KP-LAB Knowledge Practices Laboratory – Specification of the SWKM Knowledge Evolution, Recommendation, and Mining services

Pavel Smrz, Vilem Sklenak, Vojtech Svatek, Martin Kavalec, Martin Svihla, Jan Paralic, Karol Furdik, Peter Bednar, Peter Smatana, Nicolas Spyratos, et al.

27490

KP-LAB

Knowledge Practices Laboratory

Integrated Project

Information Society Technologies

D5.3: Specification of the SWKM Knowledge Evolution, Recommendation, and Mining services

Due date of deliverable: **30/09/07**
Actual submission date: **09/11/07**

Start date of project: 1.2.2006 Duration: 60 Months

Organisation legal name of lead contractor for this deliverable:
UEP: Vysoká škola ekonomická v Praze (University of Economics, Prague)

Final

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Contributor(s):	Pavel Smrz	UEP	smrz@fit.vutbr.cz
	Vilem Sklenak	UEP	sklenak@vse.cz
	Vojtech Svatek	UEP	svatek@vse.cz
	Martin Kavalec	UEP	kavalec@vse.cz
	Martin Svihla	UEP	svihla@vse.cz
	Jan Paralic	TUK	Jan.Paralic@tuke.sk
	Karol Furdik	TUK	kfurdik@stonline.sk
	Peter Bednar	TUK	Peter.Bednar@tuke.sk
	Peter Smatana	TUK	Peter.Smatana@tuke.sk
	Nicolas Spyratos	LRI-ORSAY	spyratos@lri.fr
	Hanen BelhajFrej	LRI-ORSAY	hanen@lri.fr
	Mamadou Nguer	LRI-ORSAY	nguer@lri.fr
	Vassilis Christophides	ICS-FORTH	christop@ics.forth.gr
	Dimitris Kotzinos	ICS-FORTH	kotzino@ics.forth.gr
	Yannis Tzitzikas	ICS-FORTH	tzitzik@ics.forth.gr
	Giorgos Flouris	ICS-FORTH	fgeo@ics.forth.gr
	Giorgos Markakis	ICS-FORTH	geomark@ics.forth.gr
Editor(s):	Pavel Smrz	UEP	smrz@fit.vutbr.cz
Partner(s):	UEP, TUK, LRI-ORSAY, ICS-FORTH		
Work Package:	WP5 – Semantic Web Knowledge Middleware		
Nature of the deliverable:	Report		
Internal reviewers:	Hadj Batatia	INPT	hadj.batatia@enseeiht.fr
	Markus Holi	EVTEK	markuho@evtek.fi
Review documentation:	http://www.kp-lab.org/intranet/work-packages/wp5/result/deliverable-5.3		

Version history

Version	Date	Editors	Description
0.1	June 16, 2007	Karol Furdik, Pavel Smrz	Initialization document, tasks and responsibilities.
0.2	June 29, 2007	Hanen BelhajFrej	Notification module specification.
0.3	July 18, 2007	Giorgos Flouris	Initial comments, structure modification, inputs to Knowledge Mediator section.
0.4	August 14, 2007	Karol Furdik, Peter Bednar, Peter Smatana	Integration of inputs, Requirements section, Text Mining services functionality and architecture.
0.5	August 28, 2007	Giorgos Flouris, Karol Furdik, Jan Paralic	Sections 1 and 2 upgraded by FORTH and TUK
0.6	August 30, 2007	Pavel Smrz	Sections 3.2.5 and 4.2.2 updated, + minor modifications in the Executive Summary and Introduction; an update of section 3 from FORTH included
0.7	September 21, 2007	Pavel Smrz	Integrated version, updates from all partners
0.8	September 24, 2007	Karol Furdik, Jan Paralic, Peter Bednar	Renumbering of sect. 3.2.2, new text added to sect. 3.2.2.3.
0.9	September 26, 2007	Pavel Smrz	Last changes from Giorgos, service signatures normalized
1.0	November 09, 2007	All contributors	Incorporation of reviewers comments and other minor changes

Executive summary

This deliverable presents the deep-level specification for the second release (M24) of the components responsible for advanced manipulation with the knowledge stored in the KP-Lab Semantic Web Knowledge Middleware (SWKM). The two components were defined in [D5.1] as Knowledge Mediator and Knowledge Matchmaker.

The *Knowledge Mediator* services (*change*, *comparison*, *versioning* and *registry*) aim at providing functionalities to support evolving ontologies and RDF Knowledge Bases (KBs). Upon a change request, the change service will automatically determine the effects and side-effects of the request and present it to the caller for validation. A comparison service is necessary to allow one to compare two versions of an ontology or RDF KB and identify their differences. The above functionalities are coupled with a versioning system, which is used to make different versions of the same ontology (or RDF KB) persistent, and with the registry service, which allows the user to classify the stored ontologies, using some related metadata for easy access and manipulation.

The *Knowledge Matchmaker* supports advanced *mining* and *notification* services for knowledge artefacts. It essentially enables to *cluster*/*classify* available information resources with respect to the employed ontologies, as well as, to *notify* about changes to content items produced/consumed within a group of learners according to explicitly *subscribed* preferences [DoWB].

The *Notification service* supports access to the knowledge repository for KP-Lab users (i.e. individual human users as well as various tools or software components) by keeping them aware of changes. Users will be able to *subscribe* their preferences to the KP-Lab system in order to be *notified* about the changes in the knowledge repository. Events (modifications) in the repository are *matched* with the subscriptions and notifications are *propagated* automatically to the users.

Text Mining services are used to assist users when creating or updating the semantic descriptions of KP-Lab knowledge artefacts. The *Classification Service*, after a software-training period, will classify the artefacts under some pre-defined set of categories (e.g., ontology concepts), resulting in a semi-automatic generation of semantic descriptions. The *Clustering Service* will look for clusters of similar artefacts and automatically acquire conceptual maps from knowledge artefacts. This can lead to the update or even the creation of (new) KP-Lab ontologies managed in the sequel by the Knowledge Mediator.

The services are described along with the proposed functionality for each one, based upon the motivating scenarios and the subsequent functional requirements. The functionality of the services is presented from the end-user perspective and divided into parts that form the major components of the SWKM knowledge evolution, recommendation and mining services architecture.

Table of Contents

1 Introduction

In the context of KP-Lab we need to support the creation, evolution and management of conceptualizations for various domains. Such conceptualizations are necessary for learners to engage in trialogical learning and have a shared space, upon which they can represent their own, as well as other learners', knowledge and understanding of the domain at hand. The role of the conceptualization in this respect is to be used as a mediator tool among people attempting to describe and understand the domain at hand.

The simplest way to represent knowledge in such a conceptualization is by introducing structure in a vocabulary of terms, in effect producing a *taxonomy*. A taxonomy enriched with different types of constraints, relationships and rules forms an *ontology*. There are various formal languages that can be used to represent these relationships; for the purposes of KP-Lab, we adopt the RDF model with the semantics of RDF Schema (RDF/S); for a detailed description of RDF and RDF/S, see [KMACPST04]. An ontology can be viewed as the schema upon which data can be classified; an RDF ontology coupled with data items (instances) is called an *RDF Knowledge Base* (or *RDF KB* for short).

