1 State-of-the-art

In order to justify the decision to develop a brand new Ontology Editor this section proposes a survey and performs an objective comparison of currently available tools. In particular the analysis is focused on some tools that are well-known in the research community as interesting cases of ontology-oriented editors.

In order to do that, a list of required features is preliminarily defined to measure the compliance level of such tools with the identified SEAMLESS needs. It is not an aim of this task to provide a comprehensive benchmark of all the general features that are normally considered when selecting an ICT tool. Rather we are simply verifying whether the analysed tools can meet those few requirements that are necessary to realise the foreseen solution for building ontologies as basis for translating queries and documents in a collaboration environment made of small and micro companies.

INTEROPERABILITY WITH EXTERNAL SERVICES	
Ontology persistence service	It is required that the Editor can use an external storage system to store, index and retrieve the ontologies it generates and processes.
Subscription and notification	It is required that the Editor can rely on an external service assuring subscription and notification facilities for ontology management. The subscription service is a design choice in order to guarantee the access to the reserved area for authorised users. The notification service informs subscribers of changes occurred in an ontology ³⁰ .
Ontology versioning service	It is required that the Editor can assign the management of ontology versions to an external service. Users are allowed to select among a list of versions of the same ontology in order to update or reuse them as source data in the Editor module or in the Mapper module.
EDITING FACILITIES	
Data model editing	Possibility of defining the ontology data model (data structure) reflecting the ontology concepts ³¹
Data model editing with reusable concepts support	Facilities for defining reusable concepts throughout the data model, thus speeding-up the editing activities.
Data model editing with typed concepts support	Capability of defining typed concepts according to the most common domains (text, integer, date, etc.)
eBusiness oriented data model editing	Specific functions for assisting users in the definition of eBusiness oriented ontologies.

³⁰ See requirements UC01, UC02, UC03 of deliverable D3.1.

³¹ See requirements UC09, UC14 of deliverable D3.1. In case of LOCL data model editing reference to the UC43.



Vocabulary editing	Functions for editing the ontology vocabulary ³² .
Vocabulary translation facilities	Availability of lingual translation facility for vocabulary terms.
Taxonomy editing	Functions for editing the ontology taxonomies ³³ .
Integrated ontology editing	Integration support among data model editor, vocabulary editor and taxonomy editor in terms of data consistency and information validity.
USER INTERFACE	
User profile dependent interface	Some user interface functions can be enabled/disabled depending on the user profile (read-only fields, vocabulary translation).
Technology independent interface semantics	The user interface hides the underlying implementation details of the ontology. In this case, the user is not asked to be aware of the rules regulating the specific implementation
OPEN SOURCE	
Open source license	Possibility of modifying the product source code according to the project requirements.

3.1.1 Apollo CH (<http://apollo.open.ac.uk/index.html>)

Apollo CH is an ontology editor tool for communities of interest that can be used generally in many applications. It uses its own knowledge model, which is based on frames and is independent on any knowledge modelling language.

Despite the fact that the *Apollo CH* contains language independent knowledge model, concept of its knowledge model is very similar to OKBC knowledge model in other words the *Apollo CH* deals with objects like classes, instances, slots, facets etc.

The main features of the product are enlisted below:

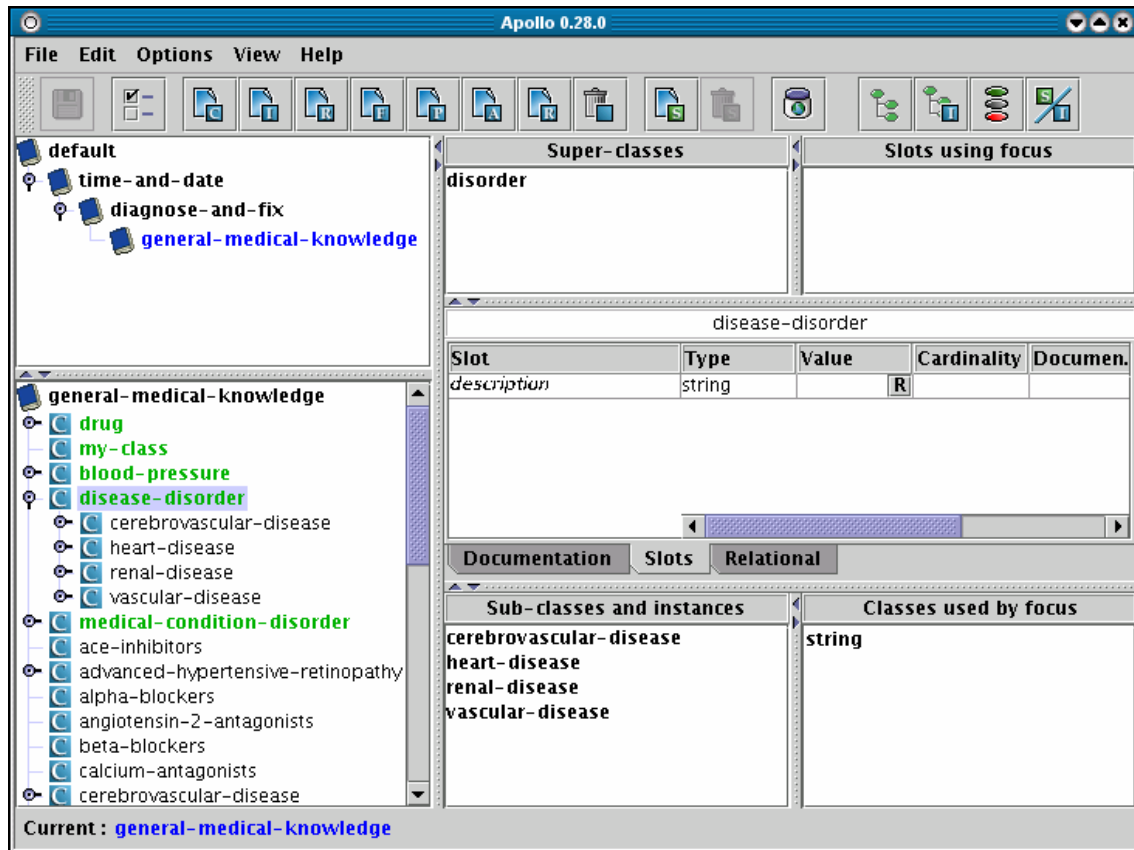
- Functionalities aimed at improving efficiency of user work when editing large ontologies or knowledge bases. These include mainly new ways of navigation in large ontologies or knowledge bases.
- Functionalities aimed at allowing the user to do ontology consolidation on a high level of abstraction. The methodology of collaborative creation of knowledge bases expected the user using rather low-level tools for comparing ontologies and integrating user's contribution with the shared consolidated knowledge base. *Apollo CH* has been equipped with modules providing the user with a means how to carry out this difficult task on a higher level of abstraction and thus easier, more reliably and more efficiently.
- Functionalities aimed at developing robust knowledge base. They ensure that the knowledge base produced by *Apollo CH* meets the necessary constrains.
- Ontologies can be exported into *RDF*, *XML*, *Meta* and *OCML* formats.

A typical screenshot of the application is:

³² See requirements UC11, UC13 of deliverable D3.1. As for the note above, consider the UC43 in case of vocabulary editing.

³³ See requirements UC11, UC13 of deliverable D3.1. As for the note above, consider the UC43 in case of taxonomy editing.





3.1.2 Protégé 2000 (<http://protege.stanford.edu/index.html>)

Protégé is a free, open-source platform that provides a growing user community with a suite of tools to construct domain models and knowledge-based applications with ontologies. At its core, Protégé implements a rich set of knowledge-modelling structures and actions that support the creation, visualization, and manipulation of ontologies in various representation formats. Protégé can be customized to provide domain-friendly support for creating knowledge models and entering data. Further, Protégé can be extended by way of a plug-in architecture and a Java-based Application Programming Interface (API) for building knowledge-based tools and applications. The Protégé platform supports two main ways of modelling ontologies:

- The *Protégé-Frames* editor enables users to build and populate ontologies that are *frame-based*, in accordance with the Open Knowledge Base Connectivity protocol (OKBC). In this model, an ontology consists of a set of classes organized in a subsumption hierarchy to represent a domain's salient concepts, a set of slots associated to classes to describe their properties and relationships, and a set of instances of those classes - individual exemplars of the concepts that hold specific values for their properties.
- The *Protégé-OWL* editor enables users to build ontologies for the Semantic Web, in particular in the W3C's Web Ontology Language (OWL). "An OWL ontology may include descriptions of classes, properties and their instances. Given such an ontology, the OWL formal semantics specifies how to derive its logical consequences, i.e. facts not literally present in the ontology, but entailed by the semantics.

The main features of the product are enlisted below. In general:

- Ontologies can be exported into a variety of formats, including RDF(S), OWL, XML SCHEMA.
- JDBC back-end design.



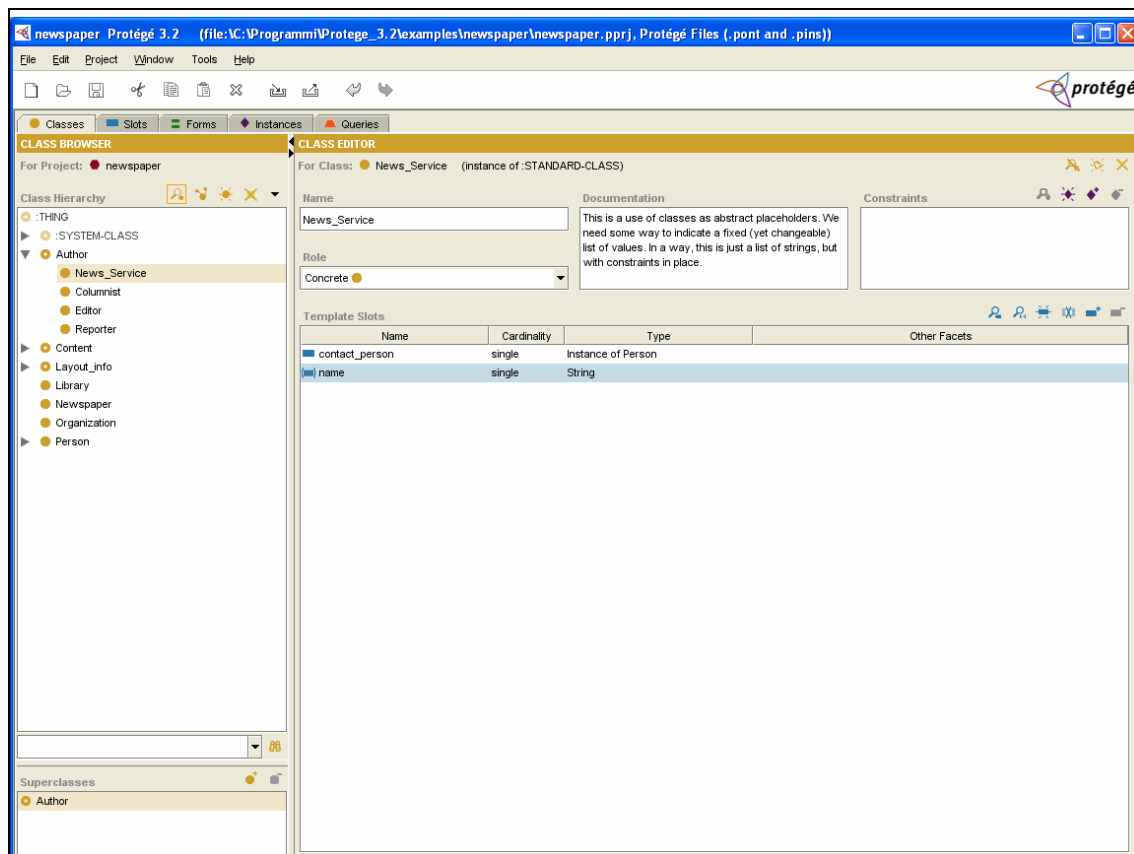
Protégé-Frames:

- A wide set of user interface elements that can be customized to enable users to model knowledge and enter data in domain-friendly forms.
- A plug-in architecture that can be extended with custom-designed elements, various storage formats (e.g., RDF, XML, HTML, and database back-ends), and additional support tools (e.g., for ontology management, ontology visualization, inference and reasoning, etc.).
- A Java-based Application Programming Interface (API) that makes it possible for plug-ins and other applications to access, use, and display ontologies created with Protégé-Frames.

Protégé-OWL:

- Load and save OWL and RDF ontologies.
- Edit and visualize classes, properties.
- Define logical class characteristics as OWL expressions.
- Edit OWL individuals for Semantic Web markup.

A typical screenshot of the application is:



3.1.3 Differential Ontology Editor DOE (<http://opales.ina.fr/public>)

It is a simple ontology editor which allows the user to build an ontology. The specification process is divided into three steps.

In the first step, the user is invited to build taxonomies of concepts and relations, explicitly justifying the position of each item (notion) in the hierarchy. For each notion, the user builds a definition following four principles which come from the *Differential Semantics* theory. Hence, the user has to explicit why a



notion is similar but more specific than its parent (two principles), and why this notion is similar but different from its siblings (two others principles). The user can also add synonyms and encyclopaedic definitions in a few languages for all notions.

In a second step, the two taxonomies are considered from an extensional semantics point of view. The user can augment them with new entities (defined) or add constraints onto the domains of the relations.

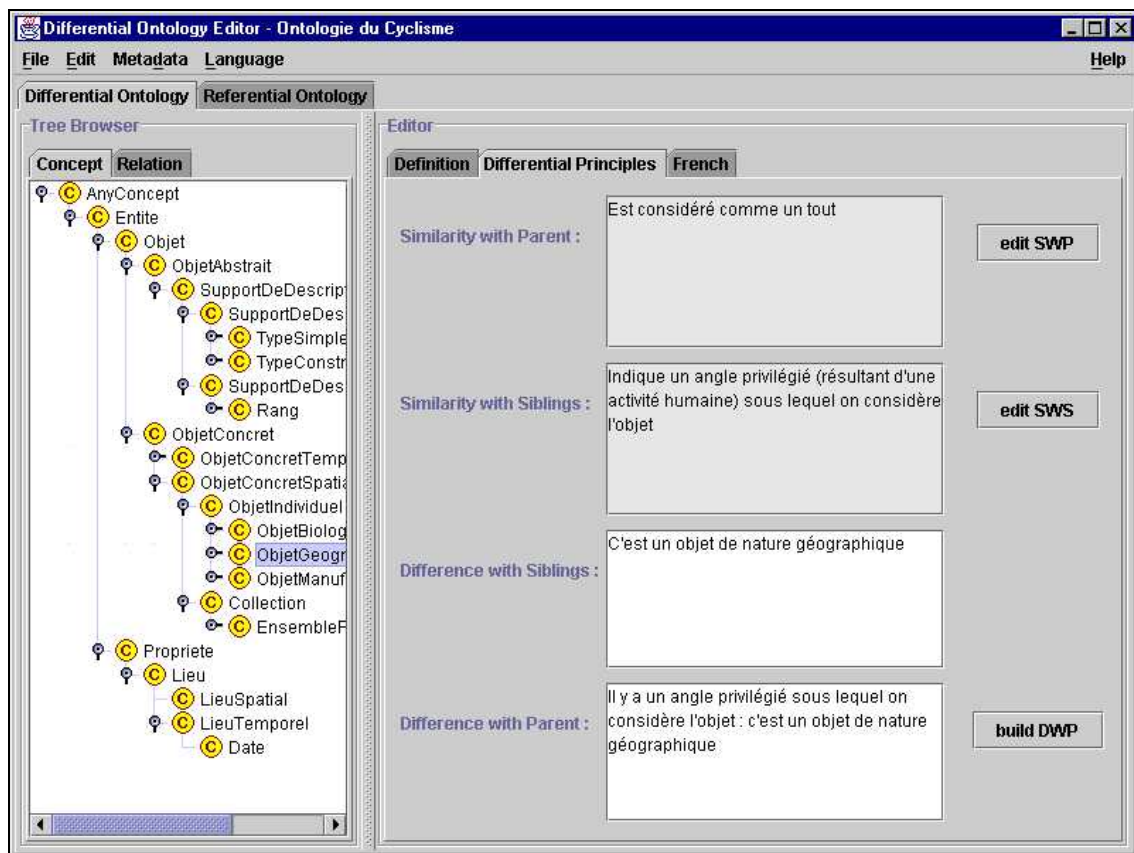
Finally, in a third step, the ontology can be translated into a knowledge representation language, which allows to use it in an appropriate ontology-based system or to import it into another ontology-building tool to specify it further: RDFS, OWL, DAML + OIL, OIL, CGXML (a language to specify conceptual graphs).

DOE is not intended as a full ontology development environment: it will not actively support many activities that are involved traditionally in ontology construction, such as advanced formal specification dealt with by tools like Protégé 2000. It is rather a complement of others editors, offering linguistics-inspired techniques which attach a lexical definition to the concepts and relations used, and justify their hierarchies from a theoretical, human-understandable point of view.

The main features of the product are enlisted below:

- It creates lattice of concepts and relationships between concepts, plus a set of instances. Concepts cannot be defined intentionally with constraints. Only types of the domains of relationships can be specified. No axiom editor is provided.
- Base language is XML and CGXML.
- No graphical view but tree view.
- Term definitions, synonyms and preference; methodology for differential definitions.
- Multilingual support.

A typical screenshot of the application is:

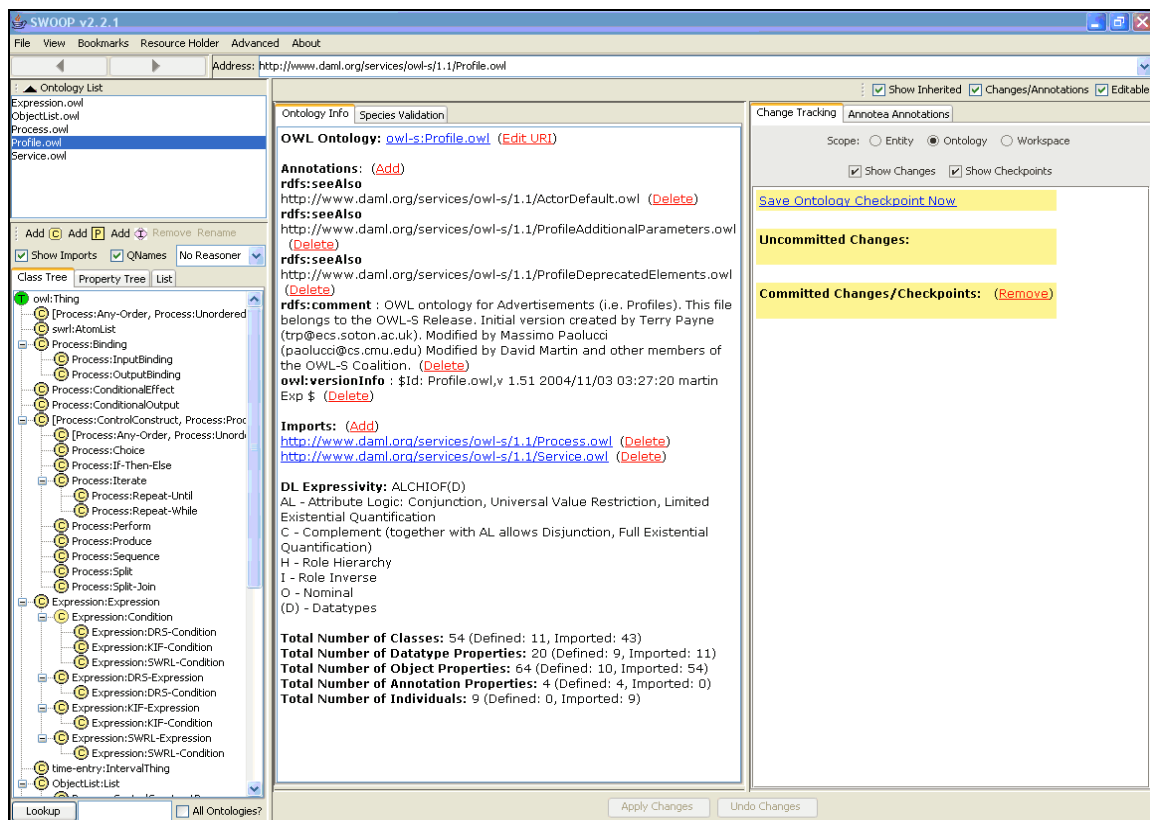


3.1.4 SWOOP (<http://www.mindswap.org/2004/SWOOP>)

It is an editor for creating, editing, and debugging OWL ontologies and taking the standard web browser as the basic UI paradigm. Swoop includes many of the familiar features of a Web browser such as an address bar and history buttons, bookmarks, hypertextual navigation etc. and applies them to the problem of browsing and editing Web based ontologies. The main features of the product are enlisted below:

- Designed for OWL, meant for rapid and easy browsing and development of OWL ontologies.
- It is simple to load ontologies from the web and to navigate within and between them. An address bar where the URI of the ontology (or class/property/individual) can be entered directly for loading; hyperlink based navigation across ontological entities (address bar URL changes accordingly); history buttons (Back, Next etc) for traversal; and bookmarks that can be saved for later reference.
- Multiple ontologies may be loaded at the same time.
- Ontologies, classes, properties, and individuals are rendered in a high level, accessible manner.
- One can “view the source” of ontologies and their entities in a number of common syntaxes (e.g. RDF/XML, the OWL Abstract Syntax, Turtle).
- OWL reasoners can be integrated for subsumption, consistency checking etc. -- default reasoners include a RDFS-like simple reasoner and Pellet, a Description Logic Tableaux Reasoner.
- Ontology change management with extensive rollback and undo mechanisms.
- Share annotations on ontologies using the annotation protocol. Also attach and distribute ontology change sets with annotations.
- Search across multiple ontologies and “find all references” of an OWL named entity.
- Compare entities using a resource holder. Export ontologies.

A typical screenshot of the application is:



3.1.5 OntoTerm (<http://www.ontoterm.com>)

OntoTerm is a new Terminology Management System. It addresses two major issues:

- **Conceptual modelling:** the object domain or subject field must be conceptually structured prior to entering language-specific terms. The construction of an ontology, i.e. a conceptually structured body of language-independent knowledge, is thus the first step in the construction of a termbase.
- **Terminology information exchange:** OntoTerm implements the ISO standard for terminology exchange: Martif (ISO 1220) and all the data categories from the CLS Framework (ISO 1620).

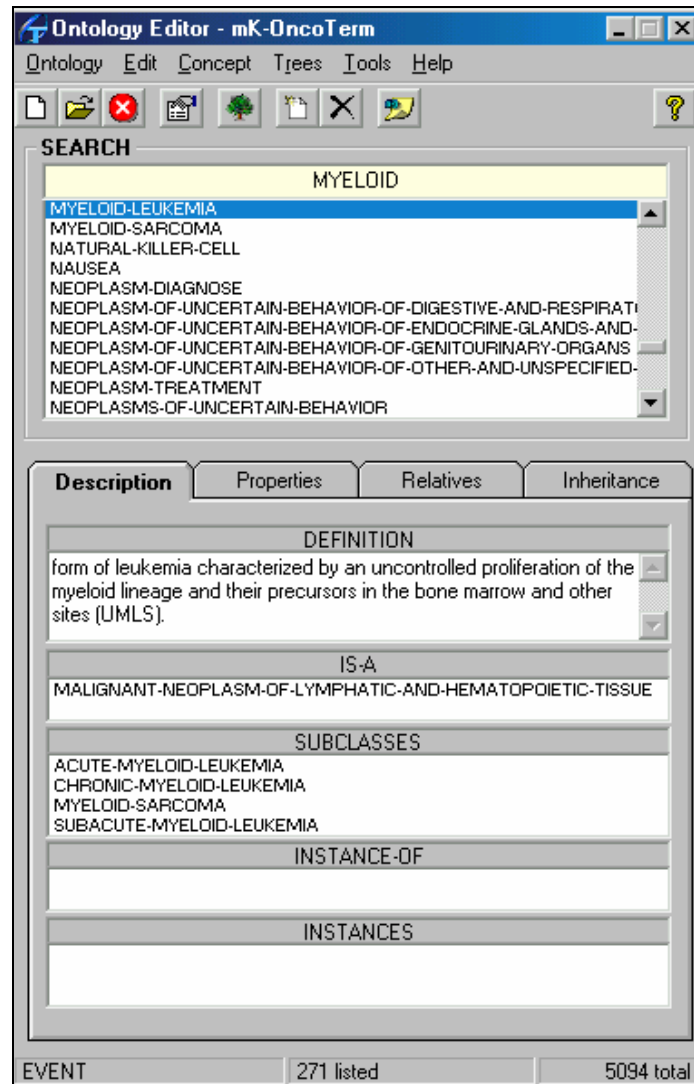
If people are familiar with conventional terminology database management systems, they will probably find OntoTerm rather different. To begin with, OntoTerm does not allow them to enter terms in a termbase unless you have previously entered and defined a concept explicitly in the ontology.