The descriptions of knowledge artefacts as well as their involved conceptualizations will be represented and handled in SWKM as RDF/S schemas and resource descriptions (i.e., ontologies and RDF KBs). In order to support personal and group knowledge management based on multiple conceptualizations the knowledge repository should be able to distinguish schemas and descriptions according to the actors (individual or group) involved in their creation. To this end, the SWKM knowledge repository will be able to store, retrieve and update RDF/S schemas and descriptions based on the *name spaces* or *graph spaces* they belong to, where a namespace is a collection of RDF/S classes and properties, whereas a graph space is a collection of RDF triples (see [D5.1] for more details). Name and graph spaces will be uniquely identified using URI references. Name and graph spaces may depend on other name or graph spaces, in the sense that they may reuse elements (e.g., classes) or declarations from other name and graph spaces (see [D5.2] for more details); such dependencies may need to be taken into account in certain services, as we will see in later sections.

Notice that mediation of activities is not limited to physical tools but encompasses linguistic, conceptual, as well as cognitive artefacts, including theories, models and languages [Sta03]; a conceptualization (and an ontology representing a conceptualization) is such a non-physical mediator tool. Apart from mediating artefacts used to carry out purposive activities, ontologies (and RDF KBs) can also be seen as knowledge-artefacts on their own. This understanding imposes a number of implications [AMR06]. First of all, the ontology (or RDF KB) is part of the activity system; as a result, its utility for the task at hand is bound to the activity itself and cannot be assessed independently. Secondly, an ontology (or RDF KB), like any other mediating artefact, is the result of a cultural-historical development process within a certain community. As mediating artefacts are objectifications of socially shared knowledge and are built on specific premises it is likely that ontologies (and RDF KBs) not only vary in their terminology but also reflect different theoretical

foundations. Thirdly, an ontology (or RDF KB) can become the object of an activity itself and can be modified or transformed.

2 Requirements

The requirements process of the WP5 software release 2 follows the general KP-Lab's design approach [D2.1] and is based on the idea of intertwined design of software components, practices and agents [D2.2]. The initial set of requirements was given by the reactions of developers and end-users, given on the first release of the SWKM. This includes all the inputs obtained from partners to the first prototype (V1.0) of the Knowledge Mediator, Repository and Manager [D5.2].

In parallel, the analysis of the current educational and professional scenarios was carried out within the Working Knots co-ordinated by WP2. The co-design process can be exemplified by the process followed in the Working Knot "Collaborative Semantic Modelling", the requirement engineering for collaborative semantic modelling was performed in a highly interactive manner [CSMWK]. The requirements are related to the tool for "Collaborative Visual Language and Models editing", for the functional specification on "Creating and modifying ontology based concept maps (visual models)" M6.2 as well as the one on "Creating and Modifying visual modelling languages" M6.4.

Motivating scenarios were specified in co-operation with pedagogical and technical partners to define a practical usage of extended ontology manipulations in a real-world learning environment. The motivating scenarios selected for presentation in this section were originally defined in the [D8.1] and further elaborated within the Working Knots "Project and Content Management" [STBPL] and "Collaborative Semantic Modelling" [CSMWK].

Prototypes for particular components and services were produced as a result of requirements elicited in the face-to-face and virtual workshops of pedagogical and technical partners. High level requirements as well as consequent usage scenarios were specified by technical and pedagogical partners. In addition, the pedagogical partners provided a set of resources – real course materials and artefacts that were used by technical partners for development and testing of knowledge evolution services (especially the text mining services for classification, clustering and concept map creation).

2.1 Motivating scenarios

This section introduces two motivating scenarios that were chosen as the most relevant for the use of SWKM services. Motivating scenarios were specified as a framework for using the collaborative modelling in practice. This includes such procedures as collaborative development of visual models as well as the underlying modelling language, specification of semantics for the modelling elements, comparing multiple model's, preserving mutual consistency of the models, etc.

The major advantage of the two scenarios presented below is that they motivate all the services described in the following sections. The scenarios were developed by "Collaborative Semantic Modelling" and "Project and Content Management" Working Knots. For more details and additional scenarios, please refer to [CSMWK] and [STBPL]. The scenarios mention domain-specific ontologies to be determined and designed by KP-Lab system users.

2.1.1 Semantic tagging

A group of students, researchers, or co-workers is given a set of research papers and asked to identify the topics discussed in these papers and to build an ontology representing the topics discussed. Moreover the group is asked to annotate the original set of papers according to the derived ontology. The members of this group should collaborate in order to carry out this task [TCFKMPS06].

Two particular subtasks can be identified within this basic scenario, namely ontology creation procedure and semantic annotation of the artefacts. These two subtasks can be supported by semi-automatic mechanisms of concept map creation, clustering and classification, using the text mining capabilities [Smrz et al 2007].

The semantic annotation of learning materials (papers, documents, or knowledge artefacts in general) according to a pre-defined classification ontology (PBL vocabulary [STPBL]) and consequent semantic search [SEMSRCH] are required capabilities of the Shared Space, since they enable to share and exchange the information together with its semantic context (meaning) between learners. The classification based on text mining methods is an effective way to support the semantic tagging in the Shared Space. That is why the semantic tagging was taken as a main motivating scenario for classification services.

The semantic tagging, also sometimes referenced as (semantic) annotation[1], is a procedure of enriching a document (or knowledge artefact in general) by an additional information that somehow expresses the content or features of the document. The information that describes the document is taken from hierarchically organised vocabulary of terms (keywords, phrases) – semantic tags. This way, the semantic tagging helps to manage and maintain the (possibly large) set of documents produced by learners during the project within Shared Space. It also enables to understand connections and relations between different documents and activities required in the producing of the product in a particular project. Consequently, the semantic tagging supports a search that can be made according to the used semantic tags – the semantic search.

In [STPBL], the PBL vocabulary was designed as a hierarchy of different types of items that are produced during the Shared Space project, and describes activities related to the project management and production of end artefacts. Furthermore, a list of terms describing the possible linking alternatives between the content items and defined tasks are presented in the end of the vocabulary. However, the PBL

[1] Semantic annotation is a broader term than tagging. The annotation enriches an artefact by means of concepts from general ontology, while tagging uses a predefined hierarchically organised vocabulary of keywords (terms, tags).

vocabulary does not present the existing metadata that is already in the current Shared Space. These are, for example, the automatically created metadata as e.g. *Creator*, *Creating date*, *Modified date*, or the user defined metadata *Responsible of*, etc.

Various variants of the basic scenario for collaborative annotation can be imagined. For example, the collaboration could be either synchronous (i.e., all learners make changes in the classification ontology and the classification data synchronously), or asynchronous (i.e., the learners edit the classification ontology and data in a separate space and commit the changes they want). Moreover, each learner could have his or her own personal space with a copy of the ontology; in such a case, the central ontology could be derived from the personal ones. Commitment of a learner's changes upon the central ontology could be either instantaneous, or it could pass through a process which could include some kind of approval mechanism, according to the policy of the user application. The latter mechanism could also include some kind of argumentation (see the next scenario).

The aforementioned group collaboration requires often changes in the classification ontology, as the members of the group constantly discuss the information found in the classification ontology, leading to additions, deletions or other edits and corrections to the ontology. The same is true during the classification of the various documents (e.g., papers) to the resulting ontology. During the process, the learners may need to keep different versions of the ontology (and classification data), so as to revisit older versions in case they want to undo some change. In addition, they may need to view the changes that some member of the group made, by comparing two different versions of the ontology (before and after the change).

Notice that the described requirements imply that there may be more than one (versions of) ontologies that are stored in the repository. This indicates the need for an ontology registry that will classify the stored ontologies based on the ontologies' related metadata information for easy access and retrieval.

The classification process may be enhanced using text mining services, whose output may be useful as a suggestion tool for a semi-automatic classification of the documents in the ontology, in effect initiating an evolution process. A notification mechanism may be also useful, as the learners may need to be notified when changes in the classification of papers arise. The same might happen when the ontology evolves.

2.1.2 (Re-)Constructing arguments

The main idea of this scenario is having a group of people (students, researchers, co-workers, etc), possibly with different backgrounds and/or from different fields, that meet in order to reach a decision on some issue. In order to scaffold this process the group is presented with an argumentation ontology which could be inspired by similar efforts in the literature (e.g., [GK97], [Tou58]). Said ontology could also be used to annotate related resources. For example, a certain claim might be backed up with a link to a respective resource.

In a scientific environment [BSD05], this scenario could involve re-constructing pre-existing scientific arguments based on a set of research papers, or explicating the

group members' own arguments. In a professional environment, it could involve the improvement of the design and function-ability of a company's new products. For example, there can be a group of a market-analyst, an information technology expert, a person responsible for PR and a businessperson from an undisclosed big Finish company which should collaboratively acquire knowledge on how to improve their new mobile phones and increase the company profit. Every member of the group prepares a set of resources describing his or her current understanding (view) of the given topic. The extraction engine produces an overall conceptual map, which integrates the individual views and provides a basis for the core discussions of the group.

The group of people collaborating in this scenario need to reconstruct their argumentation in a KB using the provided argumentation ontology; there is a single RDF KB representing the arguments of the entire group. During collaboration, differences in opinions may arise which should be discussed and resolved in a synchronous manner. Such dispute resolution will cause changes in the original construction of the argument, thus leading to changes in the original argumentation; in this case however, the changes affect the data portion of the RDF KB rather than the schema. Moreover, changes are only additions, i.e., there are no deletions or modifications.

Like in the previous scenario, the learners may need to store different versions of their argumentation and compare them using appropriate delta functions. We may have different groups of people who use different argumentation frameworks, in which case the system may need to support the storage, classification and retrieval of more than one namespaces through the use of some registry. Moreover, the learners may want to be notified for new entries in the registry.

2.2 High-level functional requirements

2.2.1 Evolution and Use of Multiple Ontologies and RDF KBs

Learners should be able to create and use different conceptualizations (ontologies and related instances) to describe the underlying domain; similarly, they should be able to describe the domain from different viewpoints and under different perspectives. This implies that the learners should not be in any way restricted to a predefined set of ontologies, but should have the ability to develop their own. Similarly, it should be possible to easily switch between changing the schema of an ontology and changing the data classified under the ontology schema.

The ability to change such ontologies and instances (called in short RDF KBs) should be provided in an integrated way by the system. This integrated functionality is based on the idea that the need to extend or change a KB arises when it is used. For example, it might become obvious that an aspect of the phenomenon to be modelled cannot be classified properly or it might appear that relations relevant for the task at hand cannot be modelled.

In this context, a learner or group of learners should have the ability to adapt given KBs to the particular needs of the activities they are involved in. This adaptation includes the evolution of both the ontology schema and the classified instances. Even

though ontologies by definition provide shared conceptualizations for a domain of interest [Gru93], they also provide the means to carry out activities and hence need to be adapted to local practices and task requirements. For example, a learner might decide that a given ontology does not provide the necessary concepts for the task at hand, and hence might want to extend it. While stable and widely accepted ontologies are useful from a technical point of view, locally adapted and adaptable ontologies seem to be more apt to the needs of trialogical learning. Furthermore, the local adaptation of the so created RDF KBs also allows creating different perspectives on a shared object of activity, which might help to get a better understanding of the phenomenon at hand.

The above requirements raise a number of needs, including some peripheral ones. First, the use of multiple RDF KBs raises certain accessibility issues, as KBs should be easily accessible by the learners. Thus, simple storage is not enough and we need to provide means to describe the stored conceptualizations; this is done through the use of some registry which stores metadata describing the ontologies represented in an RDF KB. Such metadata would help in the classification of ontologies, would simplify accessibility and would allow keeping track of an ontology's lifecycle in the KB.

The updatability requirement is mainly supported through the provision of a service that would effectively support changes in the ontologies and the related instances hosted by a KB. Such changes should be supported automatically and transparently by the system, so that the learner does not have to deal with the technicalities and side-effects of any single change upon his KB; it should be enough for him to indicate the required changes in a declarative way and let the KB do the rest. As conceptualizations change over time, different versions of a KB may need to be stored and made persistent, so a service should be in place that would keep track of such versions and their relationships. Learners should be notified for certain types of changes that are of interest either in the registry or in the KBs themselves. In other cases, it would make sense for a learner to compare the old version of a KB with the new one in order to see the newly submitted changes.

One of the central properties of trialogical learning, which is also present in the scenarios described in the previous subsection, is the element of collaboration. Collaboration implies that different professional experiences, different social and cultural backgrounds, participants' individual interests and goals, as well as inherent business rules and practices (including tacit ones) may cause misconceptions and frustrating ambiguities and misunderstandings [DLM07]. To smoothen the effects of such differences, the shared background of the collaborating group (partners) should be continuously negotiated until common concepts, characteristics and values have been agreed upon. In this respect, ontologies and RDF KBs, being shared conceptualizations of the domain under discussion [Gru93], are useful in this process, as they provide the means to describe shared resources of semantics [DLM07].

The above requirements, which arise from the need for the Knowledge Manager to support such collaborative activities and collaborative semantic modelling [CSMWK], imply that RDF KBs may have to be viewable, accessible and updatable by learners. View and access is necessary in order for a learner to grasp the understanding of other learners regarding the domain at hand, whereas updatability is

necessary in order learner to be able to provide their own arguments and feedback regarding a domain of discourse.

2.2.2 Semantic Annotation of Artefacts

Collaborative work with knowledge artefacts requires proper organization and structuring the artefacts according to their content (i.e., their meaning in the context of other artefacts), expressed by means of semantic annotation. The task of semantic annotation of an artefact can be defined as a selection of the concepts from a given ontology, that represent the content of the artefact. In other words, it can be considered as a classification of artefacts under the schema of an ontology, according to the textual content of the artefacts.

Selection of proper ontology concepts for description of an artefact can be a challenging task, especially if the set of artefacts is large and/or the domain ontology is complex. In addition, the learners need to deal with several different ontologies (conceptualisations) that were created as models of the underlying domain from different perspectives. Moreover, the ontologies can evolve in time, when the learners need to adapt given ontologies to the particular needs of the activities they are involved in. In this case, the semantic description of artefacts should also be updated according to the changes in the underlying ontology to keep the structure of conceptual model and annotated artefacts consistent, valid, and up-to-date.

The described semantic annotation of artefacts in the collaborative environment can be solved by means of text mining capabilities. This approach uses a machine learning technique to create internal mining objects (e.g. classification model, indexes and settings) from a set of already annotated (i.e., classified) artefacts. This means that a training set of artefacts (i.e., their textual content) classified to pre-defined categories (i.e., concepts from classification ontology) is required as an input for this approach. To create the mining objects properly, the global settings as a mining algorithm and its parameters need to be specified. Since the setting-up of proper algorithms and parameters is a specific feature of the mining approach, it is required that this should be hidden from users. The provided solution should select the most adequate mining algorithms and its parameters automatically, according to the quantitative properties of the training set (as e.g., the number of artefacts, frequency and distribution of words in the textual content of artefacts, etc.).

After the mining objects are created, the classification procedure will use the mining objects (especially the classification model) to examine the textual content of the rest of artefacts (i.e., those that were not included into the training set). The set of classification categories will be given in the output and provided to users (learners) as a result of the classification procedure. However, since the text mining approach to classification uses heuristic algorithms, the precision and overall quality of the results can not be guaranteed. So only a semi-automatic usage of classification results is required by users. This means that the results of the classification will not be automatically included in the semantic description, but will be provided for learners as suggestions for the annotation.