The main features of the product are enlisted below:

- **Concept-based:** the Ontology Editor provides true conceptual modelling. people can start with a pre-defined set of top-level, generic concepts to help them start out or they can start from scratch and create their own concepts. Concepts are defined by the relationships they hold to one another, rather than by their name, which should be regarded solely as a mnemonic, rather than a language-specific. Users are free to enrich this conceptual relationships as much as they want, so that the resulting ontology provides domain-specific knowledge about a given domain.
- **HTML publishing:** people are able to publish their data easily and effortlessly. OntoTerm generates web pages using the information they have entered in the databases. Generate pages individually, as subsets, or the whole termbase using the HTML Report Generator.
- **Easy-to-use user interface.** Everything in OntoTerm is where people would expect it to be. It is highly graphical and uses context menus extensively. If they get lost try right-clicking on objects, that may give them a hint as to how to proceed from there.
- **ISO compliant.** OntoTerm implements the ISO standard for terminology exchange: Martif (ISO 1220) and all the data categories from the CLS Framework (ISO 1620). In the future people will be able to define their own set of data categories, but it is very unlikely that you will ever need to do so, as the full set of the comprehensive data categories from the CLS Framework are already at their fingertips. The TermBase Editor makes the complexity of using these data categories properly a trifle by allowing the user to assign only the data categories appropriate for the selected element.

A typical screenshot of the application is:





3.1.6 TopBraid Composer

TopBraid Composer is a professional development environment for W3C's Semantic Web standards RDF Schema, the OWL Web Ontology Language, the SPARQL Query Language and the Semantic Web Rule Language (SWRL). Composer can be used to edit RDFS/OWL files in various formats, and also provides scalable database back ends (Jena, Oracle 10g and Sesame) as well as multi-user support.

TopBraid Composer is built upon the Eclipse platform and uses Jena as its underlying API. TopBraid Composer's user interface reflects the OWL standards much closer, and uses the official W3C terminology such as 'owl:equivalentClass': this requires some understanding of the OWL/RDF syntax.

The main feature of the product are enlisted below:

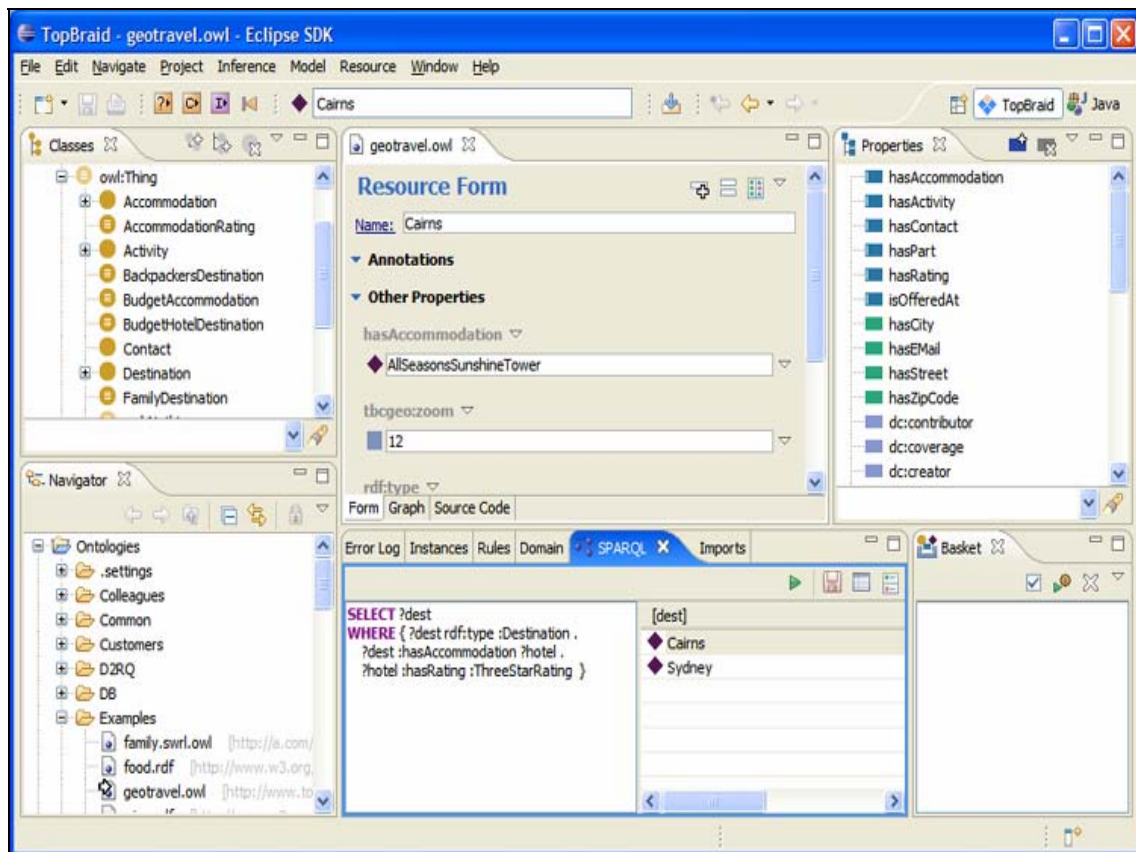
- Commercial platform.
- Built upon the Eclipse platform and uses Jena as its underlying API: this supports the rapid development of semantic applications in a single platform.
- Reflects the OWL standards deeply: it is much closer than Protégé design.
- Multi user support: Composer can be used in a single user mode working with ontologies stored as files or in a database. Multi-user mode enables multiple users to simultaneously access and



edit OWL ontologies and RDF data. Each user can do incremental changes on his local view and then commit the changes to the server.

- Ontology-Driven Forms: it creates forms based on class definitions, and presents these forms as its primary editing mechanism for RDFS/OWL resources. The forms can be configured to display one or two columns and provide drag-and-drop support throughout. Complex resource descriptions such as class definitions and rules can be edited with syntax highlighting and auto-completion.
- Imports and namespace management.
- Import of Databases, UML, XML Schema and Spreadsheets.

A typical screenshot of the application is:



3.1.7 Ontology editor comparison

This section concludes the analysis by identifying the deficiencies of the considered software tool with respect to the requirements of the SEAMLESS Ontology Editor as follows:

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

SEAMLESS requirements	Apollo CH	Protégé 2000	DOE	SWOOP	OntoTerm	TopBraid Composer
INTEROPERABILITY WITH EXTERNAL SERVICES						
Persistence service	NO	NO	NO	NO	NO	YES**



Subscription/notification	NO	NO	NO	NO	NO	NO
Versioning service	NO	YES*	YES*	YES*	NO	YES**
EDITING FACILITIES						
Data model editing	YES	YES	NO	NO	NO	YES
Data model editing + reusable concepts support	NO	YES	NO	NO	NO	YES
Data model editing with typed concepts support	YES	YES	NO	NO	NO	YES
e-business oriented editing	NO	NO	NO	NO	NO	NO
Vocabulary editing	NO	NO	NO	YES	YES	NO
Vocabulary translation facilities	NO	NO	NO	YES	YES	NO
Taxonomy editing	NO	YES	NO	YES	NO	NO
Integrated ontology editing	NO	NO	NO	YES	NO	NO
USER INTERFACE						
User profile dependent interface	NO	YES	NO	NO	NO	NO
Technology independent interface semantic	YES	YES	YES	YES	YES	YES
OPEN SOURCE						
Open source license	NO	YES	NO	YES	NO	NO

In the row referencing to the “Versioning service” the symbol YES* means that the versioning service interface with an external service is not granted, instead the ontology versioning on the local file system is supported by the tools at issue. In turn the symbol YES** associated to properties of TopBraid Composer mean that it works with external functions but these are predefined (not open).

In conclusion of this analysis we can see that none of these tools provides the Editor 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 D2.1.1.

3.2 Ontology data structure

The Editor supports creation and modification of SEAMLESS ontologies, following the description provided in the previous sections. Hence, a relevant issue is how to arrange the ontology knowledge, namely which persistent means to adopt and which data structures are the most suitable to define the contents.

About persistence, a file-based persistent strategy has been considered the most appropriate in the short term, and also considering the future interface to the SRRN distributed storage system:

- It is the most common way to represent persistent information. All the systems are nowadays able to manage a file-system storage, then this requirement should have no effective impact on the specific implementation.
- A file-based persistent mechanism can be scaled up to more reliable and distributed architectures without relevant efforts. For instance, it is quite straightforward storing file-based information in a database management system or in a web service oriented storage, since the web service communication protocols support the attachment of files.



The issue addressing the choice of which is the most suitable data format to be adopted, relies on the choice of XML as the starting point of the discussion, since XML is the de-facto standard in terms of file-based data representation.

Once defined the outmost technical specification about the Editor module, this section goes through the specific file formats and contents. In particular, the following files are required to define a SEAMLESS-compliant ontology: data model (collaboration, company, negotiation, product and service), vocabulary and taxonomy.

3.2.1 Data model structure

The ontology data model is defined by means of four files reflecting the logical organisation of the data model concepts in collaboration model, company model, negotiation model and product and service model. Since the same specifications can be applied to any of the already mentioned file, from this point on, this section will generally mention a data model file instead of explicitly specifying the four entities.

According to the SEAMLESS specification, the data model is intended to be the data structure to define the ontology concepts. The most fine-grained way for expressing data structures within a XML context is represented by the XML Schema Definition (XSD) standard. The rationale for adopting XSD as the language by which the SEAMLESS data model file is coded is:

- As best practice, XML documents should reference a XSD file in order to be validated. Since SEAMLESS documents are supposed to be XML, the data model file provides the reference schema for the documents.
- XSD is flexible enough to represent the data structures that SEAMLESS is asked to support, such companies, products, business documents and their relevant typed properties.
- XSD supports annotations on tag definitions. This let the Editor module leveraging extra information for properly save and load data model data and, taking into account that the final users are not aware of working on XSD.
- API for accessing and manipulating XSD documents are available for the most important programming environment.

XSD is a low level implementation choice that should have no impact to the final user perspective on the data structures. Hence, the Editor module is in charge of:

- Store the high level data structure provided by the users against an equivalent XSD schema.
- Retrieve the high level data structure from a previously stored XSD schema.

A relevant consequence of this scenario is that the Editor module cannot retrieve information from a generic XSD file but only from those XSD compliant with the Editor output format. This limitation is mainly due to the need of providing an high level view on the data structures in order to address the application at the widest typology of users, not only at those concerned in ICT and aware of XML. Then, on the data model point of view, the Editor module can be perceived as an XSD Editor for SEAMLESS users, typically used to deal with table names and fields domain instead of XML and XSD.

Let now examine how the Editor manages the user actions in a XSD fashion.



User perspective concept	XSD representation
<p>Root type</p> <p>A root type is intended as a data model type that can play the role of first level element within a document, e.g. Order, Company.</p>	<pre> 1<xsd:element name="" type="" nillable="" /> 2<xsd:complexType name="" extra:id=""> 3 <xsd:annotation> 4 <xsd:appinfo source="data-type"></xsd:appinfo> 5 <xsd:appinfo source="name"></xsd:appinfo> 6 <xsd:appinfo source="description"></xsd:appinfo> 7 <xsd:appinfo source="read-only"></xsd:appinfo> 8 <xsd:appinfo source="term-representation"></xsd:appinfo> 9 </xsd:annotation> 10 <xsd:sequence> 11 </xsd:sequence> 12</xsd:complexType> </pre> <ul style="list-style-type: none"> • name (1): user defined type name • type (1): Editor defined type name • nillable(1): always true since the same XSD can be contains multiple root types • name (2): Editor defined type name • extra:id (2): unique term identifier, required only when COMM editing is performed • line 4: constant value (root-type) • line 5: user defined type name • line 6: user defined type description • line 7: the type cannot be edited (normally false, true only when editing the COMM and the type has been defined in the GLOB). • line 8: vocabulary term associated to this type name • line 10-11: attribute list within this type (<i>see below</i>)
<p>Component type</p> <p>A component type is intended as a data model type that can be used only inside a root type or another component. It provides a modular modelling pattern.</p>	<pre> 1<xsd:complexType name="" extra:id=""> 2 <xsd:annotation> 3 <xsd:appinfo source="data-type"></xsd:appinfo> 4 <xsd:appinfo source="name"></xsd:appinfo> 5 <xsd:appinfo source="description"></xsd:appinfo> 6 <xsd:appinfo source="read-only"></xsd:appinfo> 7 <xsd:appinfo source="term-representation"></xsd:appinfo> 8 </xsd:annotation> 9 <xsd:sequence> 10 </xsd:sequence> 11</xsd:complexType> </pre> <ul style="list-style-type: none"> • name (1): Editor defined type name • extra:id (1): unique term identifier, required only when COMM editing is performed • line 3: constant value (type) • line 4: user defined type name • line 5: user defined type description • line 6: the type cannot be edited (normally false, true only when COMM editing is performed and the type has been defined in the GLOB). • line 7: vocabulary term associated to this type name • line 9-10: attribute list within this type (<i>see below</i>)

<p>Attribute in type Anytime an attribute is added to a type, regardless of the attribute domain, this line is added to the sequence section of the relevant type.</p>	<pre>1<xsd:element name="" type="" minOccurs="" maxOccurs="" /></pre> <ul style="list-style-type: none"> • name (1): user defined attribute name • type (1): Editor defined type name referencing an XSD type definition providing the attribute domain information (<i>see below for allowed domain list</i>) • minOccurs (1): minimum number of occurrences for this attribute • maxOccurs (1): maximum number of occurrences for this attribute
<p>Date attribute Attribute defined in a type with date domain.</p>	<pre>1<xsd:simpleType name="" extra:id=""> 2 <xsd:annotation> 3 <xsd:appinfo source="data-type"></xsd:appinfo> 4 <xsd:appinfo source="description"></xsd:appinfo> 5 <xsd:appinfo source="attr-class"></xsd:appinfo> 6 <xsd:appinfo source="read-only"></xsd:appinfo> 7 <xsd:appinfo source="term-representation"></xsd:appinfo> 8 </xsd:annotation> 9 <xsd:restriction base="xsd:date"> 10 <xsd:minInclusive value="" /> 11 <xsd:maxInclusive value="" /> 12 </xsd:restriction> 13</xsd:simpleType></pre> <ul style="list-style-type: none"> • name (1): Editor defined attribute name • extra:id (1): unique term identifier, required only when COMM editing is performed • line 3: constant value (implicit-type) • line 4: user defined attribute description • line 5: constant value (Date) • line 6: the attribute cannot be edited (normally false, true only when COMM editing is performed and the attribute has been defined in the GLOB). • line 7: vocabulary term associated to this attribute name • value (10): optional minimum date value • value (11): optional maximum date value
<p>Date time attribute Attribute defined in a type with date time domain.</p>	<p>The same as date attribute except for:</p> <ul style="list-style-type: none"> • line 5: constant value (Date Time) • base (9): constant value (xsd:dateTime) • value (10): optional minimum date time value • value (11): optional maximum date time value
<p>Duration attribute Attribute defined in a type with duration domain.</p>	<p>The same as date attribute except for:</p> <ul style="list-style-type: none"> • line 5: constant value (Duration) • base (9): constant value (xsd:duration) • value (10): optional minimum duration value • value (11): optional maximum duration value
<p>Time attribute Attribute defined in a type with time domain.</p>	<p>The same as date attribute except for:</p> <ul style="list-style-type: none"> • line 5: constant value (Time) • base (9): constant value (xsd:time) • value (10): optional minimum time value • value (11): optional maximum duration value

<p>Integer attribute Attribute defined in a type with integer domain.</p>	<p>The same as date attribute except for:</p> <ul style="list-style-type: none"> line 5: constant value (Integer) base (9): constant value (xsd:integer) value (10): optional minimum integer value value (11): optional maximum integer value
<p>Decimal attribute Attribute defined in a type with decimal domain.</p>	<p>The same as date attribute except for:</p> <ul style="list-style-type: none"> line 5: constant value (Decimal) base (9): constant value (xsd:decimal) value (10): optional minimum decimal value value (11): optional maximum decimal value
<p>Text attribute Attribute defined in a type with text domain. Text domain is a free text, not subject to any translation process.</p>	<pre> 1<xsd:simpleType name="" extra:id=""> 2 <xsd:annotation> 3 <xsd:appinfo source="data-type"></xsd:appinfo> 4 <xsd:appinfo source="description"></xsd:appinfo> 5 <xsd:appinfo source="attr-class"></xsd:appinfo> 6 <xsd:appinfo source="read-only"></xsd:appinfo> 7 <xsd:appinfo source="term-representation"></xsd:appinfo> 8 </xsd:annotation> 9 <xsd:restriction base="xsd:string"> 10 <xsd:minLength value="" /> 11 <xsd:maxLength value="" /> 12 <xsd:pattern value="" /> 13 </xsd:restriction> 14</xsd:simpleType> </pre> <ul style="list-style-type: none"> name (1): Editor defined attribute name extra:id (1): unique term identifier, required only when COMM editing is performed line 3: constant value (implicit-type) line 4: user defined attribute description line 5: constant value (Text) line 6: the attribute cannot be edited (normally false, true only when COMM editing is performed and the attribute has been defined in the GLOB). line 7: vocabulary term associated to this attribute name value (10): optional minimum text length value (11): optional maximum text length value (12): optional regular expression for text validation
<p>Term attribute Attribute defined in a type with term domain. Term domain is represented by any vocabulary item. Hence, term domain is subject to translation process.</p>	<pre> 1<xsd:simpleType name="" extra:id=""> 2 <xsd:annotation> 3 <xsd:appinfo source="data-type"></xsd:appinfo> 4 <xsd:appinfo source="description"></xsd:appinfo> 5 <xsd:appinfo source="attr-class"></xsd:appinfo> 6 <xsd:appinfo source="read-only"></xsd:appinfo> 7 <xsd:appinfo source="term-representation"></xsd:appinfo> 8 </xsd:annotation> 9 <xsd:restriction base="xsd:string"> 10 <xsd:enumeration extra:id="" value=""/> 11 </xsd:restriction> 12</xsd:simpleType> </pre> <ul style="list-style-type: none"> name (1): Editor defined attribute name

	<ul style="list-style-type: none"> • extra:id (1): unique term identifier, required only when COMM editing is performed • line 3: constant value (implicit-type) • line 4: user defined attribute description • line 5: constant value (Term) • line 6: the attribute cannot be edited (normally false, true only when COMM editing is performed and the attribute has been defined in the GLOB). • line 7: vocabulary term associated to this attribute name • line 10: optional list of allowed vocabulary terms representing the domain for this attribute. If this list is empty, all the vocabulary • value (10): unique term identifier, required only when COMM editing is performed • value (10): vocabulary term
<p>Numeric parameter attribute</p> <p>Attribute defined in a type representing a generic numeric parameter and its relevant value. A numeric parameter is a vocabulary term specialised in representing numeric information, mostly for product definition. Differently from decimal/integer attribute, this attribute is vocabulary dependent.</p> <p><i>(See vocabulary structure for more details)</i></p>	<pre> 1<xsd:complexType name="" extra:id=""> 2 <xsd:annotation> 3 <xsd:appinfo source="data-type"></xsd:appinfo> 4 <xsd:appinfo source="description"></xsd:appinfo> 5 <xsd:appinfo source="attr-class"></xsd:appinfo> 6 <xsd:appinfo source="read-only"></xsd:appinfo> 7 <xsd:appinfo source="term-representation"></xsd:appinfo> 8 </xsd:annotation> 9 <xsd:sequence> 10 <xsd:element name="param-term" type="xsd:string" minOccurs="1" maxOccurs="1"/> 11 <xsd:element name="param-value" type="xsd:string" minOccurs="1" maxOccurs="1"/> 12 </xsd:sequence> 13</xsd:complexType> </pre> <ul style="list-style-type: none"> • name (1): Editor defined attribute name • extra:id (1): unique term identifier, required only when COMM editing is performed • line 3: constant value (implicit-type) • line 4: user defined attribute description • line 5: constant value (Num Param) • line 6: the attribute cannot be edited (normally false, true only when COMM editing is performed and the attribute has been defined in the GLOB). • line 7: vocabulary term associated to this attribute name • line 10: this element contains the numeric parameter (numeric term) chosen by the user at document definition time • line 11: this element contains the value provided by the users at document definition time (typically a string representing a numeric value)
<p>Enumerative parameter attribute</p> <p>Attribute defined in a type representing a generic enumerative parameter and its relevant value(s). An enumerative</p>	<p>The same as numeric parameter attribute, except for:</p> <ul style="list-style-type: none"> • line 5: constant value (Enum Param) • line 10: this element contains the enumerative parameter (enumerative term) chosen by the user at document definition time • line 11: this element, with possible multiple cardinality, contains the value provided by the users at document definition time (a vocabulary term included in the enumerative term declaration)

<p>parameter is a special vocabulary term related to a set of other terms, mostly for product definition (e.g. colour related to red, green, blue, etc.). The actual value assumed by this attribute is vocabulary dependent. (See vocabulary structure for more details)</p>	
<p>Type attribute Attribute defined in a type whose domain is represented by a referenced component type.</p>	<pre> 1<xsd:complexType name="" extra:id=""> 2 <xsd:annotation> 3 <xsd:appinfo source="data-type"></xsd:appinfo> 4 <xsd:appinfo source="description"></xsd:appinfo> 5 <xsd:appinfo source="attr-class"></xsd:appinfo> 6 <xsd:appinfo source="read-only"></xsd:appinfo> 7 <xsd:appinfo source="term-representation"></xsd:appinfo> 8 </xsd:annotation> 9 <xsd:complexContent> 10 <xsd:extension base="" extra:id=""/> 11 </xsd:complexContent> 12</xsd:complexType> </pre> <ul style="list-style-type: none"> • name (1): Editor defined attribute name • extra:id (1): unique term identifier, required only when COMM editing is performed • line 3: constant value (implicit-type) • line 4: user defined attribute description • line 5: constant value (Type) • line 6: the attribute cannot be edited (normally false, true only when COMM editing is performed and the attribute has been defined in the GLOB). • line 7: vocabulary term associated to this attribute name • base (10): Editor defined type name representing the component type referenced by this attribute. • extra:id (10): unique term identifier of the referenced type, required only when COMM editing is performed
<p>Taxonomy attribute Attribute defined in a type with taxonomy term domain. A taxonomy term domain is represented by any vocabulary term located in the taxonomy.</p>	<pre> 1<xsd:simpleType name="" extra:id=""> 2 <xsd:annotation> 3 <xsd:appinfo source="data-type"></xsd:appinfo> 4 <xsd:appinfo source="description"></xsd:appinfo> 5 <xsd:appinfo source="attr-class"></xsd:appinfo> 6 <xsd:appinfo source="taxonomy-node-list"></xsd:appinfo> 7 <xsd:appinfo source="read-only"></xsd:appinfo> 8 <xsd:appinfo source="term-representation"></xsd:appinfo> 9 </xsd:annotation> 10 <xsd:restriction base="xsd:string"/> 11</xsd:simpleType> </pre>

<p>(See taxonomy structure for more details)</p>	<ul style="list-style-type: none"> • name (1): Editor defined attribute name • extra:id (1): unique term identifier, required only when COMM editing is performed • line 3: constant value (implicit-type) • line 4: user defined attribute description • line 5: constant value (Taxonomy Term) • line 6: list of taxonomy view-point and/or sector providing a restriction information on the allowed taxonomy terms. This means that only a taxonomy term defined beneath the provided sectors and view-points can be accepted for this attribute. The taxonomy-node-list information respects the following syntax: (<sector name>/[<view-point name>][!!!])+ • line 7: the attribute cannot be edited (normally false, true only when COMM editing is performed and the attribute has been defined in the GLOB). • line 8: vocabulary term associated to this attribute
<p>Binary attribute Attribute defined in a type with binary domain. A binary domain is a HTTP link to a binary resource available on the network.</p>	<pre> 1<xsd:complexType name="" extra:id=""> 2 <xsd:annotation> 3 <xsd:appinfo source="data-type"></xsd:appinfo> 4 <xsd:appinfo source="description"></xsd:appinfo> 5 <xsd:appinfo source="attr-class"></xsd:appinfo> 6 <xsd:appinfo source="read-only"></xsd:appinfo> 7 <xsd:appinfo source="term-representation"></xsd:appinfo> 8 </xsd:annotation> 9 <xsd:simpleContent> 10 <xsd:extension base="xsd:string"> 11 <xsd:attribute name="href" type="xsd:string" use="required"/> 12 <xsd:attribute name="mime-type" type="xsd:string" use="required"/> 13 </xsd:extension> 14 </xsd:simpleContent> 15</xsd:complexType> </pre> <ul style="list-style-type: none"> • name (1): Editor defined attribute name • extra:id (1): unique term identifier, required only when COMM editing is performed • line 3: constant value (implicit-type) • line 4: user defined attribute description • line 5: constant value (Binary) • line 6: the attribute cannot be edited (normally false, true only when COMM editing is performed and the attribute has been defined in the GLOB). • line 7: vocabulary term associated to this attribute name • line 11: link to the attached resource provided at document definition time • line 12: attached resource mime-type

3.2.2 Vocabulary structure

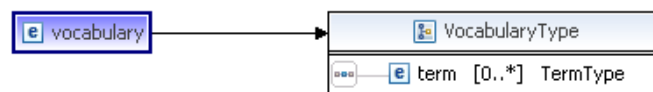
Vocabulary represents the complete list of allowed terms within the ontology. Therefore, only terms from the vocabulary can be recognised and translated when a SEAMLESS document is exchanged across the network.



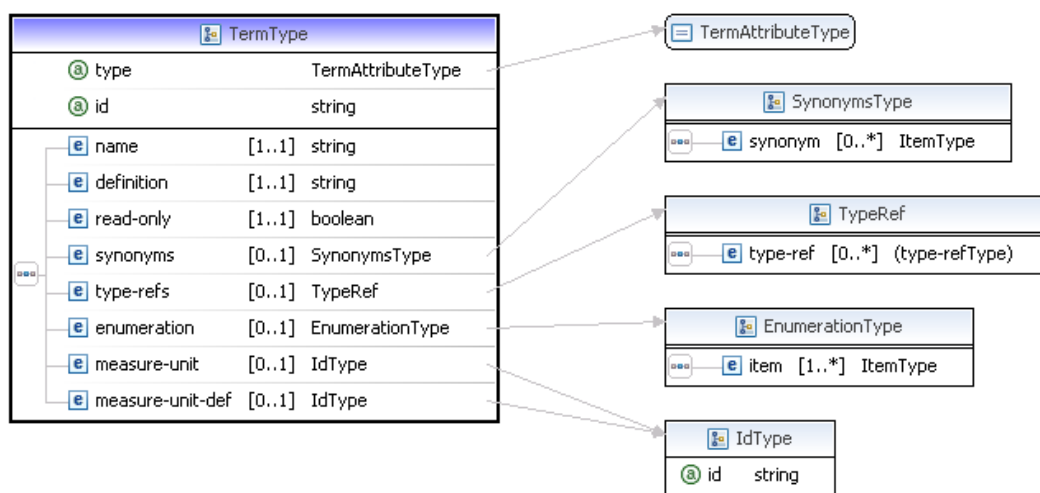
On the SEAMLESS perspective, a term is always identified by a value-definition pair, often represented respecting the format “*value :: definition*”. The rationale is that the value represents the term according to its common understanding, i.e. the word(s) identifying the concept, while the definition is a (short) sentence written in natural language defining and clarifying the concept.

This design choice was made in order to avoid possible misunderstandings due to homonymous terms. Indeed, in many languages words can assume different meanings depending on the context they are considered. By implementing the value-definition strategy, SEAMLESS provides a means for overcoming such a problem. In case of homonymous, the user editing the ontology should define as many SEAMLESS terms as the number of possible concepts that can be associated to the same word. These terms will share the same value but they will be recognisable (by the humans) because of their definition.

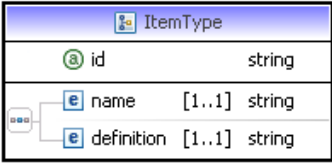
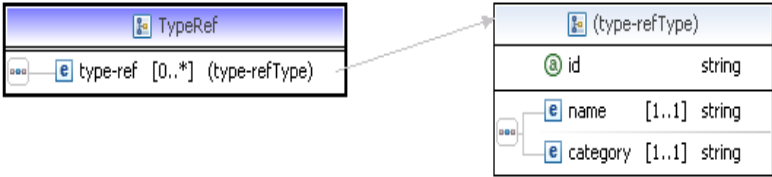
The SEAMLESS vocabulary is designed as an XML file respecting the XSD whose description follows.



On the topmost point of view, the vocabulary is collection of SEAMLESS terms (TermType XSD type)



XSD Item	Description
type (attribute)	<p>SEAMLESS vocabulary terms must belong to one of the pre-defined allowed types:</p> <ul style="list-style-type: none"> • simple term. A term which does not provide any further specialised behaviour. • numeric term. A term that can be used as numeric parameter, e.g. length, height, volume, etc. If the numeric parameter needs a measure of unit, the two relevant elements should be provided (<i>see below</i>). • enum term. A term that can be used as enumerative parameter, e.g. colour, size, etc. A list of the allowed terms representing the enumeration is supplied by the enumeration element (<i>see below</i>). • data model. A term derived by the data model definition. When the

	<p>users define a new data model type or attribute, they implicitly introduce new terms in the vocabulary. Hence, the data model terms cannot be altered within the vocabulary but only by modifying the data model.</p> <ul style="list-style-type: none"> taxonomy. A term located in the ontology taxonomy.
id (attribute)	Unique term identifier, required only when editing a COMM
name (element)	The value part of the SEAMLESS term. It can be represented by one or many words.
definition (element)	The definition part of the SEAMLESS term. A (short) sentence explaining the concept referenced by this term.
read-only (element)	The term cannot be edited (normally false, true only when COMM editing is performed and the term has been defined in the GLOB. In this case, the term can be translated).
synonyms (element)	<p>Collection of terms related to this term by a synonym relationship. A synonym is defined by this structure:</p>  <p>The diagram shows an XSD structure for 'ItemType'. It has an attribute 'id' of type 'string' and two elements: 'name' and 'definition', both of type 'string' and with cardinality '[1..1]'.</p> <ul style="list-style-type: none"> id (attribute): unique identifier of the target synonym term, required only when editing a COMM. name (element): name of the target synonym term. definition (element): definition of the target synonym term.
type-refs (element)	<p>Collection of data model types related to this term. A term can be in relationship with a data model type for one (or many) of the following reasons:</p> <ul style="list-style-type: none"> Term represents the type name Term represents one of the type's attribute name Term is involved in the domain definition of a type's attribute (enum parameter, num parameter, term, taxonomy) <p>Any of these relationship should be taken into account while managing data model and vocabulary updates. The reference type is described by the following XSD structure:</p>  <p>The diagram shows an XSD structure for 'TypeRef'. It has an element 'type-ref' of type '(type-refType)' with cardinality '[0..*]'. The 'type-ref' element is shown with a reference arrow pointing to the '(type-refType)' structure. The '(type-refType)' structure has an attribute 'id' of type 'string' and two elements: 'name' and 'category', both of type 'string' and with cardinality '[1..1]'.</p> <ul style="list-style-type: none"> id (attribute): unique identifier of the target data model type, required only when COMM editing is performed. NOTE: if the relationship is due to an attribute within the type, the id must reference the attribute id. name (element): user defined name of the referenced type.

	<ul style="list-style-type: none"> category (element): name of the category to which the type belongs (COLLABORATION, COMPANY, NEGOTIATION, PRODUCT_AND_SERVICES). 								
enumeration (element)	<p>Collection of terms representing the options related to this enumerated term type.</p> <p>The referenced type is here outlined:</p> <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 10px auto;"> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr style="background-color: #e6f2ff;"> <th colspan="2" style="text-align: left; padding: 2px;">ItemType</th> </tr> <tr> <td style="padding: 2px;">@ id</td> <td style="padding: 2px;">string</td> </tr> <tr> <td style="padding: 2px;">e name [1..1]</td> <td style="padding: 2px;">string</td> </tr> <tr> <td style="padding: 2px;">e definition [1..1]</td> <td style="padding: 2px;">string</td> </tr> </table> </div> <ul style="list-style-type: none"> id (attribute): unique identifier of the target term, required only when editing a COMM. name (element): name of the target term. definition (element): definition of the target term. 	ItemType		@ id	string	e name [1..1]	string	e definition [1..1]	string
ItemType									
@ id	string								
e name [1..1]	string								
e definition [1..1]	string								
measure-unit (element)	<p>In case of num term type, a unit of measure can be associated to this term. The unit of measure must be a vocabulary term. This element express the unit of measure term value. When COMM editing is performed, the target term identifier is required as well.</p>								
measure-unit-def (element)	<p>In case of num term type, a unit of measure can be associated to this term. The unit of measure must be a vocabulary term. This element express the unit of measure term definition. When COMM editing is performed, the target term identifier is required as well.</p>								

3.2.3 Taxonomy structure

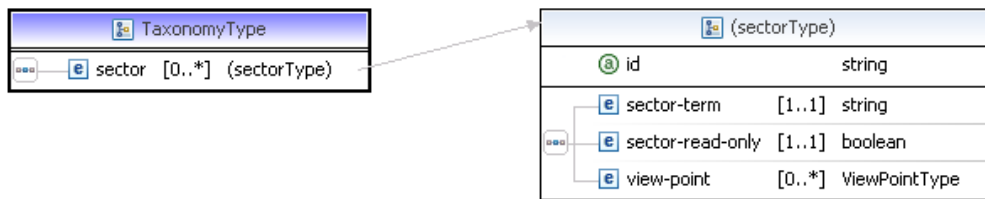
The ontology taxonomy classifies concepts (terms) into a hierarchical structure. The terms appearing in the taxonomy belong as such to the ontology vocabulary. The taxonomy organisation is founded on the following concepts:

- Sector. It represents the trade sector for which a taxonomy is defined, e.g. Building & Construction, Textile, Food. We can have “generic” ontology (representing many sectors) or single-sector ontologies.
- View point. Each sector can provide many classifications according to different view points such as processes, products, materials, uses and so on. At least one view point must be specified. Within the same ontology, different sectors are allowed to define different view points.
- Taxonomy term. A vocabulary term located in a hierarchical structure beneath a specified view point. A taxonomy term is related to only one parent (either a view point or another taxonomy term) and can be parent of many other taxonomy terms. It is allowed for a taxonomy term being used under different view points but it cannot be defined more than once under the same view point.

The SEAMLESS ontology represents the taxonomy structure by means of an XML file whose schema is depicted below.



The taxonomy is built upon sectors which can occur with unlimited cardinality.



XSD Item	Description
id (attribute)	Unique term identifier, required only when editing a COMM
sector-term (element)	A vocabulary term representing the sector name. The term must be compliant with the general term representation format, namely “value :: definition”.
sector-read-only (element)	The sector cannot be edited (normally false, true only when editing a COMM and the sector has been defined in the GLOB).
view-point (element)	<p>A sector includes set of view points according to the following schema:</p> <div style="border: 1px solid black; padding: 5px; margin: 10px 0;"> </div> <p>id (attribute): unique term identifier of the view point, required only when COMM editing is performed</p> <p>view-point-term (element): a vocabulary term representing the view-point name. The term must be compliant with the general term representation format, namely “value :: definition”.</p> <p>view-point-read-only (element): the view-point cannot be edited (normally false, true only when COMM editing is performed and the view-point has been defined in the GLOB).</p> <p>A collection of items represents the hierarchical term structure:</p> <div style="border: 1px solid black; padding: 5px; margin: 10px 0;"> </div> <p>id (attribute): unique term identifier of the item, required only when COMM editing is performed</p> <p>level (element): the taxonomy level to which this item belong to. Levels are expressed according to a dot-separated notation like “1.2.4” meaning that this level is the fourth children of the item “1.2”, and so on up to the</p>

	<p>root.</p> <p>level-read-only (element): constant (false)</p> <p>term-name (element): value of the taxonomy term located at this level of the taxonomy.</p> <p>term-definition (element): definition of the taxonomy term located at this level of the taxonomy.</p> <p>code (element): user defined code associated to this taxonomy level (empty value allowed)</p> <p>read-only (element): the item cannot be edited (normally false, true only when editing a COMM and the level has been defined in the GLOB).</p>
--	---

3.2.4 SEAMLESS ontology and OWL

As stated at the beginning of this chapter, the first assumption related to the SEAMLESS ontology data structure is the adoption of an XML-based representation. In particular, the previous sections have pointed out the detailed schema according to the specific contents (data model, vocabulary and taxonomy).

The W3C consortium released in 2004 the Web Ontology Language (OWL), an XML language designed for representing ontologies. More precisely, from the W3C site we read that: “The OWL Web Ontology Language is designed for use by applications that need to process the content of information instead of just presenting information to humans. OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF Schema (RDF-S) by providing additional vocabulary along with a formal semantics”.

Since OWL is currently considered a key element of the Semantic Web, the choice of adopting a different language for the SEAMLESS project needs a proper explanation.

The following list summarises few of the most relevant OWL features:

- OWL is a set of XML elements and attributes with standardised meaning, which are used to define terms and their relationships (ontology).
- OWL has more facilities for expressing meaning and semantics than XML and thus OWL goes beyond these languages in its ability to represent machine interpretable content on the Web.
- OWL inherits from RDF Schema the vocabulary for defining properties and classes with a semantic for generalisation-hierarchies of such properties and classes. Moreover, OWL adds more vocabulary for describing properties and classes, e.g. relations between classes, cardinality, equality, richer typing of properties, etc.
- OWL is a language conceived for defining a base of knowledge upon terms associations that can be inferred by proper query languages.

The following considerations try to point out how the OWL features impact on the SEAMLESS requirements and, consequentially, why a different strategy has been followed.

While OWL is focused on the ontology definition issue, one of the main expected outcomes of the SEAMLESS project is the possibility of converting documents compliant with a specific ontology in documents compliant with a different ontology preserving, as much as possible, the shared concepts. Therefore, there is a need to create mappings between ontologies, i.e. a link describing what are the liaisons between two different ontologies and, mostly, mechanism to convert documents.

Converting a document means performing two main activities: (i) altering the data structure of the source document to obtain a structure compliant with the target data model; (ii) find out the term associations in order to perform linguistic translations. Again, these two actions have to be carried out according to the instructions provided by the users by a means of the Mapper module.

The mapping between two OWL ontologies is definitely possible but more complex if compared to the mapping involving two XSD files or XML files:



- OWL comes with a richer semantic than XSD/XML, then the conversion is asked to address more options. This means introducing complexity both in the mapping instance definition and in the conversion process. Thus, in the first step of the project we believe that the best choice is to provide a “simple” XML mapping file³⁴, in order to better consolidate its structure and its features. After that, in a second step the same knowledge could be represented by means of the OWL language if useful.
- It must be taken into account that OWL is not a validation language like XSD. OWL has a proprietary XML schema that validates an OWL document, but OWL in itself is not a language used to validate documents, that is, it does not provide a way to verify if an XML document is conforming to the relative structure. Moreover, the XML schema (XSD) that OWL uses is oriented to define relationships among instances of OWL classes, which is different from an out-and-out data model definition.

Last but not least, even though the SEAMLESS ontology is not defined in OWL language, it could be automatically converted into a OWL instance if required for exporting purposes. Many tools, either commercial or free of charge, already provide specific facilities able to map XSD/XML file against OWL (e.g. http://www.semanticscripting.org/XML2OWL_XSLT, <http://jxml2owl.projects.semwebcentral.org/>, <http://www.topbraidcomposer.com/>, etc.).

Concerning the possibility to import OWL-based ontologies into SEAMLESS, they can be translated into the SEAMLESS language by means of ad hoc procedures and taking into account specific limitations due to the different levels of semantics provided by the two approaches. This final consideration intends to remark that the OWL language has a semantic oriented structure, integrated with logic description theories, inference and entailment concepts. As a consequence, using an OWL ontology into the SEAMLESS world implies a previous filtering and exclusion of the OWL ontology “elements”

3.3 Editor functionality

This chapter describes the design of the Editor module starting with the outline of the overall architecture and then moving to the implementation details.

3.3.1 Major design decisions

The overall design process of the Editor 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 Editor 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.
- Local user interface. The Editor module has been conceived as a locally installed application providing facilities for managing ontologies. Hence, the depicted scenario is represented by SEAMLESS users that download the Editor, install it onto their PCs / laptops and run the application to edit ontologies stored in a given Internet repository.

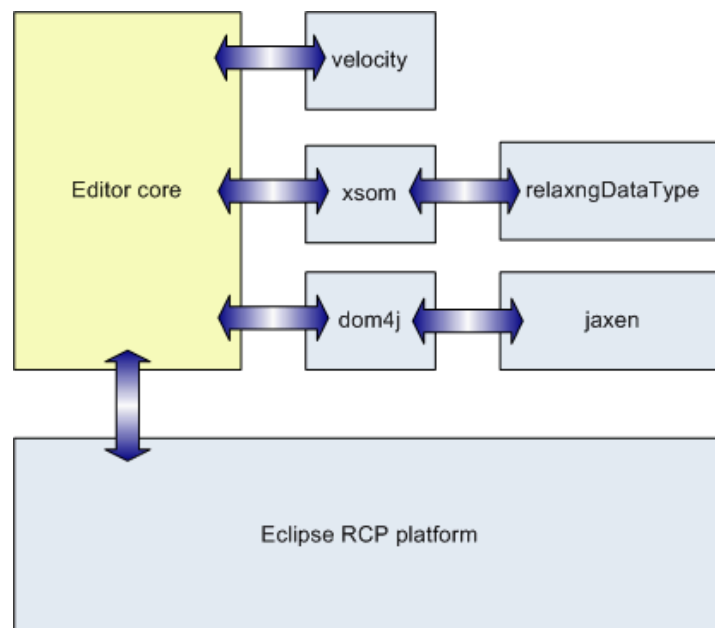
³⁴ “Simple” means that the mapping file does not include all those XML tags that OWL uses to declare the properties, relationships and class instances. In fact these tags are not requested for the description of a structural mapping. On the contrary they make complex the output mapping file and as a consequence the Semantic Translator should be modified (making more complex its structure) to cope with the OWL-dependent mapping files.



- Platform-independent solution. The adoption of Java-based technologies enforces the previous concepts and, at the same time, brings this valuable side effect. Additionally, the look and feel of the user application, the user interface responsiveness and the memory footprint are critical issues that the design process has properly addressed. In particular, the choice of the Eclipse Rich Client Platform (RCP, <http://www.eclipse.org/rcp>) as base structural framework and SWT/JFace as interface components has been considered the natural answer to the SEAMLESS requirements.

3.3.2 High level architecture

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



The **Eclipse RCP** represents the base on which the whole application relies since it offers the complete set of services that manages the user interface along with additional facilities.

The Eclipse project has been designed by IBM in order to provide an open platform for building Integrated Development Environments (IDE). The key-feature of the Eclipse project can be found in the plug-in architecture, i.e. a framework where different modules (plug-in) can be attached in order to provide their own contribution in term of facilities.

The most valuable result of this project is the Eclipse IDE, an application environment for Java development. Many plug-ins have been developed in order to add specific functions to the core IDE. The plug-in architecture relies on the concept of extension point, i.e. a specific behaviour defined by a core plug-in that can be contributed by other plug-ins respecting a predefined communication interface.

Eclipse RCP is the open-source platform on which Eclipse IDE is built without the IDE-specific plug-in. Hence, a generic application (not only IDEs) can extend the various platform extension-point in order to leverage the Eclipse facilities. In particular, the services extended by the Editor core are:

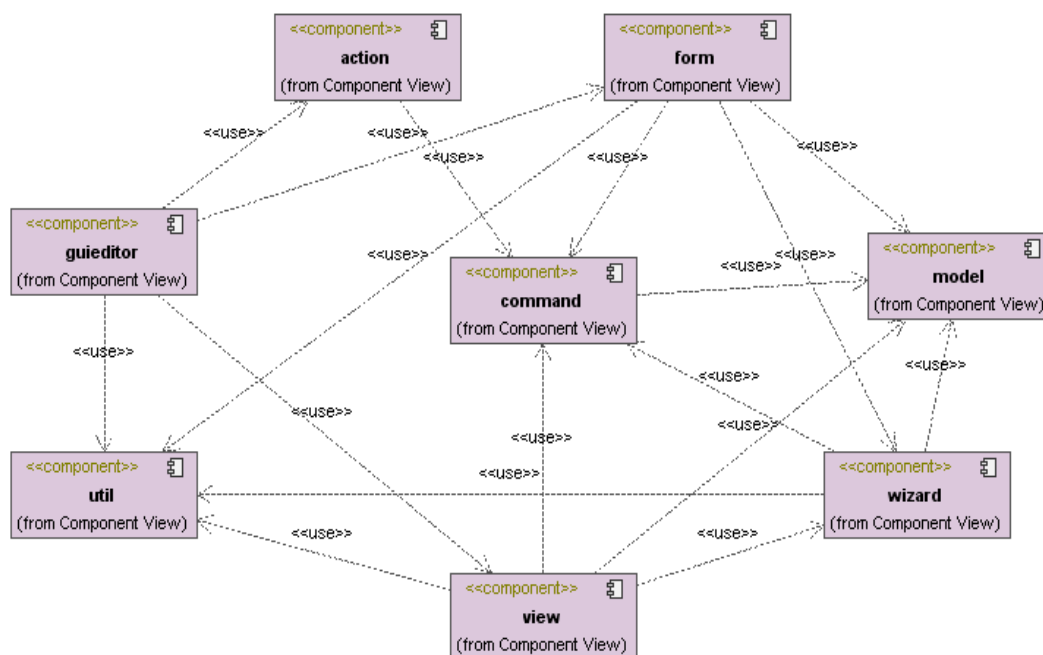
- Overall window system. It controls the overall user interface area in terms of user control event dispatching and window positioning.
- Views rendering. It manages the rendering and the layout of the view components (general-purposes areas within the main application window for displaying either tree or tables).



- Form editor rendering. It manages the rendering and the layout of the form editor components (areas within the main application window for displaying information typically for being edited).
- Action management. It provides the control on the window menus, toolbar buttons and key bindings (action triggered by a combination of key pressed).
- SWT/JFace. The library supplying the basic graphical components to be displayed in the application window (buttons, labels, text fields, etc.).
- Help service. It implements an on-line help by indexing the provided contents (html format) and rendering them within a proper area.
- Auto update. The application automatically looks for updates in predefined Web site. If updates are available, the new features are downloaded to replace the existing ones.

The **Velocity** library (<http://jakarta.apache.org/velocity>) provides services for producing contents (files) based upon textual templates. Once the template is edited, Velocity compiles it at run-time by replacing the encountered placeholders with the current Java variables specified in the Java code. The result of this operation is a template-based content produced on the fly. The Editor leverages this functionality for producing the output XML files. The template based approach enforces the readability and maintainability of the application if compared with a pure programmatic pattern.

The **Xsom** library (<https://xsom.dev.java.net>) implements a useful API for parsing XSD documents. Differently from usual XML parser, Xsom exposes classes and methods specifically oriented towards the parsing of XSD contents, e.g. types, elements, etc. The library has been adopted by the Editor module for parsing the data model file expressed in the ontology.



Generated by UModel

www.altova.com

Dom4j module (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.



The **Editor core** module has been developed in order to address the explicit SEAMLESS requirements. The component diagram realising the Editor core module is depicted in the figure of the previous page. The following sections provide a more detailed view on each component.

3.3.3 Editor core - action component

This component is implemented by classes associated to the users' action triggered by either the menu items selection or tool bar buttons. According to the RCP contract, these classes must inherit the Eclipse Action class and must be registered to the main application window (IWorkbenchWindow).

Often, Action classes do not directly implement business logic procedures but delegate them to the components belonging to the command package.

Class Summary	
DeleteTaxonomyAction	Action for deleting a view point/sector within a taxonomy
DeleteTypeAction	Action for removing an existing type from the selected data model folder
DownloadMasterOntologyAction	Action for downloading the master ontology.
ExportXSDfileAction	Action for exporting information in XSD file.
LoadOntologyAction	Loads and shows an existing ontology.
NewLevelAction	Action for adding a new level to the selected taxonomy view point/level
NewOntologyAction	Action for choosing a new ontology to be loaded.
NewSectorAction	Action for adding a new sector to the selected taxonomy folder
NewSectorViewPointAction	Action for adding a new sector view point to the selected sector folder
NewTypeAction	Action for adding a new type to the selected data model folder
OntologyShowViewAction	Action for opening the ontology view.
PropertyShowViewAction	Action for opening the property view.
SaveOntologyAction	Action for saving ontology data

3.3.4 Editor core – form component

Component collecting classes for displaying the form content. Forms are Eclipse RCP devices for containing updatable data. Often, forms include tables and tree viewers which need suitable classes (content provider, label provider, sorter) for performing the proper data visualisation.