2.2.3 Semi-automatic Building of Ontologies and Clustering of Artefacts

Manual creation of concept maps from scratch presents a tedious work. Moreover, it is often the case that authors forget to enter a concept or a relation that can be crucial for the particular domain in question. To cope with these issues, the KP-Lab system should offer services that will help to identify the most relevant concepts and relations for a particular domain.

The basic functional requirement in this respect is to identify concept candidates from a defined set of documents (the textual content or the description of knowledge artefacts). Especially for the collaborative creation of ontologies by learners, an advanced function should extract defining contexts (definitions, if they are present in the texts). As it is expected that users will interact with the tool (invoking the particular service) and choose appropriate terms representing the concepts, the candidate list should be sorted according to the estimated relevancy for the domain.

Another step in the supported building of ontologies is to identify the most significant relations in which the chosen concepts participate. Given a subset of the concepts returned in the previous step, the system should analyse the input documents and find relevant relation candidates. Browsing the resulting list of potential relations and choosing the correct ones is necessary in this case so that the list of relation candidates needs to be sorted according to the estimated relevancy for the domain again. If possible, the system should also suggest names for the extracted relations and identify the most frequent classes such as “is-a”, “part-of” etc.

Some users may prefer less interactive way of building ontologies. Providing there is enough data for the task, the system should offer a fully automatic creation of a concept map that covers the most significant terms and relations among them. The result should be provided as a named graph and stored into the Knowledge Repository for further use.

The above description of the functional requirements supposes creation of a new ontology from scratch. However, in many cases, there is an existing ontology that covers a part of the domain and the task is to extend or update it to embrace the entire field. Thus, the above-mentioned functions should take into account the possible pre-existing knowledge and adjust their results accordingly. As such an ontology can be a result of other activities in the KP-Lab project, it is expected to be stored in the Knowledge Repository in a standard form.

In addition to other modes, KP-Lab tools should support an asynchronous way of learning in which one user, e.g, a lecturer, collects and pre-processes a set of relevant materials first and other users, e.g., students, work with the prepared set later on. For ontology creation, this mode means separation of the initial data collection and pre-processing from the actual extraction of concept/relation candidates or the automatic creation of concept maps. Dividing the task to the two phases can be advantageous also from efficiency point of view – the time necessary to process a potentially large amount of text can be considerably high.

The users of the KP-Lab system are often confronted with the task to look through a lot of texts, e.g., contributions to a discussion group, and group them according to their content. This tedious work should be supported by an automatic clustering

service that will take a set of artefacts as its input and groups them based on their textual content or description.

2.2.4 Keeping Users Aware of Changes

User notification constitutes one of the key elements to the development of large scale data retrieval and dissemination systems. The notification services allow the users to register their topics of interest in the form of subscriptions and inform them whenever an event that affects the content of the application matches their subscriptions.
From a general point of view, to function, this kind of service needs 2 types of information about the users and the application content. The first one corresponds to descriptions of the data present in the application. The second corresponds to the topics of interest of the users or their subscriptions.

The notification module is "triggered" with each data update (insertion, modification and/or removal). Through a comparison of the description of the updated data and the users' subscriptions, it determines the set of the users to notify about the update. The final action is to the users or user level applications previously identified (those associated to the matched subscriptions).

In order to specify the context of the notification module, it is necessary to answer certain questions:

- Which data will be concerned with the notification?

- How the users will be notified?

- Which events will trigger the notification service?

In what follows we try to answer these question by describe the basic ingredients of the notification service based on the scenario of teachers training communities.

A - The data to notify about
For the Kp-lab project, several objects could be subject of notifications: the shared spaces, the knowledge artifact or the knowledge processes.
Indeed, all of them are concerned with updates made by certain users and these updates may interest other users.
At this stage of the project, we decided that the notification service will be interested only in the knowledge artifacts because they contain the data most likely to interest the users. However, it is possible to extend this work to the other objects later on.

B- How to notify users (The notification problem):

The notification module manages the various subscriptions of the users. When a knowledge artifact is being updated, the notification module receives the update event (including the document description) from the knowledge repository. In order to find the users who are interested in this knowledge artifact update, the Matching Module compares the description with the subscriptions of the users. The users interested in the update are those associated to at least one of the subscriptions which match the description of the knowledge artifact.

The notification problem is “matching” events to subscriptions. In other words, given an event, the problem is how to find efficiently all users that should be notified, and this under a high number of events and for a large number of subscriptions.

C - Events that fire the notification service

Once the subscription chosen, it is necessary to define the events which will use it for the notification. Indeed a user has the choice between 3 possible and nonexclusive events: the insertion of a knowledge artifact, the removal of a knowledge artifact and/or the modification of a knowledge artifact. A user do not choose to be notified about the update of a given knowledge artifact, but about all the knowledge artifacts having a description that matches at least one of the subscriptions of this user.

When he chooses a subscription, the user defines also the update event (insertion, suppression or modification) for which the notification module will check the matching of this subscription with the description of the updated knowledge artifact

2.2.5 Summary of the Requirements

The following table summarizes the above high-level functional requirements for evolution and use of ontologies, for the text mining tasks as well as user notification (see also [DoWA] and [CSMWK], where a variant of this table appeared):

Functionality	**Short description**	**What a particular SWKM service provides**
Browsing the set of available conceptualizations	Users are retrieving the available conceptualizations already stored in the system.	**Registry** allows users to browse the conceptualizations taking advantage of the metadata provided
Introducing a new conceptualization	Users are collaboratively creating a new conceptualization (a new ontology or RDF KB)	**ConceptMapCreation** can help to identify the most relevant concepts and relations among them **Import** provides the initial step to store the new conceptualization **Registry** adds metadata for easy access and manipulation **Subscription** enables users to be notified about manipulation with the conceptualization
Using/Retrieving a conceptualization	Users are retrieving and visualizing an already stored conceptualization	**Registry** facilitates access to the conceptualization in question by means of metadata **Export** provides the requested data in the appropriate format
Creating a new version of an existing conceptualization	Users are retrieving, changing and subsequently storing an already existing in the system conceptualization as a new version	**ConceptMapCreation** can help to update the conceptualization **Versioning** relates the updated conceptualization to previous versions **Registry** adds metadata for easy access and manipulation
Inserting/Updating/ Deleting an element of the conceptualization	Users are changing the conceptualization	**ChangeImpact** shows all the consequences of the manipulation step the user asked for **Update** makes actual changes **Registry** takes care of metadata for the modified conceptualization

Collecting and preparing materials for text mining	Users collect a set of documents and prepare data for semi-automatic concept map building	**Prepare4Mining** computes an internal representation to enable fast and easy use of the extracted concepts and relations
Clustering artefacts	Users are grouping knowledge artefacts according to their content	**Clustering** identifies groups of artefacts based on their textual content or metadata description
Training and setting-up the classification.	Users are creating a mining model, using a set of annotated artefacts.	**Learning Classification** processes the training set and provides the classification model
Using the classification for semantic annotation (tagging)	Users are classifying the artefacts to some pre-defined categories (i.e. ontology concepts).	**Classification** applies a previously trained classification model for a new set of knowledge artefacts.

3 Functional and Architectural Design

3.1 Knowledge Mediator

The Knowledge Mediator provides high-level registry, discovery and evolution services for knowledge artefacts. It essentially mediates access to and changes of knowledge artefacts by employing personal or group conceptualizations under the form of RDF/S ontologies and RDF KBs; such ontologies and RDF KBs are then manageable using the mediator's services, namely change, comparison, versioning and registry, which are described below.