Class Summary	
DictionaryContentProvider	Content provider for dictionary term table
DictionaryLabelProvider	Label provider for dictionary term



TaxonomyFormEditor	FormEditor representing taxonomy information
TaxonomyFormEditorInput	Editor input for TaxonomyFormEditor
TaxonomyFormEditorPage	Form page containing the taxonomy level information
TaxonomyItemContentProvider	Content provider for taxonomy item tree
TaxonomyItemLabelProvider	Label provider for taxonomy item
TaxonomyItemSorter	Sorter for taxonomy items
TermPropertiesContentProvider	Content provider for vocabulary term table
TermPropertiesLabelProvider	Label provider for vocabulary term
TypeAttributeContentProvider	Content provider for attribute table in 'TypeFormEditorPage'
TypeAttributeLabelProvider	Label provider for attribute table within the type form.
TypeAttributeSorter	Sorter for attribute table within the type form.
TypeFormEditor	FormEditor representing type information
TypeFormEditorInput	Editor input for TypeFormEditor
TypeFormEditorPage	Form page containing the table of attributes for each type
VocabularyContentProvider	Content provider for vocabulary term table
VocabularyFormEditor	FormEditor representing vocabulary information
VocabularyFormEditorInput	Editor input for VocabularyFormEditor
VocabularyFormEditorPage	Form page containing the vocabulary information
VocabularyLabelProvider	Label provider for vocabulary term
VocabularyTermSorter	Sorter for vocabulary terms

3.3.5 Editor core – model component

The model component provides the Editor core with the basic data structures on which the Editor module is built, without directly dealing with GUI issues. On the ontology data model point of view, classes defining the inner structure of types and attributes are here introduced. At the same way, the term concepts find here they location in order to represent the ontology vocabulary and taxonomy.

Anytime a previously edited ontology is loaded in the Editor module, the model is populated according to the saved data. Hence, proper classes have been designed in order to facilitate the loading process. At the same way, once the ontology is edited, the users have to store it. Again, suitable helper classes have been introduced.

Interface Summary

IFolderListener	Listener for changes occurred within a folder.
------------------------	--



ITypeListener	Listener for changes occurred within a type.
IVisible	Defines a visible objects.
IVisitor	Data model visitor interface

Class Summary	
AbstractAttribute	Model class for attribute definition
AbstractParamAttribute	Abstract base class for parameter attribute
BinaryAttribute	Attribute representing a binary object
DataModel2Xsd	Populates the data structures for creating a new XSD instance for SEAMLESS data model representation.
DataModel2Xsd.AbstractAttributeWrapper	Attribute wrapper specialised for velocity rendering.
DataModel2Xsd.TypeWrapper	Type wrapper specialised for velocity rendering.
DataModelTerm	Model object representing a term related to the data model (type or attribute name).
DateAttribute	Attribute representing a date (e.g. 1969-06-27)
DateTimeAttribute	Attribute representing a datetime (e.g. 1999-05-31T13:20:00.000-05:00)
DecimalAttribute	Attribute representing decimal
Dictionary2Xml	Converts model data to dictionary Xml representation
DictionaryItem	Represents a single item within the dictionary
DurationAttribute	Attribute representing a duration.
EnumParamAttribute	Attribute representing enum parameter
EnumParamTerm	Model object representing an enumeration term.
Folder	Model class for representing content folder
IntegerAttribute	Attribute representing an integer
NumParamAttribute	Attribute representing numeric parameter
NumParamTerm	Model object representing a numeric term.
OntologyManager	Singleton class for managing project meta-data
Taxonomy2Xml	Populates the data structures for creating a new XML instance for SEAMLESS taxonomy representation.



Taxonomy2Xml.SectorWrapper	Sector wrapper specialised for velocity rendering.
Taxonomy2Xml.ViewPointWrapper	View point wrapper specialised for velocity rendering.
TaxonomyItem	Model object representing a taxonomy item.
TaxonomyTermAttribute	Attribute representing a Taxonomy term
Term	Model object representing a simple vocabulary term
TermAttribute	Attribute representing a subset of the vocabulary terms
TermAttributeValue	Item in a term attribute value
TextAttribute	Attribute representing a text content
TimeAttribute	Attribute representing a time (e.g. 14:30:00.000-05:00)
Type	Model class for type definition
TypeAttribute	Attribute referencing an existing type
TypeEvent	Event fired by a change on a type.
Vocabulary2Xml	Converts a vocabulary model into an XML String
VocabularyHelper	Helper singleton class for providing direct access to vocabulary information.
Xsd2DataModel	Utility class for reading XSD file

3.3.6 Editor core – wizard component

Wizards are specific GUI items, typically represented as multi-page modal dialog boxes, whose role is supporting users throughout the definition and modification of concepts. Eclipse RCP provides a dedicated API built on two main classes: Wizard and WizardPage. The rationale is that a Wizard is a container for WizardPage instances. Each WizardPage instance provides graphical information about which GUI controls has to be rendered while the WizardPage instance controls the flow among pages and performs action once the Finish button is pressed.

Within the Editor module, wizard controls play the role of basic GUI means for the definition of types, attributes, vocabulary terms and taxonomy terms.

Interface Summary	
ICustomComboCellEditorProvider	Provides data for CustomComboCellEditor

Class Summary	
AttributeWizard	New attribute wizard class
AttributeWizardPage	New attribute wizard base page



CustomComboCellEditor	Cell editor interacting with the vocabulary term list.
DateAttributeWizardPage	Wizard page for date specific attribute
DateTimeAttributeWizardPage	Wizard page for date time specific attribute
DecimalAttributeWizardPage	Wizard page for decimal specific attribute
DefaultCustomComboCellEditorProvider	Provides a filter on the list of available terms.
DurationAttributeWizardPage	Wizard page for duration specific attribute
EnumParamAttributeWizardPage	Page for enumerative element creation
IntegerAttributeWizardPage	Wizard page for integer specific attribute
NewOntologyWizard	Wizard for defining a new ontology.
NumParamAttributeWizardPage	Page for numeric element creation
OntologyDirectoryPage	This class is a wizardpage.
SectorViewPointWizard	Sector view point creation and management
SectorViewPointWizardPage	Page for sector view point creation
SectorWizard	New data model class creation
SectorWizardPage	Page for sector creation
TaxonomyItemWizard	Taxonomy item wizard class
TaxonomyItemWizardPage	Wizard page for editing taxonomy item.
TaxonomyTermAttributeWizardPage	Page for taxonomy term selection from the table list creation
TermAttributeWizardPage	Page for term selection from the table list creation
TextAttributeWizardPage	Wizard page for text specific attribute
TimeAttributeWizardPage	Wizard page for time specific attribute
TypeAttributeWizardPage	Wizard page for defining an attribute referencing a data model type.
TypeWizard	New data model class creation
TypeWizardPage	Page for new data model class creation
VocabularyEnumParamTermWizardPage	Wizard page for editing enum param terms.
VocabularyNumParamTermWizardPage	Wizard page for editing num param terms.

VocabularyTermCellModifier	Manages the modification logic for vocabulary term tables.
VocabularyTermContentProvider	Content provider for table displaying a term list.
VocabularyTermLabelProvider	Label provider for table displaying a single column of term:definition data
VocabularyTermWizard	Vocabulary term wizard class
VocabularyTermWizardPage	Wizard page for editing terms.
VocabularyTranslateTermWizard	Vocabulary translate term wizard class.
VocabularyTranslateTermWizardPage	Wizard page for translating terms.

3.3.7 Editor core – view component

Views are key elements of the Eclipse RCP architecture since they represent a straightforward way for representing information within a multi-window context. Indeed, views are special windows, contained in the main workbench window, that can be resized, hidden, docked along the workbench window sides, etc., according to the users preferences.

The Editor module implements only one view in charge of delivering all the information about the current ontology. In details, the view encloses a tree representation of data model vocabulary and taxonomy concepts.

Class Summary	
DataModelLabelProvider	Label provider for the data model sub-tree.
DataModelProvider	Class for initialising the data model view.
DataModelTreeNode	Represents a node of the data model tree view.
OntologyDataView	View for ontology browsing.
OntologyDataViewLabelProvider	Label provider for ontology view

3.3.8 Editor core – util component

As the name suggests, this component provides classes implementing general purpose utility services. Hence, many components relies on this services, either for performing GUI-related actions or business logic tasks.

Interface Summary	
RunInProgress.IRunInSharedThread	Thread behaviour
RunInProgress.IRunInThread	Thread behaviour

Class Summary	
ErrorMessageLabel	Provides a widget for representing error messages within a form



ImageRegistryProxy	Registry managing images
MasterOntologyWorker	Performs pre-elaboration steps on an already downloaded master ontology.
NameValidationHelper	Helper class providing methods for performing validation against the model names
RunInProgress	Utility class for displaying run in progress dialog.

3.3.9 Editor core – guieditor component

This component contains the start-up classes required by a RCP based application. The classes provide call-back methods for interacting with the whole application lifecycle as well as the layout of views, forms and tool bar of the main workbench window.

Class Summary	
Activator	The activator class controls the plug-in life cycle
Application	This class controls all aspects of the application's execution
ApplicationActionBarAdvisor	Class for configuring the action bars of a workbench window.
ApplicationWorkbenchAdvisor	Class for configuring the workbench.
ApplicationWorkbenchWindowAdvisor	Class for configuring a workbench window.
Perspective	Generates the initial page layout and visible action set for a page.

3.3.10 Editor core – command component

This component supplies most of the business logic services, i.e. those facilities in charge of modifying the current model. In particular, a set of interfaces has been designed in order to decouple the access to the SEAMLESS storing services from the specific implementation (SRRN, file system, etc.).

Interface Summary	
IDataModelLoadingService	Provides a facility for loading data model information from a persistent storage
IDataModelStoringService	Provides a facility for storing data model information in a persistent way
IOntologyDownloadService	Service for downloading ontologies.
ITaxonomyLoadingService	Provides a facility for loading taxonomy information from a persistent storage
ITaxonomyStoringService	Provides a facility for storing taxonomy information into a persistent storage
IVocabularyLoadingService	Provides a facility for loading vocabulary information from a persistent storage



IVocabularyMappingLoadingService	Service for loading dictionary info
IVocabularyMappingStoringService	Service for storing mapping info about vocabulary terms translation
IVocabularyStoringService	Provides a facility for storing vocabulary information into a persistent storage

Class Summary	
AbstractCommand	Generic command.
AddAttributeCmd	Command for adding a new attribute to a type
AddTaxonomyLevelCmd	Command for adding a new taxonomy level
AddTypeCmd	Add a new type to the provided parent folder.
CommandObserver	Stores the command execution list.
DeleteTaxonomyCmd	Delete a taxonomy level/sector.
DeleteTypeCmd	Add a new type to the provided parent folder.
FileSystemService	Provides services for file system support
RemoveVocabularyTermCmd	Remove an existing Term from the vocabulary
UpdateDictionaryCmd	Command for storing dictionary information
UpdateSectorCmd	Add / Update a new sector to the taxonomy node.
UpdateSectorViewPointCmd	Add / Update a new sector view point to a sector node.
UpdateTaxonomyLevelCmd	Command for updating a taxonomy level
UpdateTypeCmd	Updates an existing type with information provided by another type.
UpdateVocabularyCmd	Command for storing vocabulary information

4 Ontology Mapper module

This chapter describes the SEAMLESS Ontology Mapper, the software module to support the mapping between concepts and terms of two ontologies.

This Mapper is new. It is developed within task T2.1 to fully meet the specific requirements of the SEAMLESS project. It is general enough to be used in the above-mentioned ontology management applications by adapting to the specific requirements of each of them.

The chapter provides a state-of-the-art analysis and a detailed technical specification of the transformation file data structure and the Mapper software functionality.

4.1 State-of-the-art

In order to justify the decision to develop a brand new Ontology Mapper this section presents a survey and performs an objective comparison of currently available tools. The analysis is focused on some tools that are well-known in the research community as interesting cases of ontology-related mapping, then it neglects the number of high-level commercial mappers/translators (e.g. developed by TIE, Sterling Commerce, Seeberger, Influe for application integration) that are generally conceived to support ambitious integration projects in medium-large enterprises.

In order to do that, a list of required features is preliminarily defined to measure the compliance level of such tools with the identified SEAMLESS needs. It is not an aim of this task to provide a comprehensive benchmark of all the general features that are normally considered when selecting an ICT tool. Rather we are simply verifying whether the analysed tools can meet those few requirements that are necessary to realise the foreseen solution for translating queries and documents in a collaboration environment made of small and micro companies.

INTEROPERABILITY WITH EXTERNAL SERVICES	
Mapping outcome persistence service	It is required that the Mapper can use an external storage system to store, index and retrieve the transformation files it generates.
Ontology loading service	It is required that the Mapper can use facilities for retrieving ontologies in a distributed storage system. Users must be put in condition to search for ontologies stored in these systems to see their contents and load them as source and destination ontologies in the Mapper application.
Conversion logic separated from mapping instructions	The mapping instructions provided by the Mapper module should be available for being interpreted and executed by an external translation service(see D.2.2). The translator implementation is intended to be independent from the Mapper implementation.
Editor module compliance	The Mapper has to be able to accept and work on the ontologies generated by the SEAMLESS Editor.
MAPPING FACILITIES	
Data model mapping	It is required to mapping the source data structure against the destination one by means of proper functions as simple copy, concatenation, split, etc. ³⁵ .
eBusiness oriented data model mapping	The data model represents concepts belonging to four main areas, namely, Company, Product/Service, Negotiation, and Collaboration. The mapper must facilitate the user in moving across the four areas when relating concepts to each other.

³⁵ See requirements UC12, UC16, UC17 of deliverable D3.1.



Vocabulary mapping	Functions for mapping the ontology vocabulary. This should be carried out by matching the source ontology terms against the proper destination terms ³⁶ .
Taxonomy mapping	Functions for mapping the ontology taxonomies. This should be carried out by matching the source ontology taxonomy terms against the proper destination taxonomy terms ³⁷ .
Two-way mapping facility	Possibility of producing, at the same time, the direct (from source ontology to destination ontology) and the inverse (from destination to source) transformation files. In fact a complete mapping process between two ontologies needs both mapping instances.
USER INTERFACE	
Technology independent interface semantics	The user interface hides the underlying implementation details of the ontology. In this case, the user is not asked to be aware of the rules regulating the specific implementation.
OPEN SOURCE	
Open source license	Possibility of modifying the product source code according to the project requirements.

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

MapForce 2007 is a visual data mapping. It is up to perform any combination of XML, database, text, EDI, and/or web service data. MapForce 2007 uniquely lets you develop mappings between data formats in an intuitive, visual manner, then auto-generates the stylesheets or program code required to implement your custom data integration and Web services applications server-side.

The main features of the product are enlisted below. Features described in this section refer to the Altova Map Force Enterprise edition, which corresponds to the most complete one among the entire set of available map force versions. Interesting features follow below:

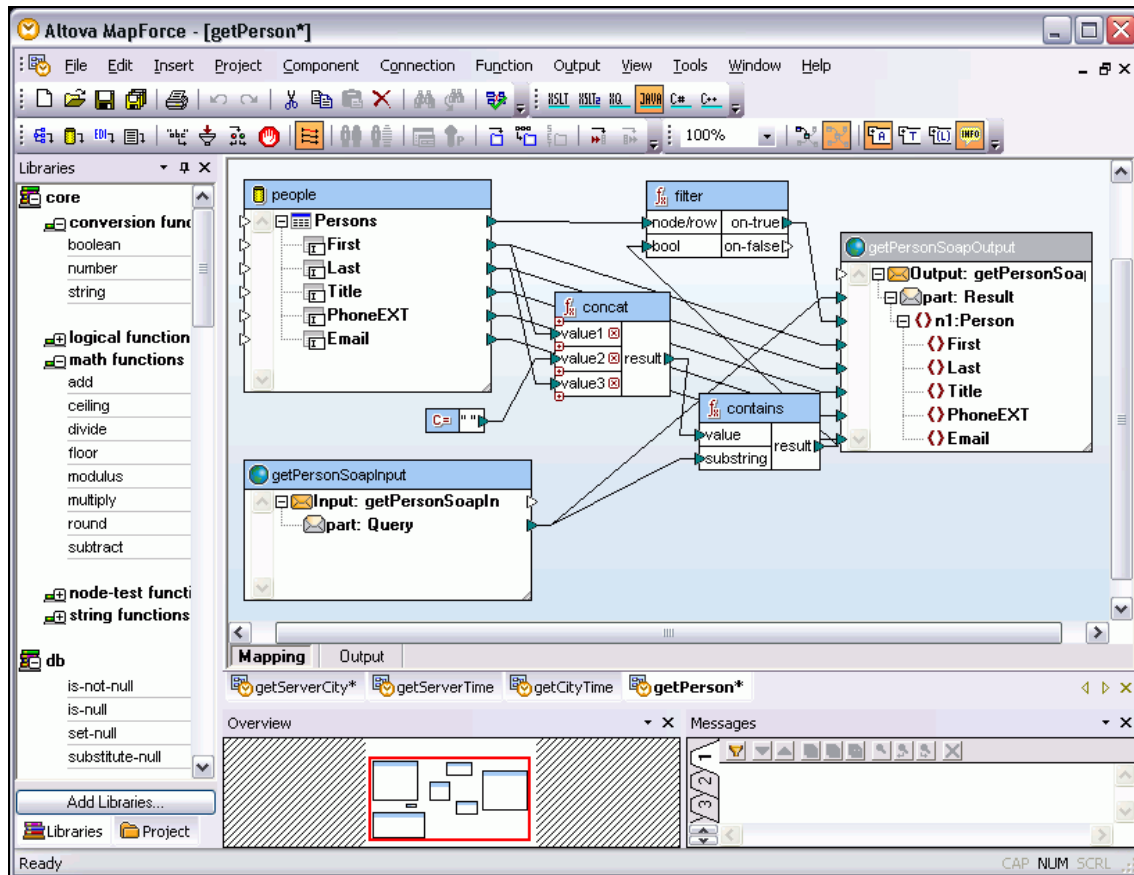
- Mapping multiple sources to multiple targets.
- Data processing output.
- XSLT 1.0/2.0 stylesheets for XML-to-XML transformations. As users are visually designing data mappings, **MapForce 2007** is generating an XSLT stylesheet for their behind the scenes. At any time, users can preview the XSLT stylesheet code by clicking on the XSLT tab at the bottom of the main design window.
- XML, Database, EDI, Flat File mapping: it consists of a bi-directional XML, database, flat file, EDI, Web services data mapping. Unique GUI for visual design of transformation. Support for advanced user-defined function, able to operate over the entire xml tree.
- The tool allows users to preview program code and output for XML / database / flat file / EDI data mappings.
- Database & legacy support: supported database include Microsoft Access, Microsoft SQL Server, MySQL, Oracle, Sybase, IBM DB2, Any ADO/ODBC database.

A typical screenshot of the application is:

³⁶ See requirements UC11, UC16, 18 of deliverable D3.1.

³⁷ See requirements UC11, UC16, UC18 of deliverable D3.1.





4.1.2 Delta 4.0 (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.

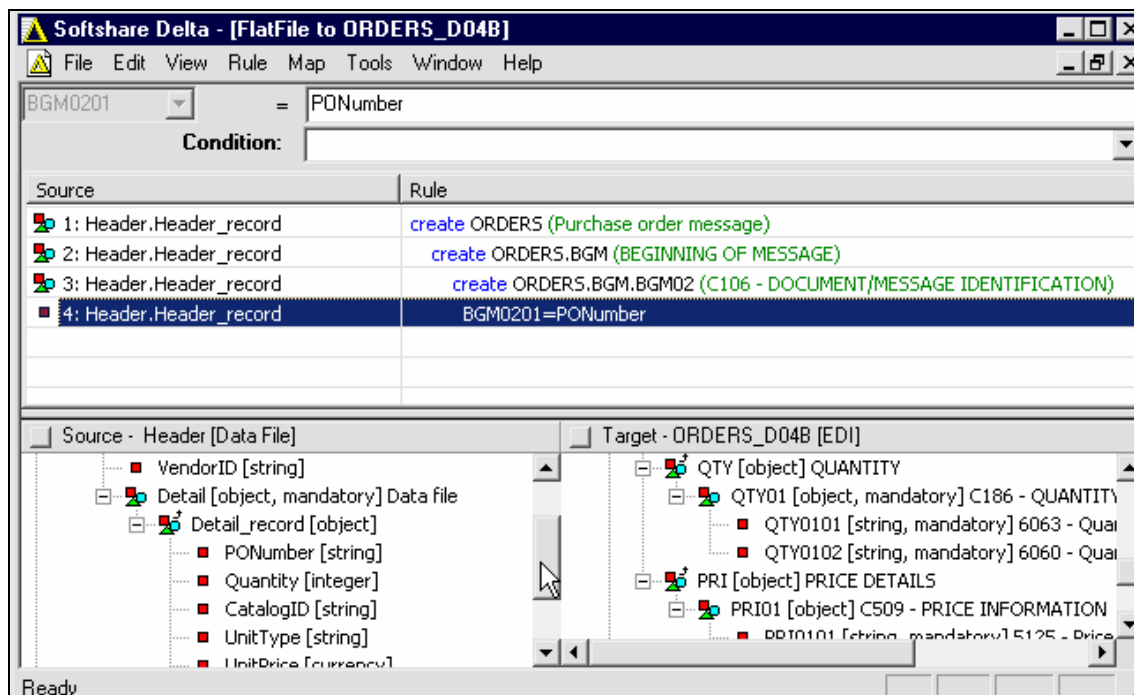
No back-end application should be an island. Using Delta, you can derive additional benefit from back-end systems such as SAP, Oracle, JDE, Epicor, the Sage MAS products, and many others by integrating disparate data sources into one powerful e-commerce enterprise.

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:



4.1.3 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 many 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're 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.

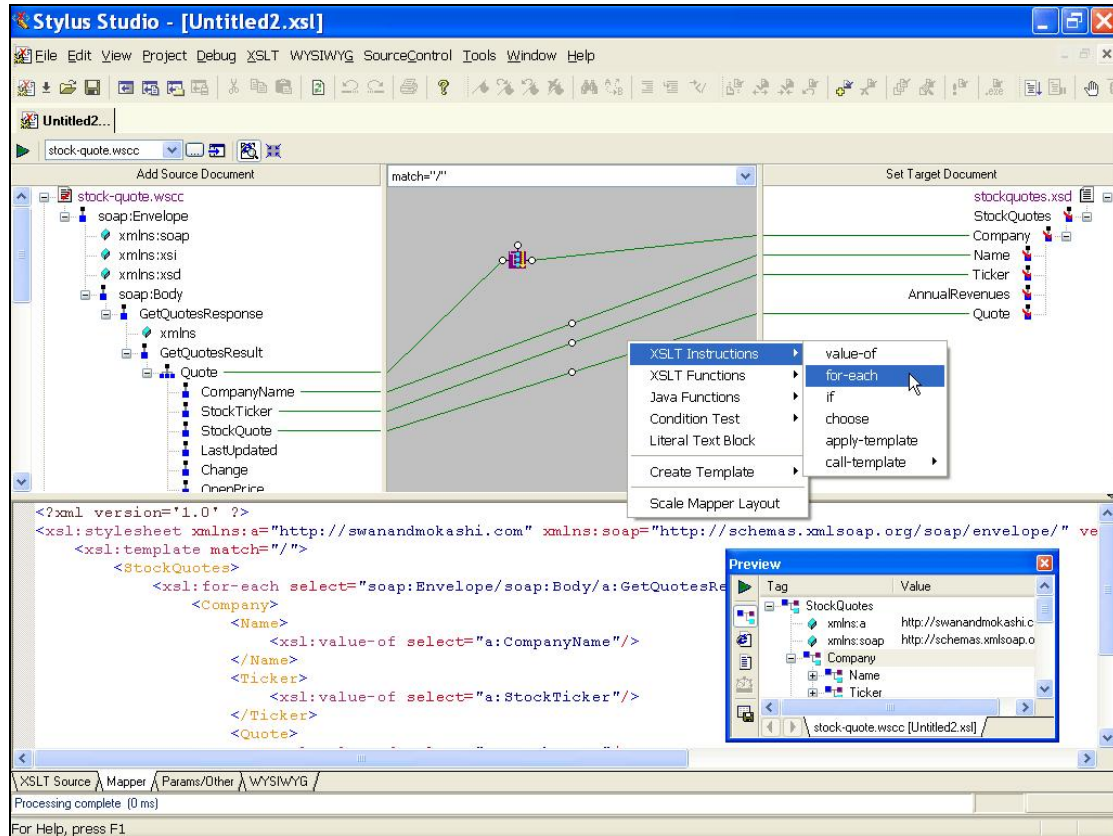
The main features of the product are enlisted below:

- 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:



4.1.4 Mapper comparison

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

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

Features	MapForce 2007	Delta 4.0	Stylus Studio
INTEROPERABILITY WITH EXTERNAL SERVICES			
Mapping outcome persistence service	YES*	YES*	YES*
Ontology loading service	YES*	YES*	YES*
Conversion logic separated from mapping instructions	NO	NO	NO
Editor module compliance	NO	NO	NO

MAPPING FACILITIES			
Data model mapping	YES	YES	YES
eBusiness oriented data model mapping	NO	NO	NO
Vocabulary mapping	NO	NO	NO
Taxonomy mapping	NO	NO	NO
Two-way mapping facilities	YES	NO	NO
USER INTERFACE			
Technology independent interface semantic	YES	YES	YES
OPEN SOURCE			
Open source license	NO	NO	NO

The symbol YES* means that these software products can interoperate with external services (proprietary or private) but not with the SRRN distributed storage system.

In conclusion of this analysis we can see that no of these tools provides the mapping 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 D2.1.1.

4.2 Transformation file data structure

The Mapper supports the creation and maintenance of a data structure describing how a SEAMLESS ontology can be mapped against another SEAMLESS ontology.

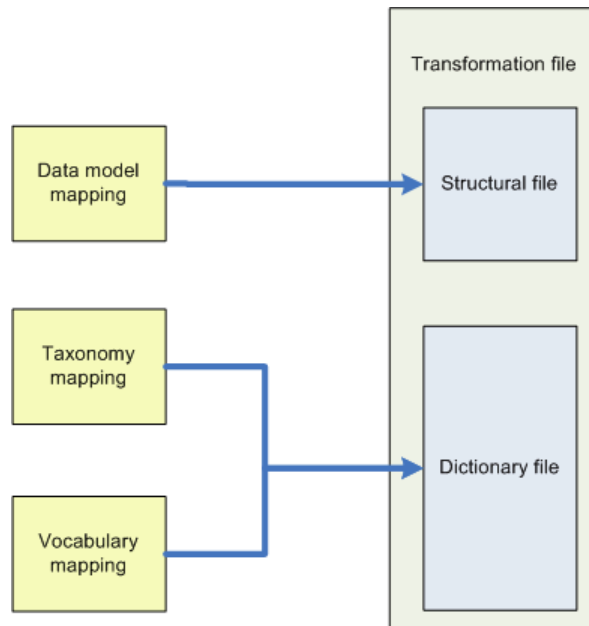
From the operational point of view, mapping two different ontologies means the following:

- **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 similar 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 between terms from both source and destination taxonomies.
- **Vocabulary mapping.** Similarly to taxonomy mapping, correspondences between terms should be defined for the vocabulary terms.

As mentioned about the Editor, a file-based persistent strategy has been considered the most appropriate in the short term, and looking forwards to interface the SRRN storage system, for storing the mapping output as well. In particular, XML has been considered the most suitable format.

Let us now go through the details of how the outcomes of previously introduced actions are persisted.



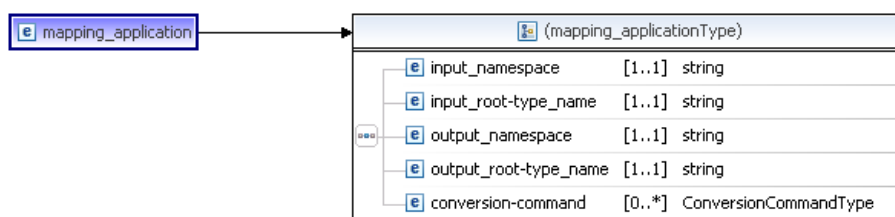


4.2.1 Structural file

The data model transformation instructions have to be stored in a proper XML file, the **structural file**. The design of this data structure has been inspired by the following key points:

- The document transformation, in terms of lingual translation and data structure conversion, is carried out by a specialised module named SEAMLESS Translator, detailed in deliverable D2.2.2.
- The transformation file is in charge of providing the Translator module with the instructions, parameters and options for performing the transformation.
- The Mapper module is not concerned on how the transformation instructions are executed, delegating the implementation details to the Translator module.

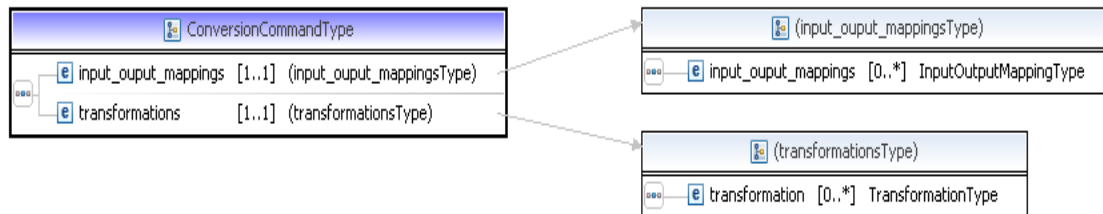
The detailed structure of the structural file follows:

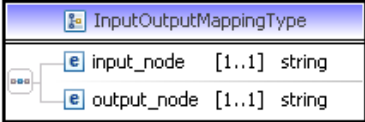
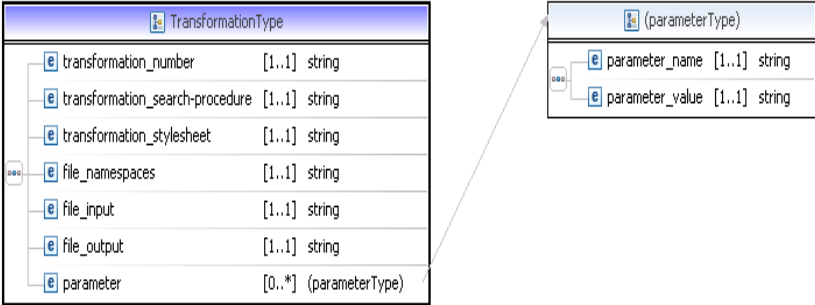


XSD Item	Description
input_namespace (element)	The single namespace declared in the input document
input_root-type_name (element)	The root type name, of the source document, to which the transformation is applicable
output_namespace (element)	The single namespace declared for the output document

output_root-type_name (element)	The root type name of the destination document
conversion-command (element)	List of commands that the converter executes at translation time.

Each conversion command follows the following organisation:



XSD Item	Description
input_output_mappings (element)	<p>List of data respecting the schema below:</p>  <ul style="list-style-type: none"> input_node (element): a valid XPATH expression pointing to a node of the source document. The node must exist after applying the transformations defined in this conversion command. output_node (element): a valid XPATH expression pointing to a node of the destination document. <p>The rationale here is that the converter module, at translation time, for each conversion command firstly applies the transformations and secondly copies the content of the input_node beneath the output_node. (See D2.2.2 for more details on the converter implementation).</p>
transformations (element)	<p>List of transformations to be applied within this conversion command:</p>  <ul style="list-style-type: none"> transformation_number (element): identifier of the transformation, just for logging purposes. transformation_search-procedure (element): identifies which searching algorithm the converter has to apply while executing this transformation. Allowed values: NONE, search, searchCommonPath, searchOccurrence (See D2.2.2 for more details on the converter implementation).

	<ul style="list-style-type: none"> • transformation_stylesheet (element): the name of the XSLT file that the converter employs for carrying out this transformation. The path should be led by the “{stylesheet_url}” string in order to make the converter point to the proper directory. • file_namespaces (element): constant value (“complete_namespace_url”). • file_input: always empty (the converter is in charge of filling this text) • file_output: always empty (the converter is in charge of filling this text) • parameter (element): set of name-value pairs providing transformation-specific parameters to the converter.
--	--

Note that each command in the transformation file includes ‘n’ input-output pairs and ‘n’ transformations. In fact we expect that the Translator executes first those transformations on source document nodes that are passed as parameters (and not on the input paths listed in the input-output pairs). The input paths simply represent the ‘result’ of the execution (something like a buffer) that must be copied into the respective output nodes. Moreover, the input-output pairs can be more than one for each command because of the nature of the transformation. E.g. for the ‘split operation the input path represents a single piece of the split value and the output path represents its destination.

Note also that in the present version of the Mapper the transformation operation field is always specified to be a stylesheet, then the transformation file is not independent of the conversion logic. This was just to obtain an early proof of concept of the editing/mapping/translation triad. An improved version is already under development to overcome this limitation.

4.2.2 Dictionary file

The Dictionary file is in charge of providing the information requested by the processes of lingual translation. Lingual translation is applied whenever a document has to be translated between two ontologies with different vocabularies.

The dictionary information can be produced according to the following situations:

Actor	Source ontology	Destination ontology	Component	Enabled translation
GLOB (Gx) Mgt application	Gx	Gy	Mapper	Gx → Gy
COMM (Cx) Mgt application	Gx	Cx	Editor	Gx → Cx, Cx → Gx
LOCL (Lx) Mgt application	Cx (COMM)	Lx	Mapper	Cx → Lx
LOCL (Lx) Mgt application	Lx	Cx (COMM)	Mapper	Lx → Cx

The table above points out which actor is in charge to create a certain dictionary and, mostly, which component is adopted according to the involved ontologies.

When using the Editor of the COMM Mgt application, the first action to perform is downloading the reference GLOB then the COMM is produced by selecting and translating the GLOB concepts. At the download time, the Editor automatically generates the dictionary file where no term is translated. As long as the COMM Mgt translate terms, the dictionary is updated together with the other ontology files.

When using the Mapper in the GLOB or LOCL Mgt applications, the vocabulary translation process is integrated with the data model mapping process (see Chapter 2 for more details), and the dictionary is generated together with the other components of the transformation file.

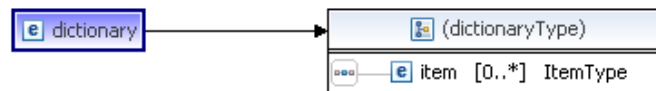
Taxonomy and vocabulary mapping relies on the structure outlined below. This means that all the mapping actions performed by the users concerning either the taxonomy mapping or the vocabulary



mapping are collected in a single dictionary file. The only relevant difference can be found in how the term are represented within the file.

While the vocabulary mapping follows the standard term representation format “value :: definition”, the taxonomy term is represented by the full path derived by its location within the taxonomy. The path is obtained by concatenating the sector name and the view point name, the ancestors of the taxonomy term and, finally, the term self. The concatenation string is “!!!”. Then, a typical term derived from a taxonomy mapping should follow this pattern: “sector value :: textile !!! view point value :: materials !!! ancestor1 value :: animal material !!! ... term value :: pure wool”.

The SEAMLESS dictionary, regardless of how it is edited, is implemented as an XML file whose schema is represented below:



The dictionary is a collection of items. Each item is represented in the following way:



XSD Item	Description
key (element)	A vocabulary term of the source ontology, represented by the term standard format “value :: definition”. The key is unique within this dictionary
id (attribute of key element)	Identifies the key vocabulary term. In case of dictionary file produced as Editor outcome, this id is used to retrieve the original (GLOB) term after a translation has been carried out.
value (element)	A vocabulary term of the destination ontology, represented by the term standard format “value :: definition”. The value term represents the translation of key term.

The key-value structure ensures that a term belonging to the source vocabulary is unambiguously translated into a destination vocabulary term (if the translation is available). It must be stressed that the same dictionary could be used for performing the inverse translation, i.e. from destination vocabulary term to source vocabulary term. However, in this situation, possible ambiguities could occur since there is no guarantee that the same value is associated to a single key.

4.3 Mapper functionality

This chapter presents the design of the Mapper starting with the overall architecture followed by the implementation details.

4.3.1 Major design decisions

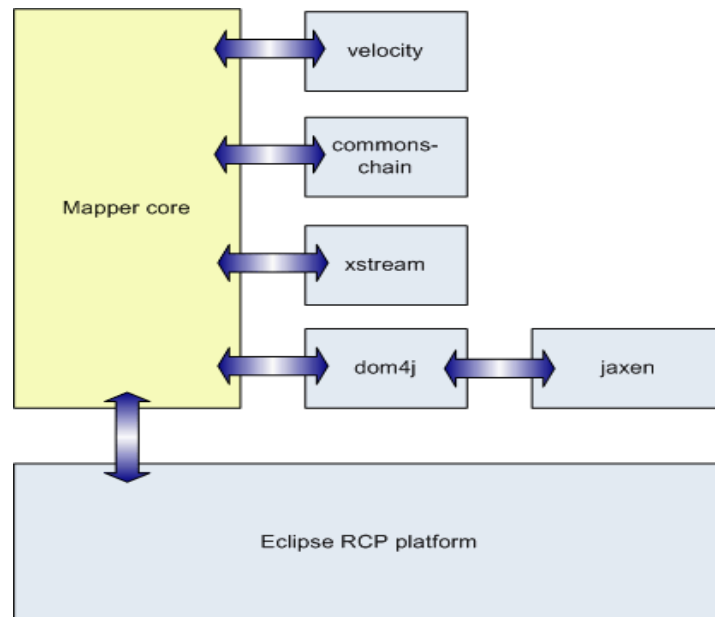
The overall design process of the Mapper has been inspired by the same guidelines already mentioned for the Editor, namely open source components, component-based approach, leveraging on well-documented and stable third party components in order to exploit their services, interoperability, local user interface, and platform-independent solution.



In addition, the user interface should be designed by taking into account that SEAMLESS is conceived as a system for small organisation and, therefore, for users that could not be familiar with specific ICT concepts like XML standards. In other words, The Mapper module should rely on simple, high level primitives simply understandable and accessible to non expert ICT users.

4.3.2 High level architecture

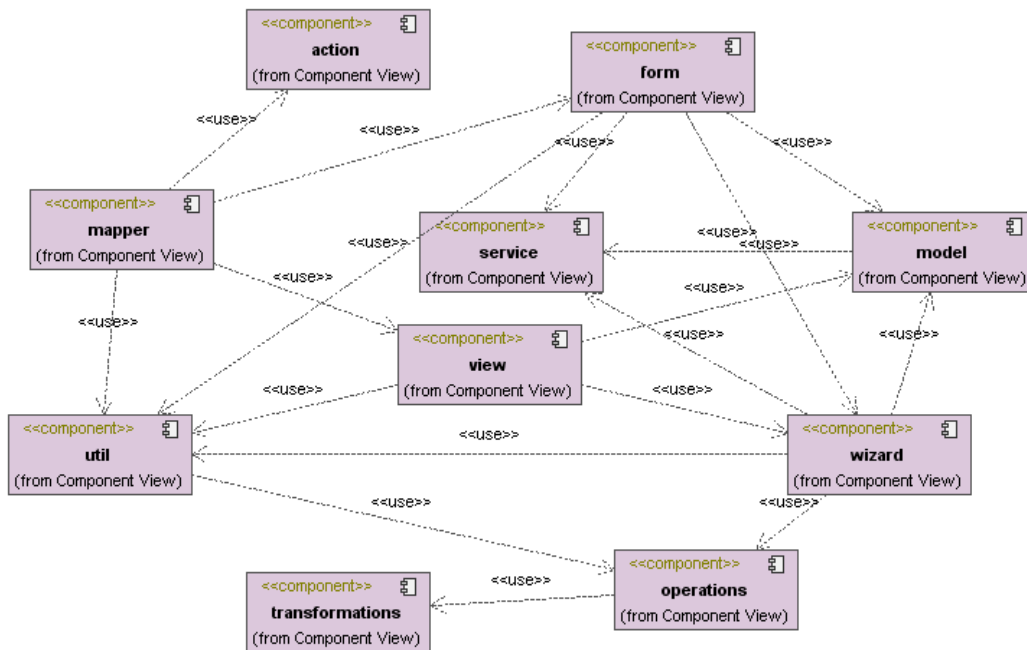
The following figure introduces (in the analogy with the Editor) the high level architecture of the Mapper module, where the yellow block represents the Mapper core, the light blue the external libraries and the blue arrows the communication among different modules:



The meaning of the represented parts, namely Eclipse RPC, Velocity, and Dom4j is given in section 3.3.2 of this document (page 53). Additional parts are Commons-chain and Xstream.

The **Commons-chain** library (<http://jakarta.apache.org/commons/chain>) implements a basic API for performing a list of operations in a predefined sequence according to an object-oriented pattern. This API has been introduced mostly to address form and wizard validation issues by adopting reusable components.

The **Xstream** component (<http://xstream.codehaus.org>) provides services for serialising Java objects to XML format as well as retrieving Java instances from XML. Since Xstream works at run-time, it has been considered the proper solution for storing and loading Mapper model information on a persistent support.



Generated by UModel

www.altova.com

The **Mapper core** module has been developed in order to address the explicit SEAMLESS requirements. The component diagram realising the Mapper core module is here depicted:

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

4.3.3 Mapper core - action component

This component is implemented by classes associated to the users' action triggered by either the menu items selection or tool bar buttons. According to the RCP contract, these classes must inherit the Eclipse Action class and must be registered to the main application window (IWorkbenchWindow).

Often, Action classes do not directly implement business logic procedures but delegate them to the components belonging to the command package.

Class Summary	
AddOperationAction	Action for adding a new structural mapping operation.
CancelUpdateAction	Action for closing a structural operation modifying section.
ChangeOperationDataAction	Action for changing parameters of a structural mapping operation.
ChangeOperationNodesAction	Action for changing nodes and parameters of a structural mapping operation.
DeleteOperationAction	Action for deleting a structural mapping operation.
ExpandSubTreeAction	Action for expanding the sub-tree of structural data-model, having the selected node as root.
OpenMapperFormEditorAction	Action for opening the mapper form editor.



SaveAction	Action for saving mapping data
ShowHistoryViewAction	Action for opening the operations history view.
UpdateAction	Action for performing product update.

4.3.4 Mapper core – form component

Component collecting classes for displaying the form content. Forms are Eclipse RCP devices for containing updatable data. Often, forms include tables and tree viewers which need suitable classes (content provider, label provider, sorter) for performing the proper data visualisation.

The Mapper application is designed as a single form window arranged in four different pages: overview (general settings and mapping configuration), structural (data model mapping), taxonomy (taxonomy mapping) and vocabulary (vocabulary mapping). Each page is responsible of providing and managing its specific content. On the other hand, the form class is in charge of providing the saving capabilities.

Class Summary	
DestinationTreeListener	Implements the checked behaviour upon a tree selection.
MapperFormEditor	Form representing the main mapper view
MapperFormEditorInput	Input class for MapperFormEditor
MapperOverviewFormEditorPage	Page representing preliminary information on mapper
MapperStructuralFormEditorPage	Page representing structural mapping
MapperTaxonomyFormEditorPage	Page representing taxonomy mapping.
MapperVocabularyFormEditorPage	Editor page collecting information for vocabulary mapping.
SelectionProviderIntermediate	IPostSelectionProvider implementation that delegates to another ISelectionProvider or IPostSelectionProvider.
SelectionTableContentProvider	Info table content provider
SelectionTableLabelProvider	Label provider for info table
SourceTreeListener	Implements the checked behaviour upon a tree selection
TreeContentProvider	Content provider for structural data-model tree.
TreeLabelProvider	Label provider for structural data-model tree.
TreeSorter	Sorter for structural data-model tree.
VocabularyTermContentProvider	Content provider for Vocabulary Term table.
VocabularyTermLabelProvider	Label provider for Vocabulary Term table.
VocabularyTermSorter	Sorter for vocabulary terms



4.3.5 Mapper core – model component

The model component provides the Mapper core with the basic data structures on which the Editor module is built, without directly dealing with GUI issues. On the ontology data model point of view, classes defining the inner structure of types and attributes are here introduced. At the same way, the term concepts find here they location in order to represent the ontology vocabulary and taxonomy.

Once a previously edited ontology is loaded in the Editor module, the model is populated according to the saved data. Hence, classes have been designed in order to facilitate the loading process. In the same way, once the ontology is edited, the users have to store it. Again, suitable helper classes have been introduced.

Interface Summary	
IOperation	Interface for high-level mapping operations between source nodes and destination nodes
IStructuralModelListener	Listener for events fired by the structural model.
ITaxonomyModelListener	Listener for events fired by the taxonomy model
IVelocityTransformation	Transformation supporting a Velocity-based layout mechanism

Class Summary	
ConversionCommand	A collection of basic transformations
IOMapping	Mapping information between an input node and an output node
StoreMappingFile	Stores a new transformation file
StoreMappingFile.CommandConversionWrapper	Wrapper for a single command conversion, used during transformation file saving operation.
StructuralMappingModel	Model class representing the structural mapping information.
StructuralModelEvent	Event for structural data model.
TaxonomyFolder	Represents a node of a tree-like taxonomy representation
TaxonomyMappingItem	Item representing a mapping between two taxonomy terms.
TaxonomyMappingModel	Model class representing the taxonomy mapping information.
TaxonomyModelEvent	Event for taxonomy model.
TaxonomyTerm	Term associated to a path representing the taxonomy hierarchy
Term	A vocabulary term



VocabularyMappingItem	Item representing a mapping between two terms.
VocabularyMappingModel	Model class representing the vocabulary mapping information.

4.3.6 Mapper core – wizard component

Wizards are specific GUI items, typically represented as multi-page modal dialog boxes, whose role is supporting users throughout the definition and modification of concepts. Eclipse RCP provides a dedicated API built on two main classes: Wizard and WizardPage. The rationale is that a Wizard is a container for WizardPage instances. Each WizardPage instance provides graphical information about which GUI controls has to be rendered while the WizardPage instance controls the flow among pages and performs action once the Finish button is pressed.

Within the Mapper module, wizard controls play the role of basic GUI means for the definition of parameters associated to the structural mapping. Since wizards are populated by complex component as editable tables, additional helper classes are here included thus providing the enabling logic for such components.

Interface Summary	
ICustomComboCellEditorProvider	Provides data for CustomComboCellEditor

Class Summary	
AbstractWizard	Abstract wizard.
AbstractWizardPage	Abstract wizard page.
ConcatSplitParamModel	Model for concatenation or split parameters.
ConcatWizardPage	Wizard page for editing concatenation parameters.
CustomComboCellEditor	Custom cell editor containing a combo box.
DefaultCustomComboCellEditorProvider	Provides a filter on the list of available terms.
DefaultWizardDialog	Default implementation of the WizardDialog
OperationSelectionWizardPage	Wizard page for operation selection.
OperationWizard	Wizard for creating or editing structural mapping operations.
SplitWithDelimitersWizardPage	Wizard page for editing split (with delimiters) parameters.
SplitWithFieldLengthsWizardPage	Wizard page for editing split (with field lengths) parameters.
TranslateVocabularyTermWizard	Wizard for translating vocabulary pages
TranslateVocabularyTermWizardPage	Page for TranslateVocabularyTermWizard
TrunksOrderCellModifier	Cell modifier for concatenation or split parameters table.



TrunksOrderContentProvider	Content provider for concatenation or split parameters table.
TrunksOrderLabelProvider	Label provider for concatenation or split parameters table.

4.3.7 Mapper core – view component

Views are key elements of the Eclipse RCP architecture since they represent a straightforward way for representing information within a multi-window context. Indeed, views are special windows, contained in the main workbench window, that can be resized, hidden, docked along the workbench window sides, etc., according to the users preferences.

The Mapper module implements only one view, the `HistoryView`, in charge of displaying the already performed mapping operations. Since the main application window is arranged in four different pages, the view content has to reproduce the same behaviour in an automatic fashion. In other words, the view has to automatically adapt its content according to the currently selected form page. Hence, the need of defining pages in this component as well.

Class Summary	
EmptyHistoryPage	Default history empty page
HistoryView	History View representing form-related information
StructuralHistoryContentProvider	Structural view content provider
StructuralHistoryLabelProvider	Label provider for structural history table
StructuralHistoryPage	History view page for structural data
TaxonomyHistoryContentProvider	Taxonomy view content provider
TaxonomyHistoryLabelProvider	Label provider for taxonomy history table
TaxonomyHistoryPage	History view page for taxonomy data

4.3.8 Mapper core – util component

As the name suggests, this component provides classes implementing general purpose utility services. Hence, many components relies on this services, either for performing GUI-related actions or business logic tasks.

Interface Summary	
RunInProgress.IRunInThread	Thread behaviour

Class Summary	
CustomLogger	Utility class for logging functions.
ErrorMessageLabel	Provides a widget for representing error messages within a form.
ImageRegistryProxy	Registry managing images.



OperationEnablingManager	Utility class for managing operations enabling logic.
RunInProgress	Utility class for displaying run in progress dialog.
UIManager	Helper class for addressing shared concerns related to the UI.

4.3.9 Mapper core – mapper component

This component contains the start-up classes required by a RCP based application. The classes provides call-back methods for interacting with the whole application lifecycle as well as the layout of views, forms and tool bar of the main workbench window.

Class Summary	
Activator	The activator class controls the plug-in life cycle
Application	This class controls all aspects of the application's execution
ApplicationActionBarAdvisor	Class for configuring the action bars of a workbench window.
ApplicationWorkbenchAdvisor	Class for configuring the workbench.
ApplicationWorkbenchWindowAdvisor	Class for configuring a workbench window.
Perspective	Generates the initial page layout and visible action set for a page.