3.1.1 Change Service

The *Change Service* is responsible for determining the actual changes that should occur on an ontology or the related instances in response to a change request. Recall that in an RDF KB ontologies are represented by RDF namespaces while their instances by RDF graphspaces. The actual changes are not always the same as the requested ones, as the original change request could lead to invalidities if performed straightforwardly. In short, given a change request, the change service attempts to apply it to the target name or graph space in a straightforward way; if this naïve application leads to an RDF KB that is meaningless, invalid or does not obey the RDF formation rules [KFAC07], then additional updates (called *side-effects*) are added to the original request to guarantee validity.

As an example, consider the removal of an ontology class shown in Figure 1. In that case, the removal of class B would render all associations of this class with neighbouring classes invalid. In such cases, the change service needs to determine additional change operations (side-effects) to be executed along with the original change request which would restore the validity of the KB. In our example, one such set of side-effects would be to remove all invalid associations. In addition, the implicit subsumption relation between A and C that exists (implicitly, as a consequence of the other subsumptions) in the original RDF KB, need not be lost, so it is reinstated in the result, this time in an explicit manner; this is another type of side-effect, which guarantees that only information relevant to the update is lost during the change.

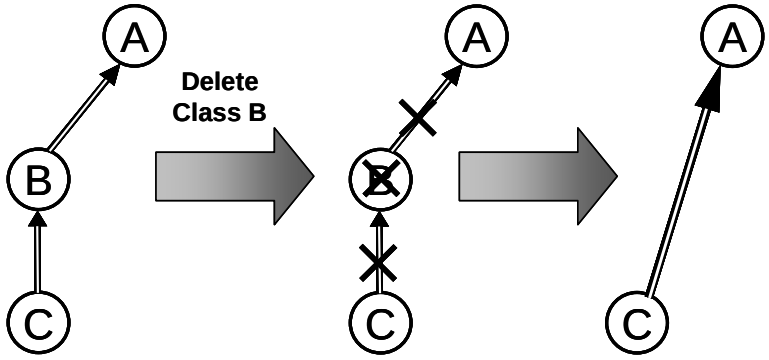


Figure 1: Change service - removal of a class

The main input to this service is an RDF KB and a change request. The RDF KB is specified using any, arbitrarily large, collection of name and/or graph spaces. The change request could affect any of the RDF triples in this collection. However, the side-effects of the request could potentially affect triples in other, depended or depending name or graph spaces; as a result, in order for the change request to be processed in a correct way, all the depended and depending name and graph spaces should be taken into account. Therefore, the RDF KB in this case is the union of all the triples that appear in all the name or graph spaces that are directly or indirectly depending on (or are dependants of) the given ones.

Having said that, the caller of the service is given the option to restrict the considered KB, as well as the changes and their side-effects to happen in the given collection of name or graph spaces, plus, of course, those name or graph spaces that the members of this collection depend on; it should be clear that this option may not give the best possible results, as certain side-effects may not be computed.

A simple update can be either a removal or an addition of a specific RDF triple in the RDF KB. Such simple updates can be arbitrarily combined in the same update request, to form a more complicated request; thus, in principle, an update request can be an arbitrarily large set of primitive additions and removals. For example, a simple update request would be “Remove Class B”, whereas a complex update request would be “Remove Class B; Remove A IsA C; Add property P with range A and domain C”.

The output of the service is of the same form, i.e., a set of change operations (additions and removals), capturing all the effects and side-effects of the original change request upon the target KB (actual changes). In the example of Figure 1, the output would contain the deletion of B (direct effect), the deletion of the two IsAs (side-effect) and the explicit addition of the previously implicit IsA (side-effect). These effects and side-effects are returned to the caller, in order to be visualized and either accepted or rejected.

The set of effects and side-effects that is produced in the output has been designed to satisfy certain properties. Firstly, the output update request should have no side-effects of its own, i.e., the straightforward application of the service’s output upon the original KB should always result to a valid KB. This is necessary in order for the output update request to be easily implementable without further post-processing.

Secondly, the original change request should be part of the output, i.e., no operation belonging to the input should be ignored. This is intuitively necessary, as the user wants his update request to be part of the actual changes executed. However, there are two exceptions to this rule. The first is technical and related to the operations of the input change that encode void requests (e.g., a request to add a triple that is already present in the KB); as far as the output change is concerned, it makes no difference whether such void requests will be included or not, so, for efficiency reasons, the resulting set of effects and side-effects is filtered out. Secondly, it could be the case that a change request is *infeasible*, i.e., that the operation is such that it is not possible to implement it without rendering the KB invalid, regardless of what side-effects we choose to use; in such cases, the update request is rejected in its entirety (an exception is returned by the service). An example of an infeasible operation would be "Remove Class B; Add an IsA between A and B"; such an operation is infeasible, because the addition of the IsA presupposes the existence of class B, so the operation of removing class B cannot be executed together with the addition of the IsA.

Notice that, in many cases, there may be more than one possible actual changes (i.e., side-effects) that satisfy the above properties; in such cases, the service will select the action that has the minimal possible impact upon the original RDF KB, without negating its validity. In other words, the result of the change should be "as close as possible" to the original KB, according to the "Principle of Minimal Change" [Gar92], i.e., the actual change should have the "mildest" possible effects and side-effects upon the original KB. One possible manifestation of this principle can be found in Figure 1, in which case it caused the explicit addition of the subsumption relation between A and C, to avoid unnecessary loss of information.

The impact of a change upon an RDF KB is measured by means of a preference ordering, which allows the service to determine the most plausible out of the different options for side-effects that restore the KB's validity (i.e., the one with the minimal impact) by comparing the impact of different sorts of update operations (side-effects) upon the RDF KB. Therefore, this preference ordering is a critical parameter that affects the determination of the actual change, thus implicitly allowing us to fine-tune the behaviour of the service (i.e., the returned side-effects). One such preference ordering is currently built-in into the current implementation of the service, but its modular design allows for alternative preference ordering can be used in the place of default one..

As already mentioned, an update request can contain any number of simple operations (additions or removals of triples). It should be emphasized that there is no particular order of execution of these simple updates, i.e., the entire update request is treated as a whole, in a transactional and deterministic manner, and, while searching for the minimal impact of such an update request, we consider the impact of the entire request, rather than the impact of each change operation separately. Notice that the selected (minimal) set of side-effects computed in this manner may be different from the one we would get if we processed each update operation separately.

In order for the system to guarantee the described behaviour in a consistent and deterministic manner, the service implementation is backed up by a formal theory which is described in detail in [KFAC07]. Based on this theory we have developed a

general-purpose algorithm that has been proved to exhibit the described behaviour for any kind of update request (simple or complex).

This general-purpose algorithm is backed up by a set of special-purpose algorithms which calculate the proper effects and side-effects for simple operations only; this way, we are able to provide faster, special-purpose implementations of our general-purpose algorithm, which are applicable only for simple update requests (thus trading generality for performance). The special-purpose algorithms exhibit the same behaviour as the general-purpose one, but are no substitute for it; recall that there is an infinite number of possible update requests, so this effort is inherently incomplete, and we will necessarily have to resort to the general-purpose algorithm for certain update requests. The process of selecting the proper algorithm (special-purpose or general-purpose) to use for a particular update request is transparent to the user: the service determines whether the given update request is supported by a special-purpose algorithm and adapts the execution sequence accordingly.