4.3.10 Mapper core – service component

This component supplies services for enabling the communication with external modules. In particular, a set of interfaces has been designed in order to decouple the access to the SEAMLESS storing services from the specific implementation (SRRN, file system, etc).

Interface Summary	
IFileSystemNames	Provides constants for filesystem access control.
IMappingStoreService	Store mapping info information.
IOntologyLoadService	Load ontologies information.

Class Summary	
FileSystemMappingStoreService	Stores mapping information within the filesystem
FileSystemOntologyLoadService	Filesystem implementation of the ontology load service interface.
OntologyServiceFactory	Factory for accessing ontology services.

4.3.11 Mapper core – operations component

This component defines the set of operations that the users are allowed to perform throughout the structural mapping process. Each operation provides suitable methods for reading/writing parameters,



checking whether the operation is enabled according to the current node selections, information on how store the operation into the transformation file.

Class Summary	
Concatenation	Concatenates the value of n attributes into a text attribute. Parameters: a: Concatenation string b: Concatenation order
Copy	Basic copy operation.
Division	Performs a division operation between the values of selected source attributes (including a new constant operand), in the order provided by the user.
MappingOperation	Abstract class representing a structural mapping operation.
Multiplication	class that allows the multiplication of source fields each other or with a constant value
SplitWithDelimiters	class allowing the splitting of a text into one or more parts, with the use of separator element.
SplitWithFieldLengths	Class allowing the splitting of a text into one or more parts, based on field lengths.
Subtraction	class allowing the subtraction operation among fields or among fields and constants
Sum	class allowing sum operation among fields or among fields and constants

4.3.12 Mapper core – transformations component

These classes supply the logic for actually rendering the transformation file. Indeed, each operation is implemented by one or many transformations executed at conversion time. The logic for storing an operation into the transformation file invokes all the suitable transformations thus producing the actual mapping instance.

The distinction between operation (users point of view) and transformation (converter point of view) provides a useful decoupling point between the user perspective of the structural mapping and the implementation of the conversion rules.

Class Summary	
AddElementTransformation	Transformation for adding a xml element.
ArithmeticOperation2to1Transformation	Transformation for performing an arithmetic operation between two xml elements.
CopyElementTransformation	Transformation for copying an xml element.
Fusion2To1WithSeparatorTransformation	Transformation for performing a fusion with separator between two xml elements.



OccurrencesFusionWithArithmeticOperationTransformation	Transformation for performing a fusion with arithmetic operation among the occurrences of a xml element.
OccurrencesFusionWithSeparatorTransformation	Transformation for performing a fusion with separator among the occurrences of a xml element.
RemoveLastSeparatorTransformation	Transformation for deleting last separator from a xml element created by performing a fusion with separator transformation.
RenameElementTransformation	Transformation for renaming xml element.
SingleOperandArithmeticOperationTransformation	Transformation for performing an arithmetic operation between a xml element and a costant operand.
SplitWithNFieldLengthsTransformation	Transformation for performing a split of a xml element based on field lengths.
SplitWithNSeparatorsTransformation	Transformation for performing a split of a xml element based on delimiters.

5 Final remarks

This document reports the technical specification of the ontology Editor and ontology Mapper modules that have been designed and developed to meet the specific requirements of the SEAMLESS project. An analysis of similar existing tools is documented which justifies the decision to develop two brand new software tools.

The state of play at the end of task T2.1 is here summarised:

- Ontology Editor and Mapper have been prototyped, alpha-tested and are now delivered together with this document. They will undergo a stronger testing and validation process once they will be used to edit and map the ontologies that will be developed and used in the frame of the project. Following this on-field validation phase, further requirements might be identified that will result in releasing new versions of the two software modules.
- GLOB, COMM, LOCL Management applications. They are clearly defined and their main components, namely the Editor and the Mapper, now delivered in a form that is perfectly adaptable to the three different applications. These applications will be developed in the third semester (January to June 2007) as part of the mediator platform (WP4).

Editor and Mapper are now delivered so that:

- Ontology experts in the SEAMLESS consortium can start developing one or more reference global ontologies (likely, one generic GLOB and two sector-specific GLOBs) to be used in the rest of the project.
- Mediator partners in the SEAMLESS consortium have got a description of the applications they will have available at the SEAMLESS node platforms to use and work on local and common ontologies.
- Technical partners in the SEAMLESS consortium have a sound basis for the design and development of the other foreseen services and applications (on either the user or the storage system sides).

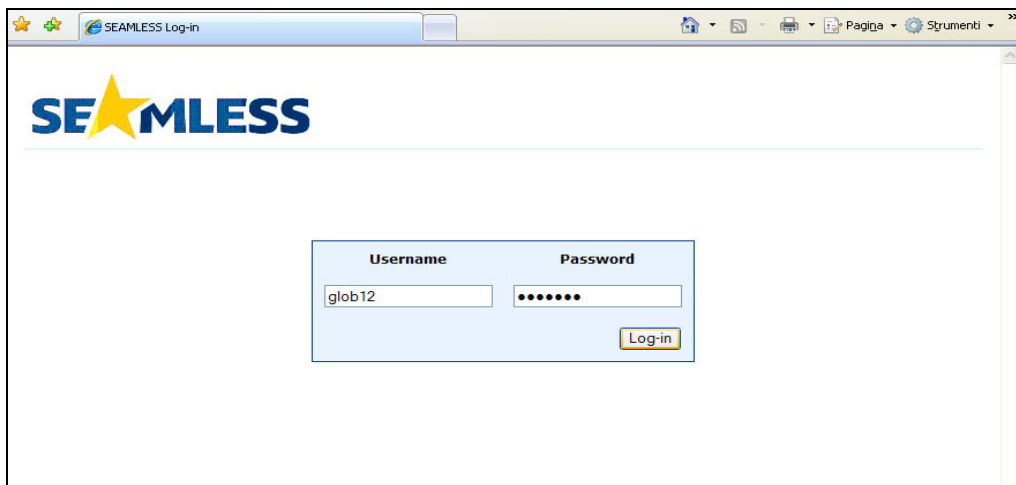
In conclusion, task T2.1 has been performed according to the plan and delivers the intended software and documents without relevant changes or delay.



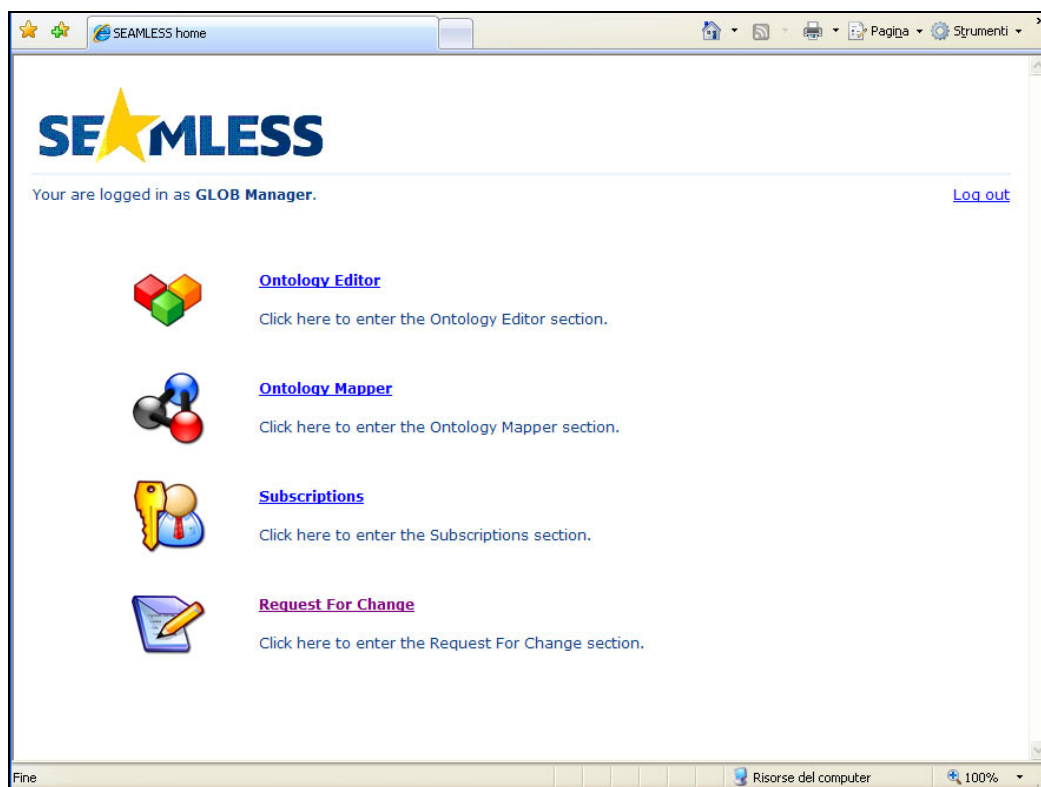
Appendix A: Ontology Mgt user interface

The GLOB, COMM and LOCL Management applications present user interfaces with many common parts and few differences. They are conceived as web applications where users log in by means of username and password and can access to the functions for which they are enabled.

Here the application user interface is simply sketched by means of few screenshots just to provide an overall idea of what could be found, while detailed presentations of the Editor and Mapper user interfaces are given in the following two appendices.

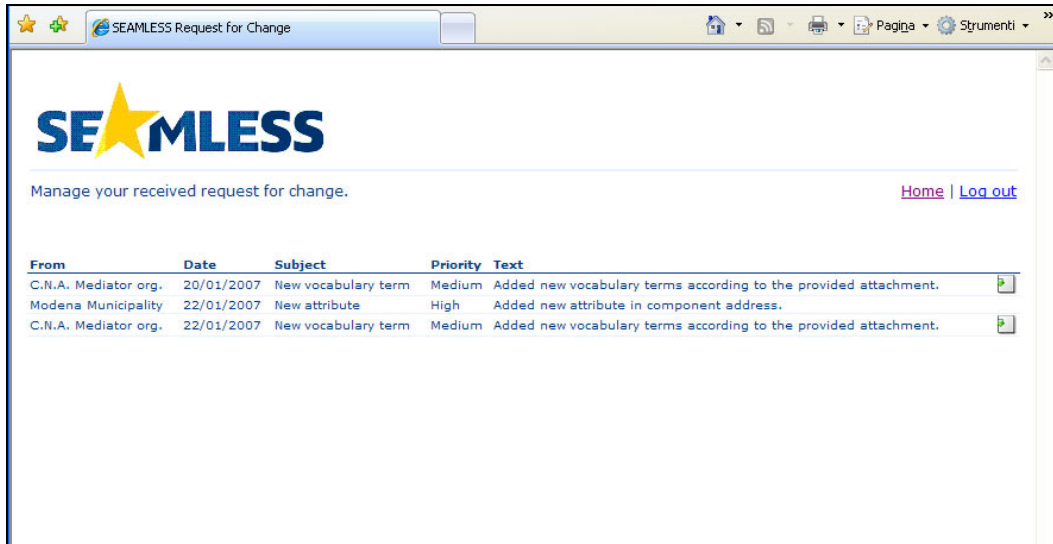


The first picture represents a standard login form by which the SEAMLESS users access the application.



The application home page represents the starting point by which users access to the enabled services and application. In this case, it is supposed that the logged user is a GLOB Manager then it can access both to the Ontology Editor and to the Ontology Mapper application.

According to typology of application and service, the home page links can lead the users to web pages rather than launching a local application by means of Java Web Start technology. For instance, the Editor and Mapper are designed as local application while the Request For Change management is conceived as a web application.



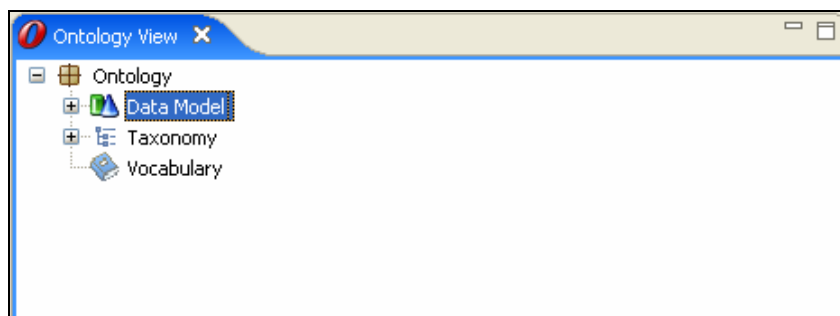
The Request For Change section should allow the GLOB Manager to browse the received requests and take the proper actions accordingly. For those profiles supposed to issue requests (e.g. COMM Manager) this section should provide a form in order to collect all the required information and submit the request to the selected GLOB Manager.

Appendix B: Ontology Editor user interface

The Ontology Editor user interface is described through screenshots with the twofold aim of showing the module usability and recalling its main functions.

Ontology view

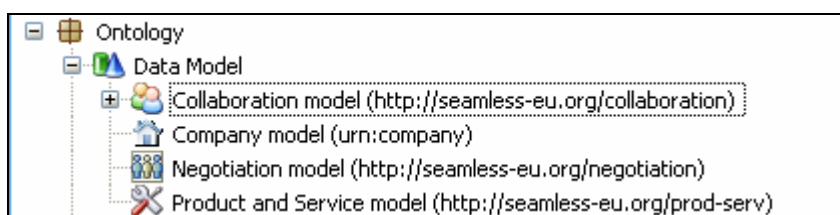
This view let the user manage the different ontologies which are constituted by tree fixed section: data model, taxonomy and vocabulary. Each section plays a specific role to create the ontology contents. More details can be found in the appropriate section. The view is organized as expandable tree showing the ontology name as root, at the second level the data model section with the taxonomy and the vocabulary.



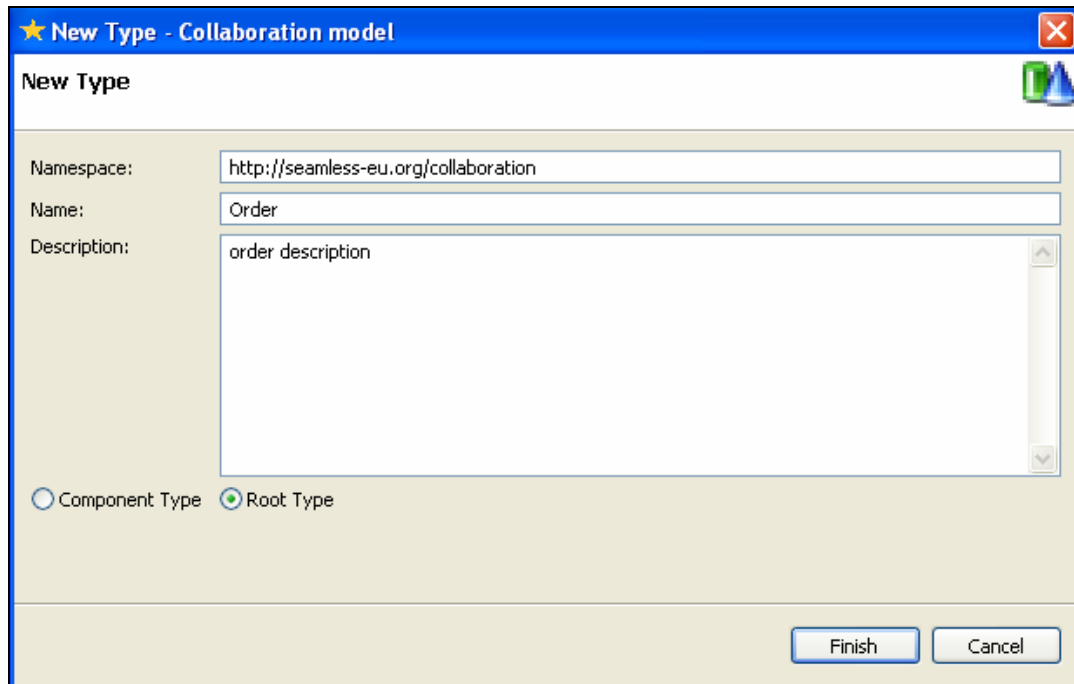
Data model

It references concepts needed to support partner description and search, negotiation and collaboration between partners. It is designed in four separate sub-trees mandatory component so that editor users are encouraged to organize their data in the same way: collaboration model, company model, negotiation model and prod/serv model.

Where the first component should contain business document (including order, invoice and dispatch advice), the second the description of the main concepts related to interesting companies, the third data involved in the negotiation process and the last one to represent family of products and their parameters description. Each component label consists of an image, the name of the component itself and the namespace.

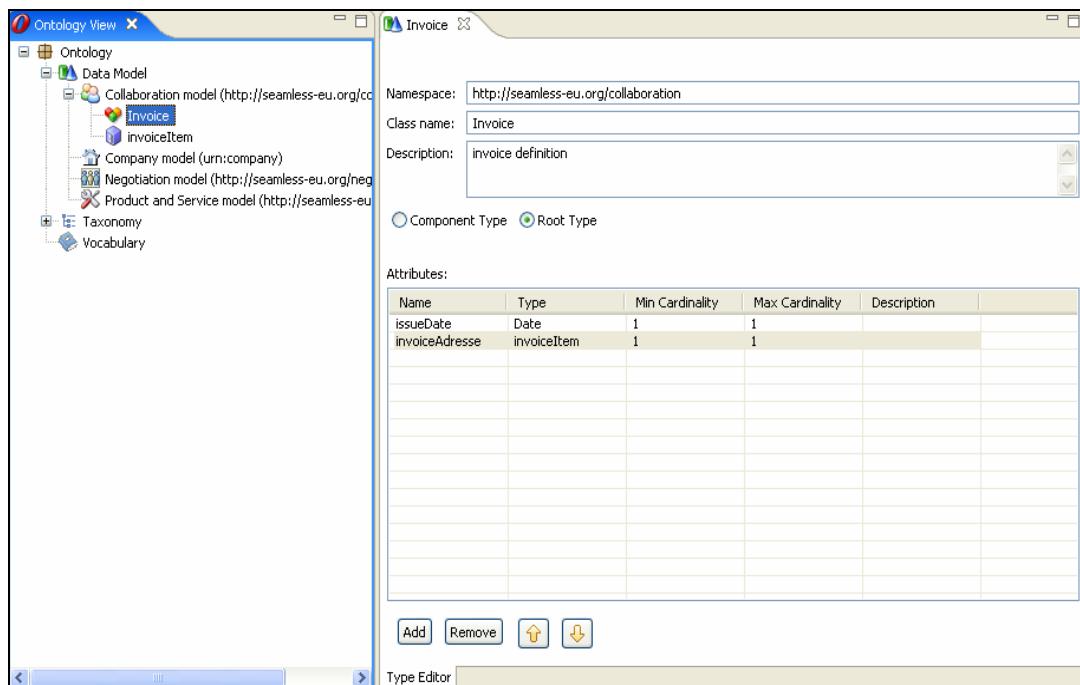


User can add a data model type to a component that will be hooked as son of it (in the example see Collaboration model).



Data model type

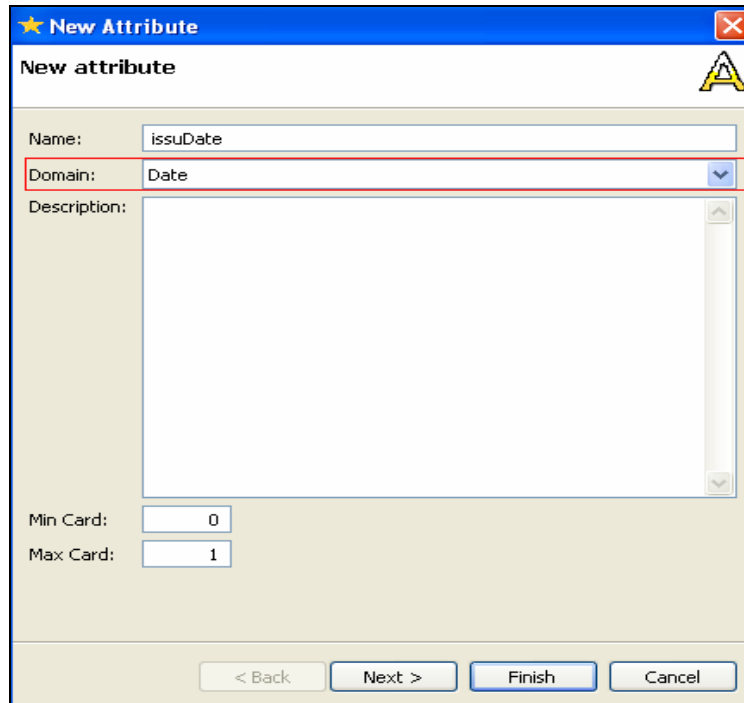
Once the new type is added, user is allowed, clicking on the type in the tree, to add or modify type main features (namespace, name, definition), to choose the type category (component or root type) or to add attribute.



To complete the definition of a type, user can add the attributes. Attributes are concepts that logically belong to a type. How to add or modify an attribute follows below.

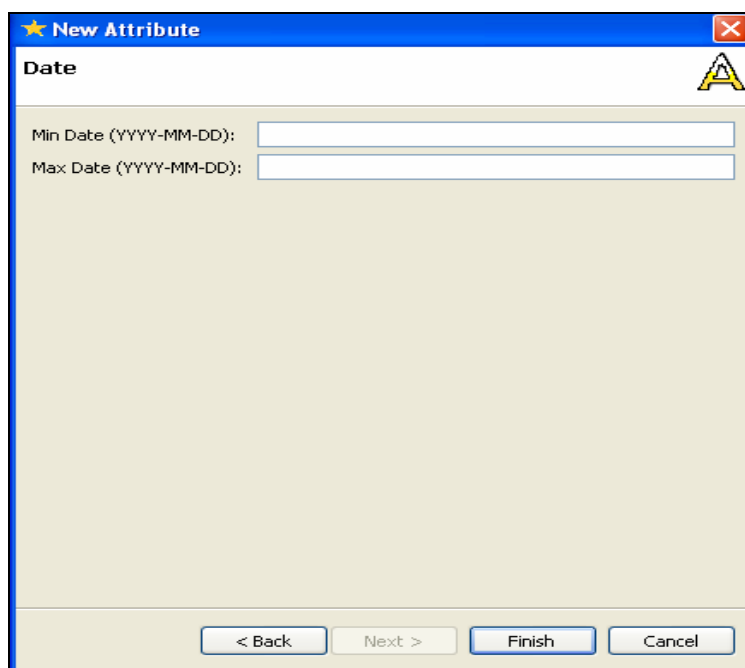
Data model type attribute

User chooses among a predefined set of attributes type which domain consists of: binary, date, date time, decimal, duration, enum param, integer, num param, taxonomy term, term, text, time and type. For all these types of attribute, user has to add the name, the definition, the min e max cardinality:



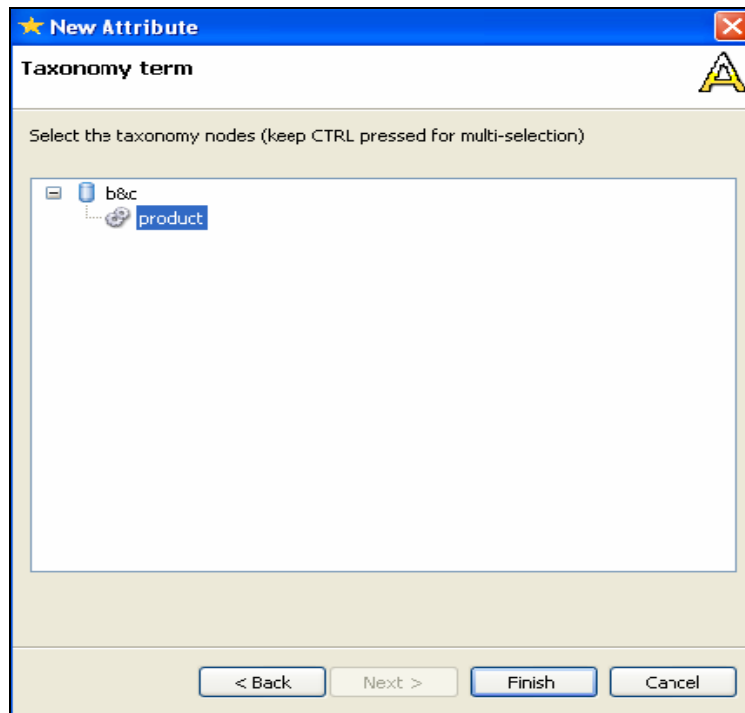
The screenshot shows a dialog box titled "New Attribute". It has a title bar with a star icon and a close button. The main title is "New attribute". There is a "Name:" field with the text "issuDate". Below it is a "Domain:" dropdown menu currently showing "Date". Underneath is a large "Description:" text area. At the bottom left, there are two input fields: "Min Card:" with the value "0" and "Max Card:" with the value "1". At the bottom right, there are four buttons: "< Back", "Next >", "Finish", and "Cancel".

In case of binary, enum param, num param, selection user is allowed to finish or to cancel the operation. In case of date, date time, decimal, duration, integer, text and time there is an ad-hoc second wizard page into which user can indicate min e max terminal point of each type (this page is not mandatory), e.g.:

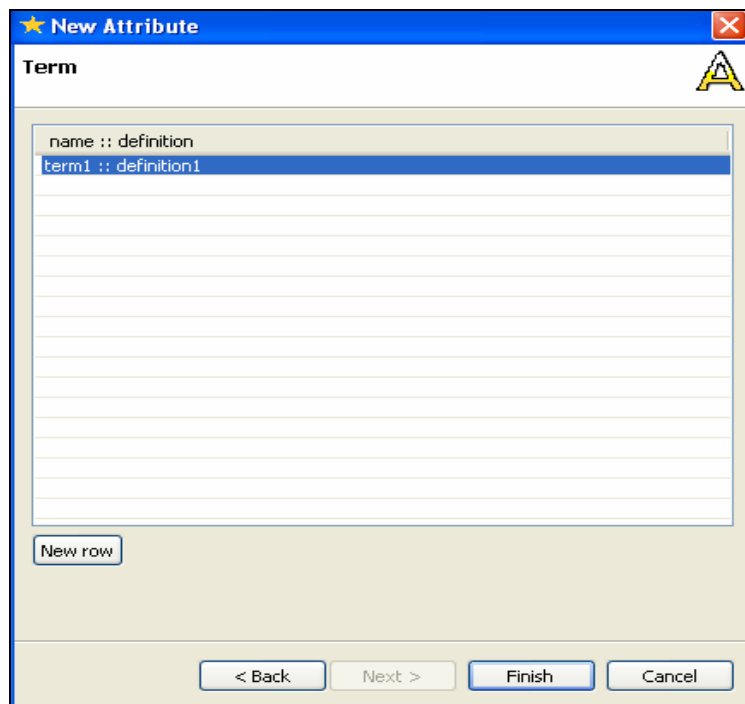


The screenshot shows a second dialog box titled "New Attribute". The title bar is the same. The main title is "Date". There are two input fields: "Min Date (YYYY-MM-DD):" and "Max Date (YYYY-MM-DD):". At the bottom right, there are four buttons: "< Back", "Next >", "Finish", and "Cancel".

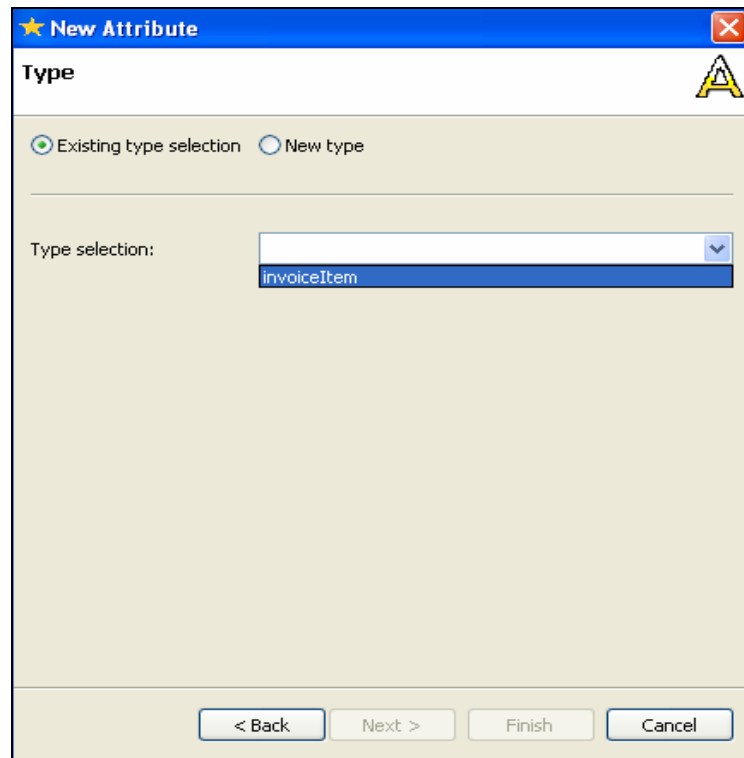
In case of taxonomy term selection, the non mandatory second wizard page provides the possibility to limit the taxonomy term instance values allowed. User can select a sector or a view point; the multi selection is permitted. Each sector corresponds to a cylinder image, the gear wheel to the view point.



In case of generic term selection, the non mandatory wizard page provides the way to limit the term instance values allowed. In this case user can choose one or more terms between existing terms or he can add a new term with definition.



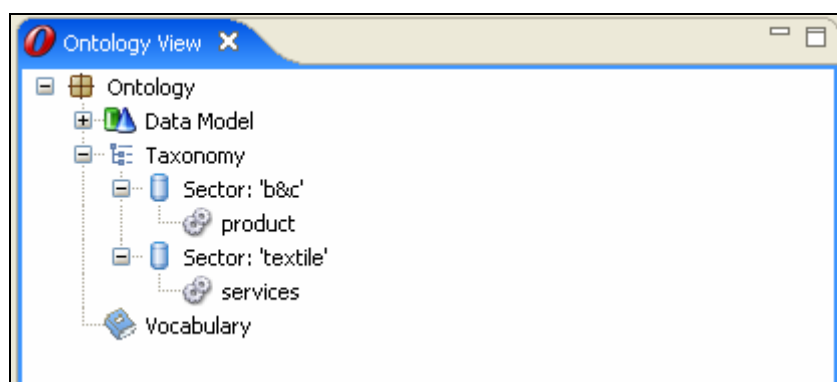
In case of type selection, user can decide to reference an existing type in the same component father (in this case COLLABORATION) or to define a new type (already seen above). In the first case the wizard appears as follows.



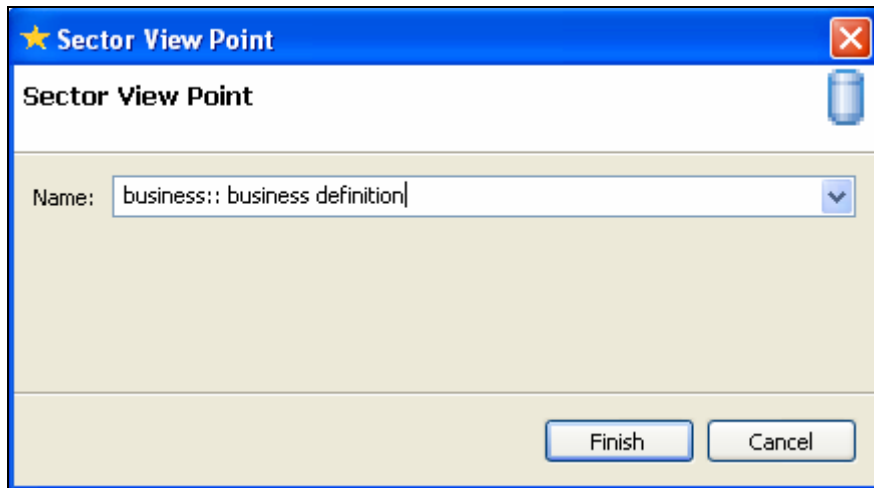
In general, each attribute in the table can be added, modified, or can be changed its position moving an attribute add or down.

Taxonomy

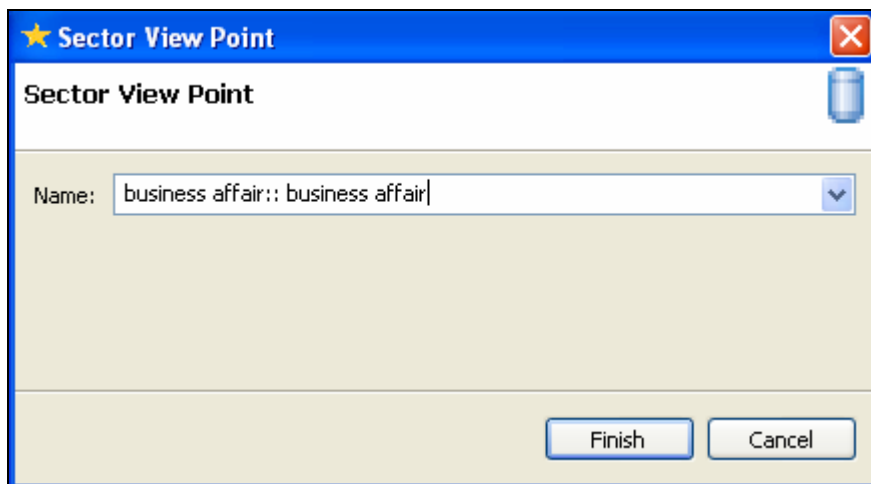
This section is used to classify products and services in term of general or sectoral standards. At the second level as 'Data Model' and 'Vocabulary', we can find the taxonomy sub tree. From this view, user is allowed to add one or many sectors and one or many view points for each sector.



In case of new sector, the wizard is the following:



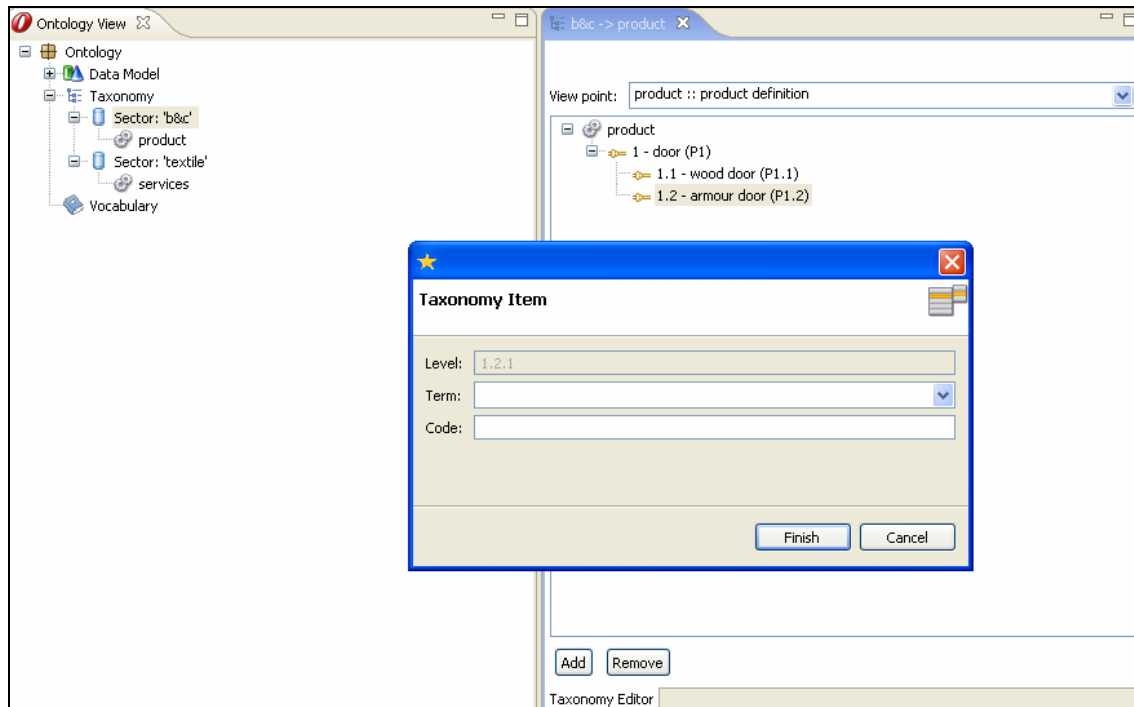
In case of new view point the wizard is the same.



Taxonomy Sector → Viewpoint

Once a view point has been added, user can add new taxonomy term to it. Interface allows also to modify the view point name and its definition at will.

To add a new taxonomy term, user has to choose at which view point level the term has to be added. Then user chooses the term and inserts the code (not mandatory), see below. This term is added at the chosen level, with the label representing the level, the term name and the optional code. In general, user can add or delete a term from a point of view selected.



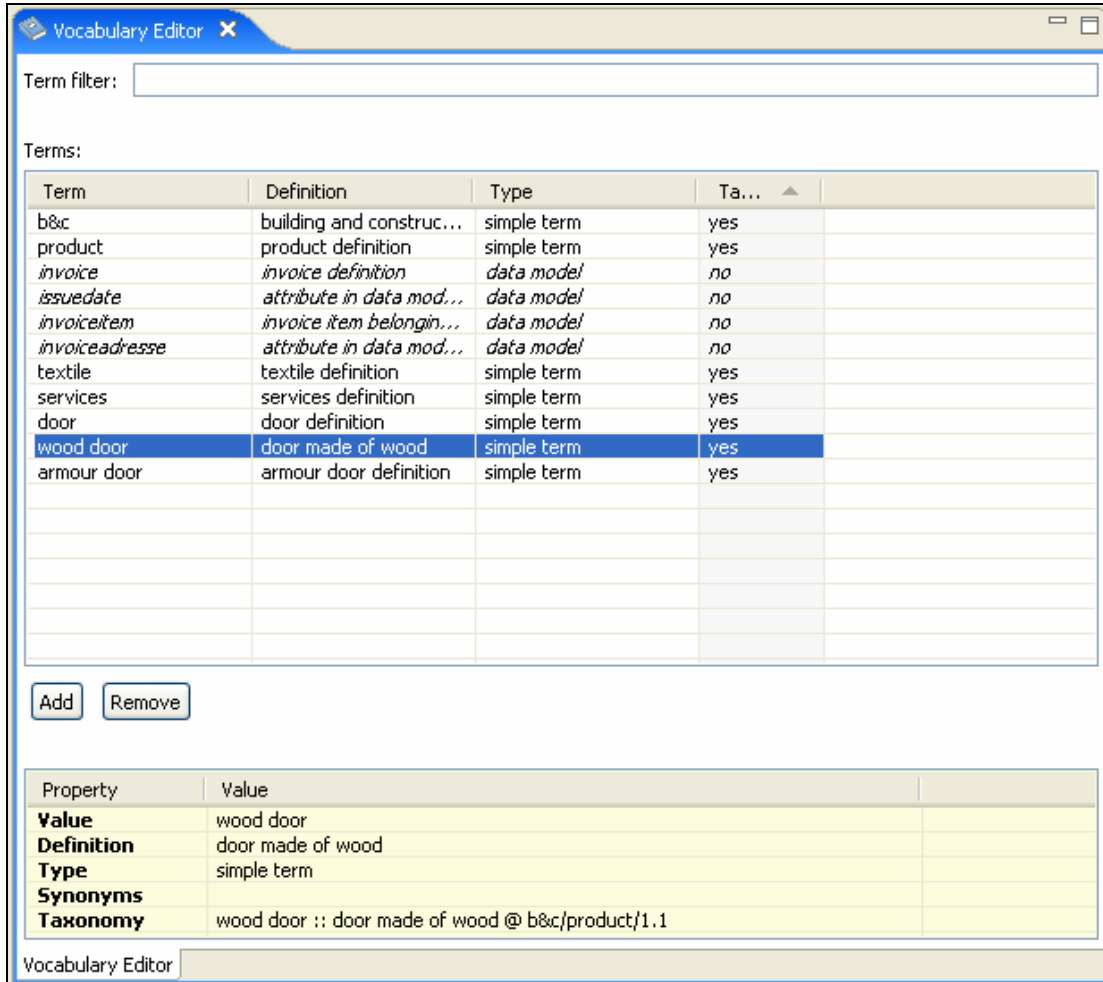
Vocabulary

It includes taxonomy terms, data model concepts and other general term. Its structure has two facets: one is thought in case of local ontology or global ontology (because one language a time must be managed) and the other in case of common ontology (because lingua franca and local language are managed at the same time). Both the first and the second share the fact that user is allowed to search and check vocabulary term properties.

The vocabulary editor page consists of a table containing read-only and changeable terms. The first set of terms contains the data model terms defined in the Data Model sub tree, while the second contains the remaining terms (taxonomy and vocabulary). The second table, that is hooked under the terms table, shows the properties of a term item selected in the table above. Its rows change on the base of the item selected.

Vocabulary (local language)

The table term consists of four columns, from left to right they are: term name, term definition, type of term (data model term rather than taxonomy or only vocabulary term) and the last one that explicitly tell us if the term was previously defined in the taxonomy sub tree.



The screenshot shows a software window titled "Vocabulary Editor". At the top, there is a "Term filter:" input field. Below it, a "Terms:" section contains a table with the following data:

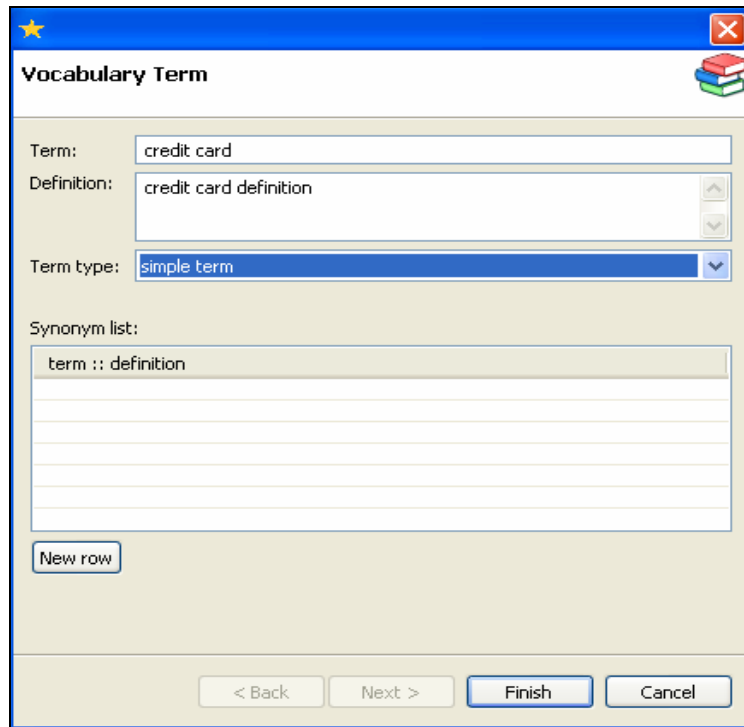
Term	Definition	Type	Ta...
b&c	building and construc...	simple term	yes
product	product definition	simple term	yes
invoice	invoice definition	data model	no
issuedate	attribute in data mod...	data model	no
invoiceitem	invoice item belongin...	data model	no
invoiceadresse	attribute in data mod...	data model	no
textile	textile definition	simple term	yes
services	services definition	simple term	yes
door	door definition	simple term	yes
wood door	door made of wood	simple term	yes
armour door	armour door definition	simple term	yes

Below the table are "Add" and "Remove" buttons. At the bottom, a detailed view of the selected "wood door" term is shown in a table:

Property	Value
Value	wood door
Definition	door made of wood
Type	simple term
Synonyms	
Taxonomy	wood door :: door made of wood @ b&c/product/1.1

The user is allowed to show the term properties, to add or delete a term from the table. The 'Term filter' field allows the filtered search among the entire set of term contained in the table.

In case of new term addition, user has to set some feature: name, definition, and specify the type of term (simple, enum, numeric). Moreover, each term can have more than one synonyms to be added as new term or to be chosen among the entire set of the already existing vocabulary terms. Term and definition are mandatory. In case of simple term selection, user then is allowed to terminate the term adding operation. Term addition mask is the following.



Vocabulary Term

Term: credit card

Definition: credit card definition

Term type: simple term

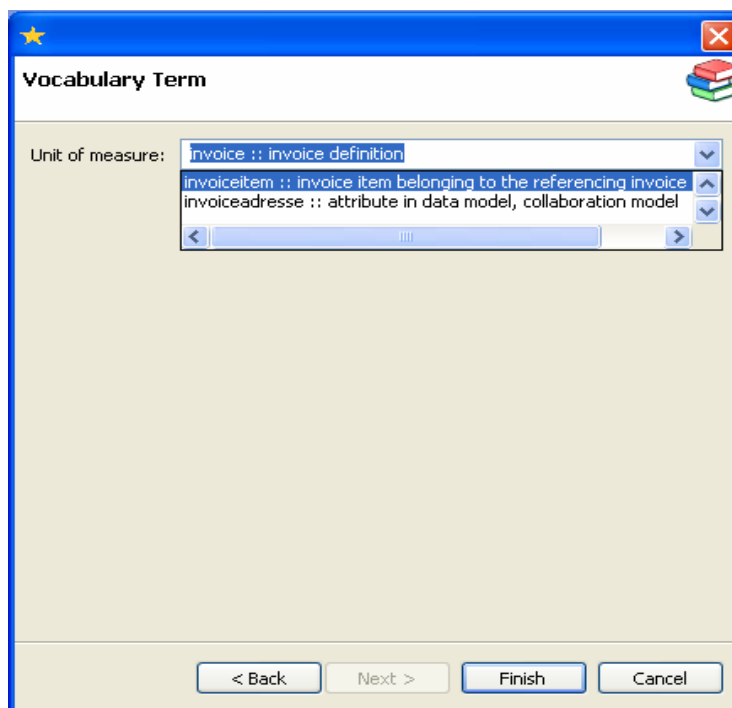
Synonym list:

term :: definition

New row

< Back Next > Finish Cancel

In case of term type equals to 'numeric type', user is driven to add the referencing unit of measure as new term or as vocabulary already existing term.



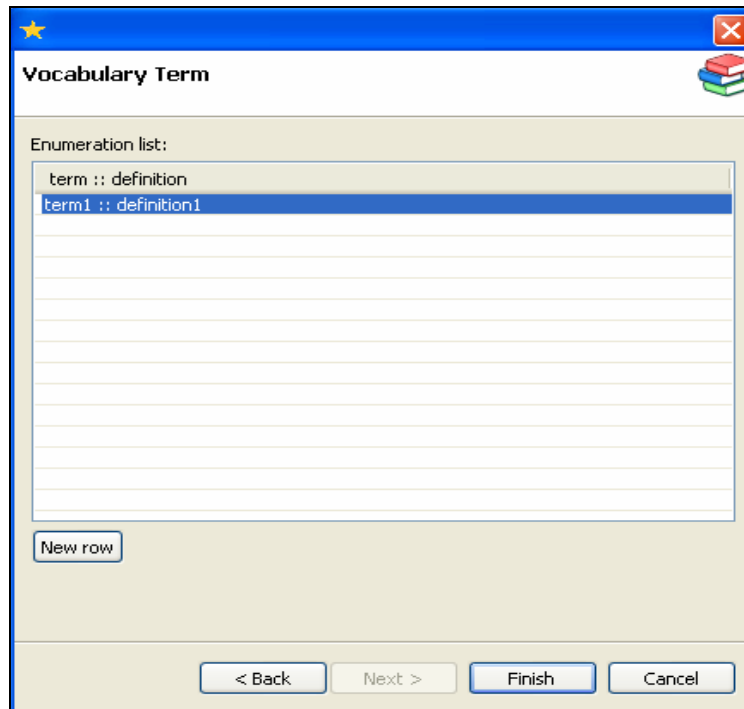
Vocabulary Term

Unit of measure: invoice :: invoice definition

- invoiceitem :: invoice item belonging to the referencing invoice
- invoiceadresse :: attribute in data model, collaboration model

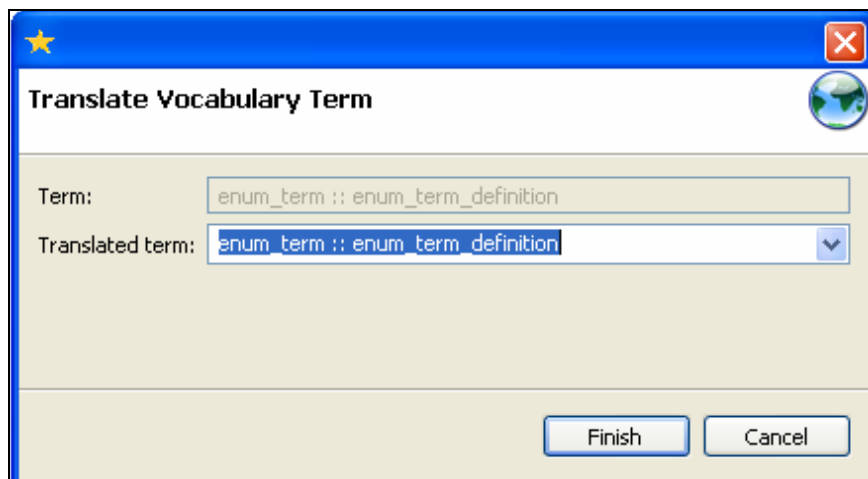
< Back Next > Finish Cancel

In case of term type equals to 'enum type', user is driven to add the referencing domain of allowed terms, representing the enumerative values of the added term. User is allowed to add at least one enumerative value while the max value is unbounded. The mask is the same.