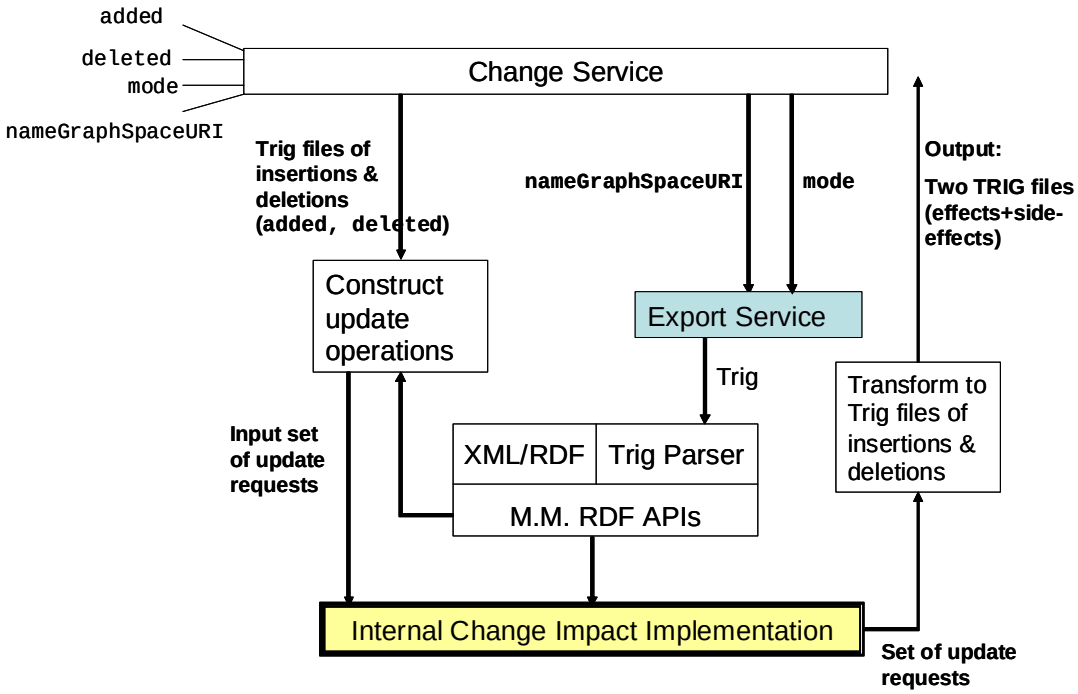


Figure 2: The high-level view of the Change Service

Figure 2 shows the general architecture of the web service. As shown in the figure, the change service exposes a single service which is used to apply an update request upon an RDF KB. The signature of the method is as follows:

```
String[] changeImpact(String added, String deleted,
  String[] nameGraphSpaceURI, String mode)
```

The output of the above method is a pair of strings; the first string represents the RDF triples that should be added to the KB, whereas the second represents the RDF triples that should be removed from the RDF KB. Both strings should encode the triples in TRIG format. As already mentioned, these triples include both the effects that were directly dictated by the original update request, and the ones dictated by validity

considerations, i.e., the side-effects used to avoid introducing invalidities in the original RDF KB due to the update request. Void additions and removals have been filtered from the output.

The input of the method is the update request and the RDF KB upon which the update should be applied, as well as a flag (*mode*) indicating the mode of the change. The *nameGraphSpaceURI[]* parameter is an array of strings, each string representing the URI of a name or graph space. Depending on the *mode* parameter, the update request will be applied either upon the union of the triples in those URIs and those that these URIs depend on, or upon the union of the triples in all name or graph spaces that are directly or indirectly depended or depending upon the URIs in the *nameGraphSpaceURI[]* parameter (i.e., their full dependency closure). These parameters are passed to the Export Service in order to get the exact triples that the implementation of the Change Impact Service will take into account in order to calculate the result of the change operation and are parsed to produce the necessary data structures to be used in the rest of the implementation.

The update request is specified using the string parameters *added* and *deleted*, representing the set of triples that should be added and deleted respectively from the RDF KB (i.e., the original update request). The triples should be encoded using TRIG syntax. The added and deleted triples are combined with the parsed output of the Export Service in order to determine the types of update operations that need to be executed upon the RDF KB and are ultimately fed, along with the RDF KB that was produced by the parsed output of the Export Service, to the Internal Change Impact Implementation to produce the output. A related restriction is that all the schema resources (classes, properties) that are used inside the *added* and *deleted* parameters (i.e., all the schema resources that appear in the update request) should have the same URI (including version ID – see the versioning service below) as (one of) the URI(s) of the input describing the RDF KB (i.e., one of the URIs in the *nameGraphSpaceURI[]* parameter); in a different case, an error is reported by the service.

3.1.2 Comparison Service

The *Comparison Service* is responsible for comparing two collections of name or graph spaces (KBs) already stored in the repository and compute their delta in an appropriate form. The result of the comparison is a “delta” (or “diff”) describing the differences between the two collections of name or graph spaces, i.e., the change(s) that should be applied upon the first in order to get to the second (see Figure 3 for an example). The intended use of the service is the comparison of two different versions of the same name or graph space to identify their differences; comparing unrelated name or graph spaces (i.e., name or graph spaces which are not different versions of the same name or graph space) would give results which have no intuitive meaning.

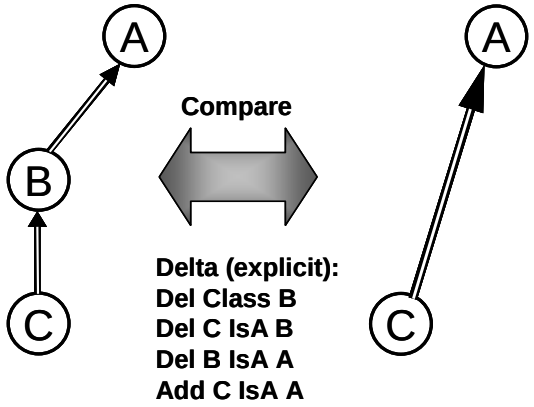


Figure 3: Comparing two name spaces

This problem is related to the problem of evolution that is handled by the Change Service; in the case of the Change Service, we know the original conceptualization and the changes that occurred, and want to determine the most adequate new conceptualization of the domain; in the case of the Comparison Service, we know the old and the new conceptualization of the domain, but lack the knowledge (control or access) of what caused the transition (i.e., we would like to determine what forced us to change our conceptualization).

Notice that the problem of comparing two name or graph spaces is very different from the problem of comparing the source files (e.g., TRIG files) which describe them. This is true because (a) a name (or graph) space carries semantics, as well as implicit knowledge which is not part of the source file; (b) there are alternative ways to describe syntactically the same construct (triple) in a name or graph space, which could result to erroneous differences if resorting to a source file comparison method; and (c) source files may contain irrelevant information, e.g., comments, which should be ignored during the comparison.

It is clear by the above analysis that the comparison should be based on semantic, rather than syntactic considerations, so our comparison service will be based on the comparison of the triples contained in the name or graph spaces. Our research has shown that there are alternative methods for computing a semantic delta between name or graph spaces [ZTC07]. In particular, the implicit knowledge (i.e., the inferred triples) contained in the two name or graph spaces may or may not be taken into account, leading to the following four cases:

- **Delta Explicit (Δ_e):** Takes into account only explicit triples
  - $\Delta_e(K \rightarrow K') = \{Add(t) \mid t \in K' - K\} \cup \{Del(t) \mid t \in K - K'\}$
- **Delta Closure (Δ_c):** Takes also into account inferred triples
  - $\Delta_c(K \rightarrow K') = \{Add(t) \mid t \in C(K') - C(K)\} \cup \{Del(t) \mid t \in C(K) - C(K')\}$
- **Delta Dense (Δ_d):** Returns the explicit triples of one KB that do not exist at the closure of the other KB
  - $\Delta_d(K \rightarrow K') = \{Add(t) \mid t \in K' - C(K)\} \cup \{Del(t) \mid t \in K - C(K')\}$
- **Delta Dense & Closure (Δ_{dc}):** resembles Δ_d regarding additions and Δ_c regarding deletions
  - $\Delta_{dc}(K \rightarrow K') = \{Add(t) \mid t \in K' - C(K)\} \cup \{Del(t) \mid t \in C(K) - C(K')\}$

In the above bullets the operator C(.) stands for the consequence operator, which is a function producing all the consequences (implications) of a name or graph space K, i.e., all the inferred triples of K. In the example in Figure 3, only the explicit knowledge is taken into account in the comparison, so the shown result corresponds to Δ_e. If the implicit knowledge was also taken into account, the result would be different (e.g., Δ_c, Δ_d and Δ_{dc}, would not report the addition of the [C IsA A] triple).

One of the main properties that we intuitively expect to hold in a comparison function is that its output, when applied upon the first name or graph space, should give the second; this property is called *correctness*. In order to study which of the four delta functions guarantees correctness, we should first determine what it means for the output of the diff service to be "applied" upon the first name or graph space. The latter issue is related to the semantics of the update operations considered, i.e., a formal description of how the output of the diff should be "applied" upon the name or graph space.

There are three options in this respect, namely: (a) that the operations (additions and deletions of triples) that are included in the delta are viewed as plain set additions and deletions (plain semantics – U_p); (b) that they are coupled with redundancy elimination and computation of logical implications (inference and reduction semantics – U_{ir}); or (c) that they are handled using the change semantics introduced by the Change Service (change service semantics – U_{cs}).

Using this definition of update semantics, in [ZTC07] it was shown that only certain pairs of delta functions with update semantics are correct, namely: (Δ_e,U_p), (Δ_{dc},U_{ir}) and (Δ_c,U_{ir}). Most existing comparison tools rely on the (Δ_e,U_p) pair. If we consider the update semantics U_{cs}, then the Δ_c function guarantees correctness, so (Δ_c,U_{cs}) is also correct. Based on this result, we can guarantee that the output of the Comparison Service is compatible with the Change Service, i.e., that the output of the Comparison Service (under the Δ_c function) is a set of primitive update operations which, if applied (using the Change or Update Service) to the first name or graph space, would result to the second one.

Another critical consideration is related to the size of the delta; in this respect, delta dense (Δ_d) is best, compared to any other delta function, whereas Δ_{dc} gives smaller in size delta than Δ_c; on the other hand, Δ_c and Δ_e are incomparable. Notice however that, as we saw above, Δ_d (the smallest possible delta) does not guarantee correctness.

For the purposes of the KP-Lab project, we don't adopt any particular policy regarding the "correct" or "best" delta function; in particular, the delta function to be used is just a parameter of the service, and the caller is assumed to understand the implications of using any particular delta function.

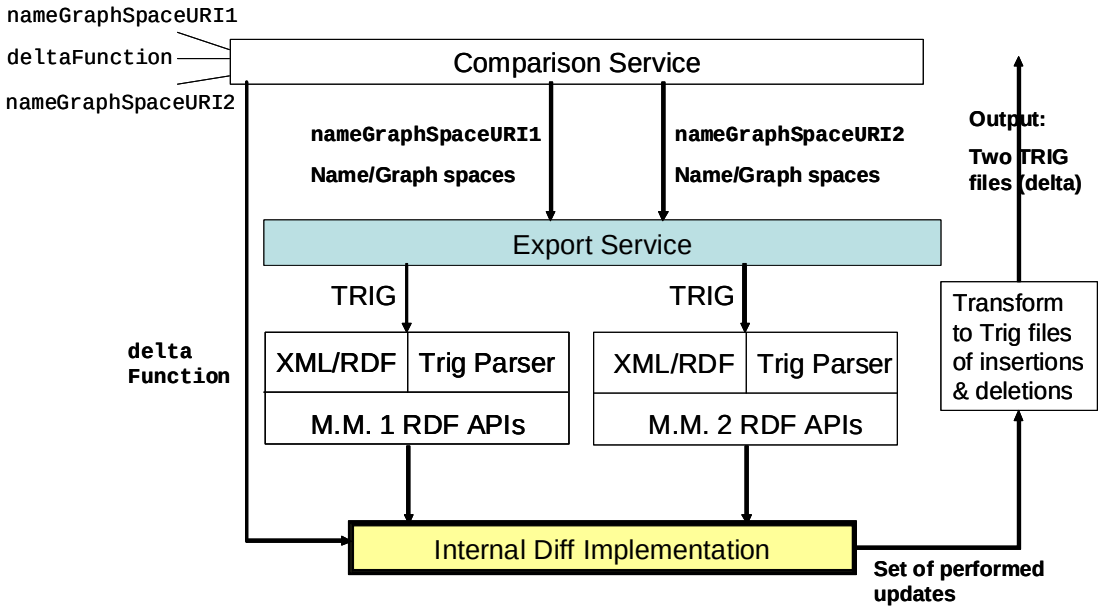


Figure 4: The Comparison Service

Figure 4 shows the general architecture of the web service of diff. As shown in the figure, the Comparison Service exposes a single service which is used to compare two collections of name or graph spaces and return their delta (diff) according to the selected delta function. The signature of the method is as follows:

```
String[] diff(String[] nameGraphSpaceURI1, String[]
  nameGraphSpaceURI2, String deltaFunction)
```

The output of the above method is a pair of strings representing the delta of the two models. In particular, the first string of the pair represents the RDF triples that exist in the second model but don't exist in the first, whereas the second represents the triples that exist in the first but not in the second. This way, the delta can be viewed as an update request (see also the Change Service above), which, when applied to the first model, will (should) result to the second; under this viewpoint, the first string of the output can be viewed as the added triples, while the second can be viewed as the deleted triples. Both strings encode those triples in TRIG format.

The input of the method is the two collections of the name or graph spaces to be compared, as well as a parameter indicating the mode of the comparison (delta function). These two collections are passed using the *`nameGraphSpaceURI1[]`* and *`nameGraphSpaceURI2[]`* parameters. Each such parameter is an array of strings, each string containing the URI of a name or graph space (so each of *`nameGraphSpaceURI1[]`* and *`nameGraphSpaceURI2[]`* represents a collection of name or graph spaces). It should be emphasized that the comparison is not performed upon the name and graph spaces in the input only, but also upon the name and graph spaces that they depend on. In other words, the compared conceptualizations occur by taking the union of the triples in the URIs indicated by *`nameGraphSpaceURI1[]`* (and *`nameGraphSpaceURI2[]`*) plus the triples in the name or graph spaces that the input name or graph spaces depend on. This is implemented through two independent calls to the Export Service (one for each of the

compared collections), followed by the parsing of the results to produce the related data structures used by the Internal Diff Implementation.

The `deltaFunction` parameter indicates the type of the delta function to be used in the comparison. In the current implementation, possible values for the `deltaFunction` parameter are: "D1", indicating that Delta Dense (Δ_d) should be used; "D2", indicating that Delta Closure (Δ_c) should be used; "D3", indicating that Delta Explicit (Δ_e) should be used; and "D4", indicating that Delta Dense & Closure (Δ_{dc}) should be used. The information on the delta function to be used, along with the parsed output of the Export Service are then fed into the Internal Diff Implementation to produce the output (diff) of the service.

3.1.3 Versioning Service

The *Versioning Service* is responsible for constructing a new persistent version of a name or graph space already stored in the repository, in effect allowing the creation of several versions of an ontology or their instances in an RDF KB, while keeping the logical relationships between each of its versions, i.e., which version was created as an evolution of which pre-existing one etc.