Vocabulary (local and lingua franca)

This vocabulary version is obtained when the master ontology is downloaded by the user. The description is oriented to underline the differences with the local language structure previously described. Additionally, this term table has a column to manage the master term (English language). In fact the new feature of this vocabulary structure is the possibility to manage the multilingual translation (from English to local language). If user select a master term to be translated, the mask is the following.



In general all the adding, updating, deleting operation are equal to the VOCABULARY (LOCAL LANGUAGE) version.

Appendix C: Ontology Mapper user interface

The Mapper interface is described following the purpose of showing the UI (user interface) together with its main functionalities. When user starts with the Mapper execution then UI opens a multi tab form page, starting with the “Overview” page visualization. The other tabs alongside of it are “Structural”, “Taxonomy” and “Vocabulary”. These tab pages with their content and their functionality will be described below.

Mapper form editor

It is done of four parts: “Overview”, “Structural”, “Taxonomy”, “Vocabulary”. The starting page is the “Overview”, which is exploited by the user in order to choose the source and destination ontology information to be mapped. In particular, both for source and destination, user must select the ontology (first combo box choice on top), the structural section (second combo box choice) and the structural root type (third combo box choice) to load.

Structural section recalls the ontology editor data model design, in fact it is divided into the four data model parts: “Collaboration”, “Company”, “Negotiation” and “Product and Service”. Structural root-type is thought to manage one by one root types contained in the chosen structural of that particular ontology; this is done so that the mapping operation is atomic between a source and destination root type.

The “Overview” page is described by the screenshot below. In this example all the choice have been already performed.

Source Ontology		Destination Ontology	
Choose the source ontology information		Choose the destination ontology information	
Ontology:	Default source ontology	Ontology:	Default destination ontology
Structural section:	Collaboration	Structural section:	Collaboration
Structural root-type:	Fattura	Structural root-type:	CBHdrInv
<input type="button" value="Init Model"/>			
Overview Structural Taxonomy Vocabulary			

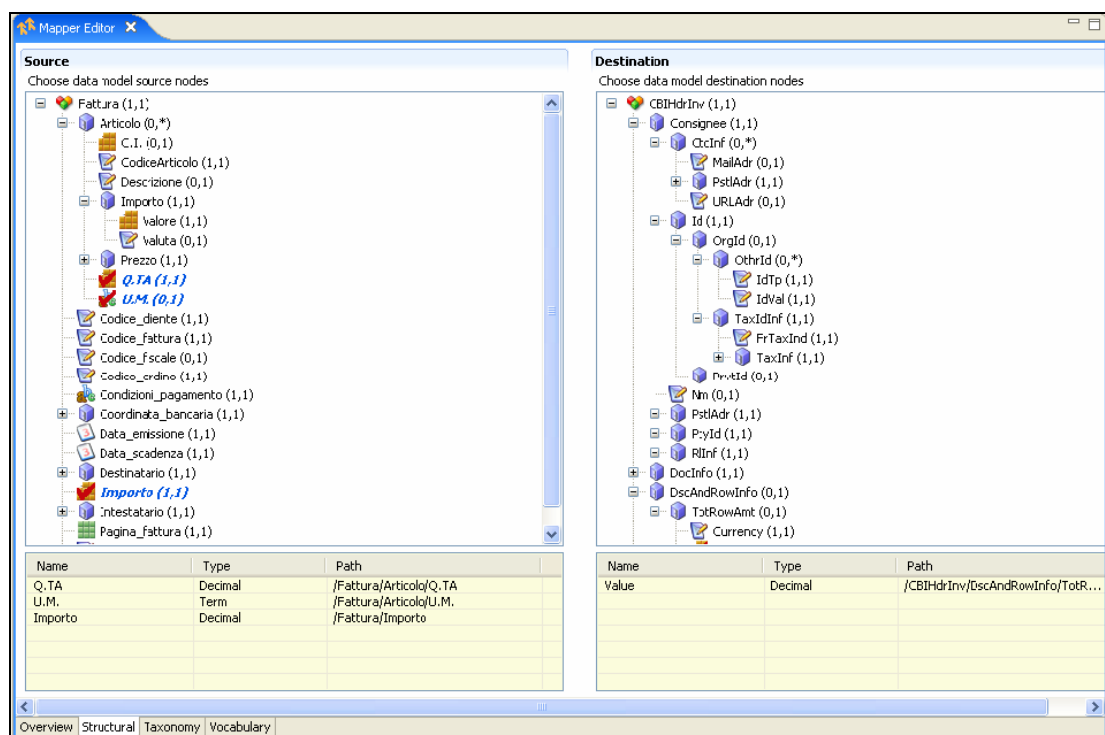
By pushing the “Init Model” button the ontology is loaded, the structural page contains the chosen source and destination root types, the taxonomy page contains the source and destination taxonomies and the vocabulary page contains the entire source vocabulary and the destination vocabulary.

Structural mapping

This page is organized in order to visualize a left side containing the source ontology root type selected in the overview page and a right side containing the destination ontology root type. Both source and destination root types are tree structures containing component and simple type. The mapping associations can be performed only among simple types. Root type, component type and simple type are characterized by a type-unique icon, a name indicating the node name and by the minimal and maximal occurrence of that element between “()” parenthesis.

Each tree is logically linked to a table (positioned under it) which shows some characteristics about the node/s selected: name, type of the simple attribute and relative root type path. User is allowed to click on a row of the table to search the respective node in the tree; it can be useful when the table contains many rows referencing nodes of the tree that are hidden up or down respect to the window containing the shown part of the tree. The selected node replaces his icon into an icon with the same symbol plus a red ‘V’.

The relative image is the sequent.

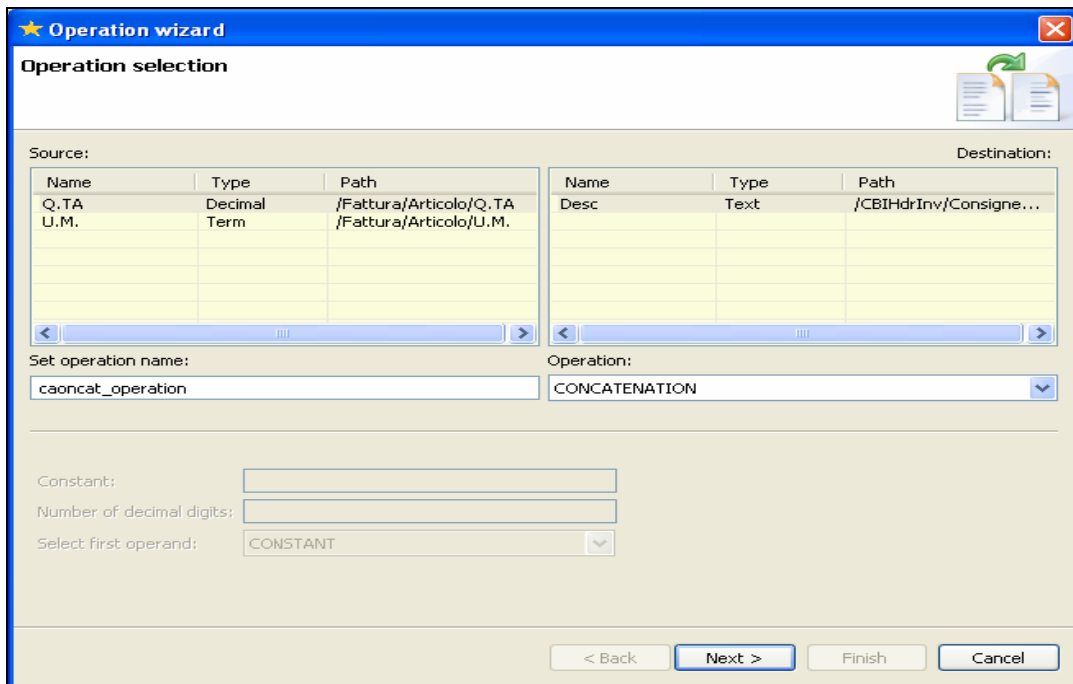


Structural mapping operation

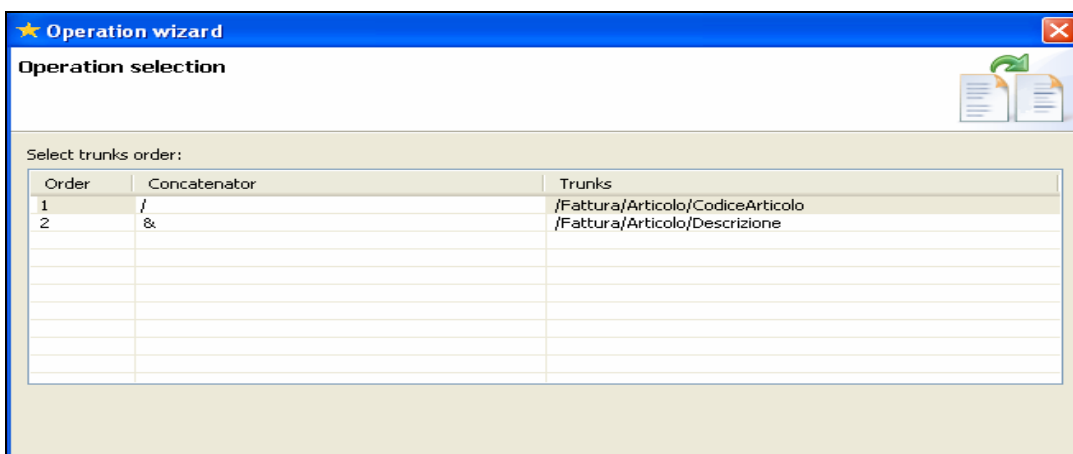
Once the user has selected at least one node both in the source and in the destination tree, then he is allowed to perform a mapping operation. The entire set of operation includes: copy, arithmetic operation (addition, subtraction, division, multiplication), split (with delimiters, with length fields). The system, automatically, on the basis of the selected source and destination nodes, permits the user to exploit a subset of the total mapping operation set.

To visualize the allowed set of operations and to perform one of them, user pushes the button available on the toolbar or selects the voice in the “Add operation” in the action menu. Then a wizard is shown so that the user is able to assign a fictitious name to the operation that will be selected sideways. In case of arithmetic operation user is allowed to choose a numeric constant to be comprised in the calculation; moreover, user can decide the number of decimal digits allowed in the result and the first operand in the calculation (by far necessary in case of division and subtraction operation).

The screenshot below shows the case of a concatenation operation.



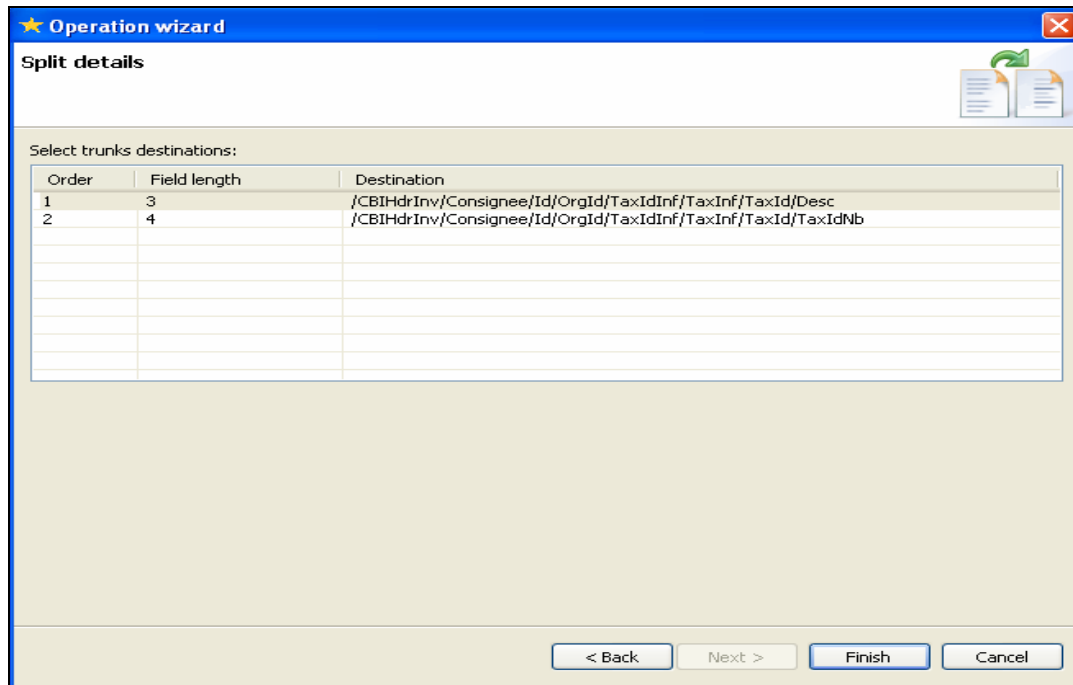
- In case of **COPY** (copy of content), user is allowed only to finish the operation.
- In case of **SUM** (arithmetic sum), **DIVISION** (arithmetic division), **SUBTRACTION** (arithmetic subtraction) and **MULTIPLICATION** (arithmetic multiplication), user can fill the constant and the number of decimal digits fields and then he can select the first operand of the operation (only for subtraction and division). Then he is allowed to finish the operation.
- In case of **CONCATENATION** (the act of linking together as in a series or chain), user goes into the next wizard page with the scope of selecting the elements (trunks) order and of choosing a concatenator (separator) to add after each element. The concatenator that usually is a character or a generic string is not mandatory. The wizard's screenshot of the above description follows.



In case of **SPLIT_WITH_FIELD_LENGTHS** (separating a string in many pieces indicating the length of each part), user goes into the next wizard page to select the order of the destination nodes (receiving

the content of the source split parts) and the length fields (only an integer is allowed) of the source nodes. The field length information are mandatory.

The relative screenshot is:



In case of **SPLIT_WITH_DELIMITERS** (separating a string in many pieces indicating the splitting characters), user goes into the next wizard page to select the order of the destination nodes (receiving the content of the source split parts) and the splitting characters of the source nodes. The relative wizard page is similar to the image above, referencing to the other type of splitting, with a difference in the second column that can be filled with both characters and numbers.

In general, after a mapping operation is performed, the destination node/s involved are characterized by a new icon (the same icon of the selected case, but grey scale coloured) indicating that they will not be used for other mapping instances. In fact the Mapper rules force the use of a destination node just once.

To show, modify or delete mapping instances, user selects an other visualization view called "History view" that will be treated later.

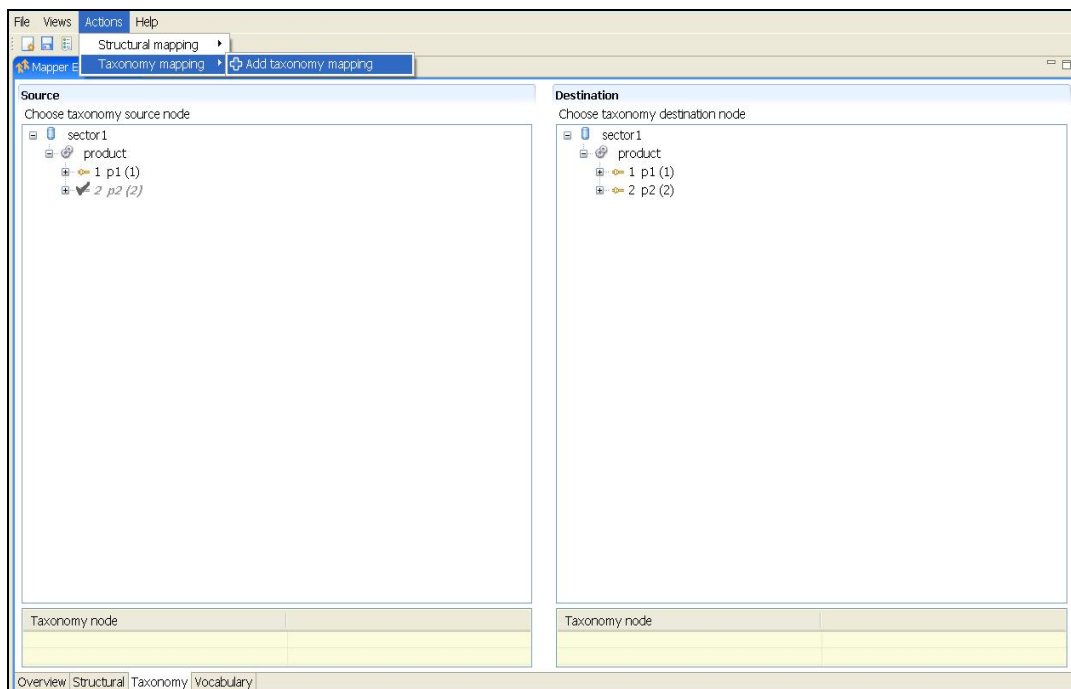
Taxonomy mapping

The UI of this page is similar to the structural mapping page. In fact on the left user can find the tree structure representing the source taxonomy, on the right the same structure for the destination taxonomy. Both taxonomies are linked to a table which contains a string representing a concatenation of the sector name with its definition, view point name with its definition and the node name with its definition all information referencing to the node selected. In order to perform a mapping operation, user initially must choose a source node and a destination node, then he has to click "Add taxonomy mapping" choice of the "Actions" menu.

In general, after a mapping operation is performed, the source node involved is characterized by a new icon (the same icon of the selected case, but grey scale coloured) indicating that it will not be used for other mapping instances.

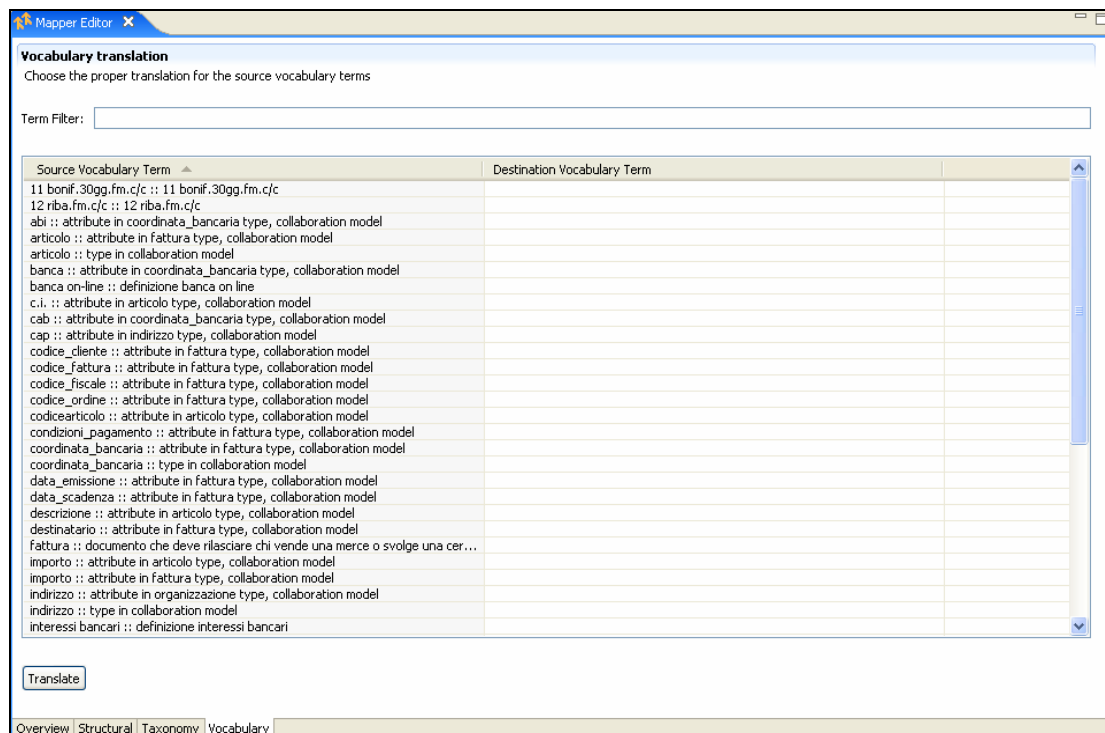
To show, modify or delete mapping instances, user selects an other visualization view called "History view" that will be treated later.



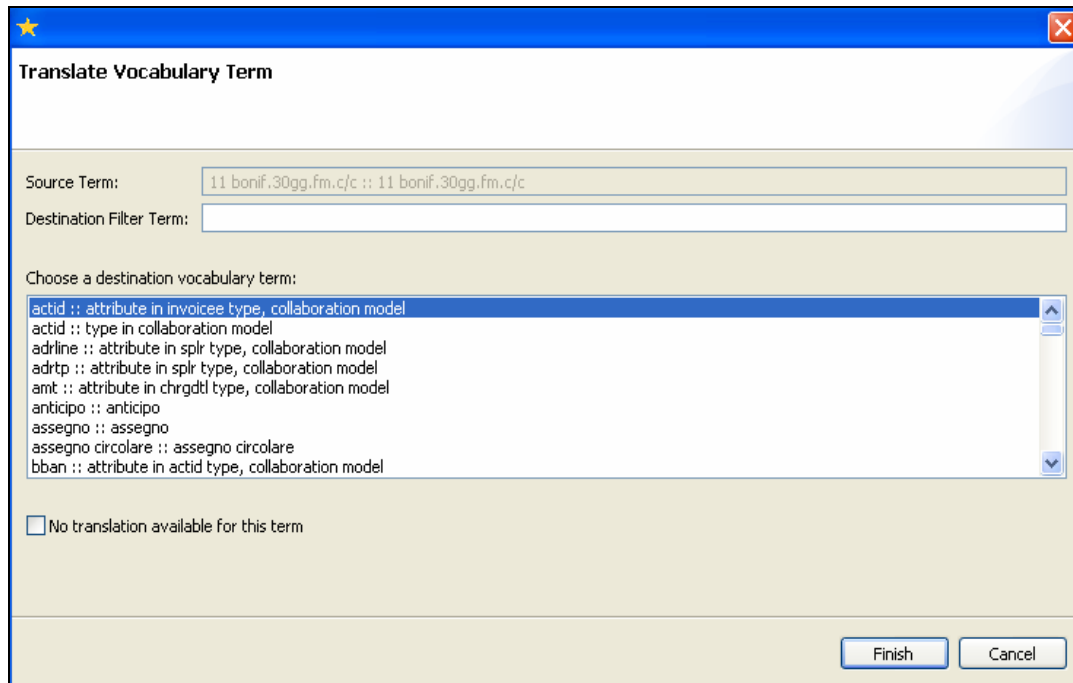


Vocabulary mapping

The vocabulary interface is quite different from the taxonomy and structural ones. The main part is a double column table. The left column contains the source vocabulary term (term and definition), the right one contains the destination vocabulary term that is chosen in a further step that will be explained below.



In order to apply a transformation of a source term into a destination one, user is requested to push the “Translate” button. Immediately a wizard opens. This wizard allow the user to select a destination term name from a list (to be able to do that, user must uncheck the check button down). The image of this wizard follows:



In case of change of a mapping row of the vocabulary table, user is allowed to select the row and push again the “Translate button”. After that he repeats the steps explained above in case of new mapping instance.

Mapper history view

This view that is hidden to the user when the Mapper is launched, can be accessed by clicking a button,

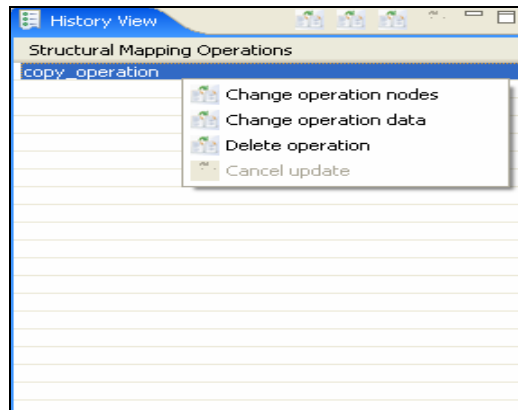


or selecting the menu voice “History view” in the “Views” menu button. This view is used by the structural mapping and the taxonomy mapping. It is a list of the mapping operation saved. Its content is double and depends on which mapping the user is working to at that moment (i.e. in case of structural mapping the history view contains only the structural mapping operations). In both cases, the history view contains a list of operations.

Structural mapping history view

This version permits the user to show (with a mouse double click), to modify and to delete information involved in the selected operation mapping (with a mouse right click). If the user double clicks on a operation, the source and destination nodes, involved in the mapping, are pointed out with a different colour respect to the background.

Right clicking on the operation row or pushing the buttons in the history view toolbar, the user is allowed to change node both in the source and in the destination, or to change operation data (the wizard for the operation selection, described above, is opened), or to delete the operation and finally to cancel the update operations just performed.



Taxonomy mapping history view

In this case the user is able to point out the nodes in the source and destination tree involved in a mapping operation, or to cancel the mapping row (it contains a concatenation of source and destination taxonomy terms with respective definitions).