The initial functionality of the Versioning service will offer versioning at the level of single RDF name or graph spaces. To this end, it takes as input the information regarding the version's URI, the parent versions' URI and the contents of the new version and creates a persistent version of the name or graph space in the given URI, with a new version ID.

More specifically, the URI of a version is assumed to be "split" in two appropriately delimited parts; the first part contains the URI prefix, which is shared between all different versions of the same name or graph space, while the second part contains the version ID that allows us to discriminate between the various versions. For example, the URI of version v1 or namespace ns1 would be "ns1~_~v1".

The version IDs are generated automatically by the service each time a new version is requested. The service guarantees that no two versions of the same name or graph space will get the same version ID. The user of the service relies on the use of the full URI to refer to the name or graph version, whereas the Registry Service offers the necessary functionality for accessing the different versions and querying their interrelationships, in a transparent way.

Figure 5 summarizes the functionality of the service. Initially, a new version identifier is created; this identifier will be associated with the new version. Moreover, the contents of the new version are validated before being fed to the Import Service (along with the new version ID), which will make the version persistent. During the import, the URIs of the various elements of a namespace need to be changed as they no longer correspond to the same elements as the old ones. As an example, consider a resource A that exists in version v1 of the namespace ns1; then its full name (fylly-qualified) will be "ns1~_~v1#A". Following the creation of the new version, say v2, the name of A will change to "ns1~_~v2#A". This renaming process is necessary because, if any particular triple appeared unchanged in both versions, we would end up having the same triple appearing in more than one namespaces, which is invalid.

As the uniqueness requirement is true only for the namespaces, the renaming process is performed only for namespace versions. Following the renaming, appropriate calls to the Registry Service guarantee that the new version is properly recorded in the registry; to this end, the information on the new version's parent(s) is necessary.

It should be emphasized that the creation of the new version does not remove the old version(s) from the repository. Since the old version's URI does not change, references to old versions, are still valid. Changes of references to old versions is under the responsibility of the programmers.

One of the requirements of the method is that the new version and its parents should have the same URI prefix, as they are assumed to be different versions of the same name or graph space. Therefore, the validity of the input URIs should be verified before making the new version persistent, and success of the validation (and the import) is a prerequisite for the new version to be recorded in the registry. If validation succeeds, the Registry Service is used to record the new version of the name or graph space. The final output of the service is a URI that includes the URI prefix and the version ID of the new version.

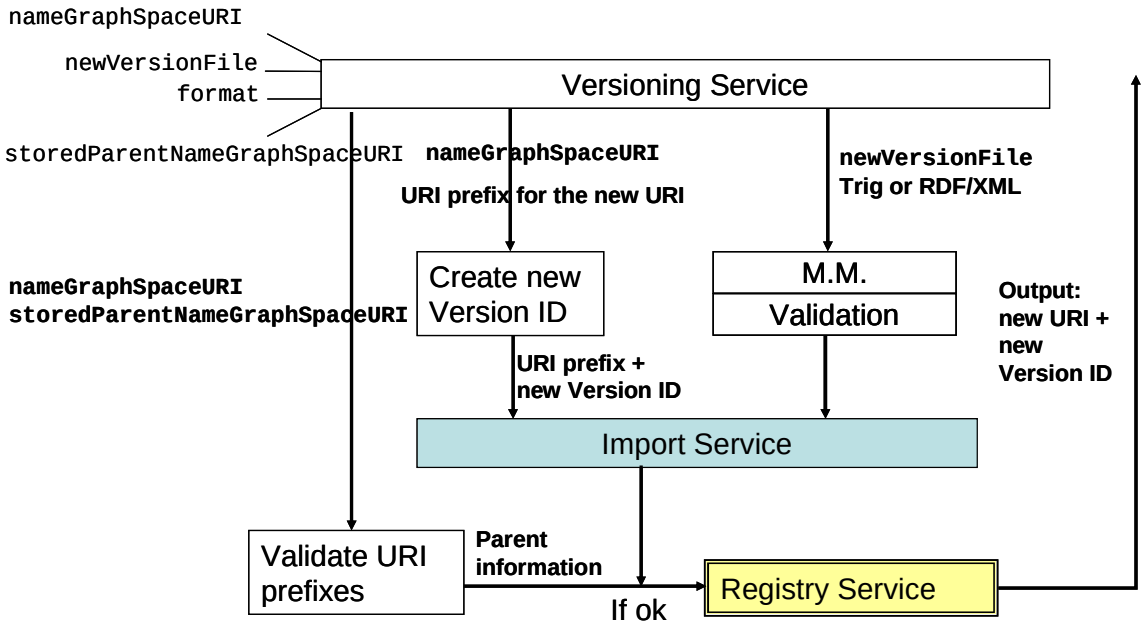


Figure 5: The Versioning Service

Programmatically, the versioning service exposes a single service for making a particular name or graph space persistent. The signature of the method is as follows:

```
String importVersion(String nameGraphSpaceURI, String[]
  storedParentNameGraphSpaceURI, String newVersionFile,
  String format)
```

The output of the above method is a string containing the full URI, which includes both the URI prefix (i.e., the common URI prefix that is shared among all the versions of this name or graph space) and the version ID of the new version. This URI could be

later used by the caller in order to get the contents of the new version, through a call to the Export Service.

The input consists of the *nameGraphSpaceURI* parameter, which is used to determine the URI prefix to be used in the new version's URI. The *storedParentNameGraphSpaceURI[]* parameter is an array of strings, each containing the URI of one of the parent(s) of the current version. If there is no previous version of the given name or graph space (i.e., if the currently created version is the first one), then there are no parents, so the array is empty. Notice that the URI prefix could also be determined using the parents' prefixes, but this approach would fail for versions with no parents (i.e., for new name or graph spaces).

The *newVersionFile* parameter contains a string describing all the triples of the new version of the name or graph space. These triples should be stored as the content of the new version. The format of the string in *newVersionFile* could be either TRIG or RDF/XML; the exact format is determined using the *format* parameter.

3.1.4 Registry Service

The role of the *Registry Service* is to record and manage metadata information about ontologies, schemas or namespaces stored in the knowledge repository. Furthermore, the registry offers the possibility to keep track of the development lifecycle of a schema through the support of storing versions, their metadata and the relationships among them. Both schema and version information follow the Ontology Registry Schema that is stored in the knowledge repository and is appropriately instantiated for each schema and version stored. Applications using the registry have the possibility to update and retrieve information about the already recorded schemas and their versions by using the available service methods. Notice that the Registry Service offers support for namespaces only; extending the service to also support graphspaces is rather straightforward and can be implemented if this is deemed necessary.

A comparison of some of the existing registries is presented in [DF01]. All of the mentioned systems provide certain searching facilities, but only some of them support editing functions that modify stored information about ontologies and add new ones (such as WebOnto [Dom98], Ontolingua [FFR96] and Ontology Server [ONTSRV]). Moreover, only a few provide reasoning mechanisms that make it possible to derive a query-answering mechanism such as WebOnto and Ontolingua. Furthermore, only one of the systems, SHOE [HHS99], supports a versioning mechanism in order to maintain the changes of ontologies in the registry. Our ontology registry provides all of the aforementioned functionalities, since it is using a query/update service based mechanism. Furthermore, it supports versioning in its more general sense as it will be described later.

The Registry Service is implemented as a web service and the different functionalities offered by it are implemented as web methods. However, this web service is not a self-contained module but rather depends on and uses the services provided by the knowledge repository, such as the Import, Update and Query Services. In particular, the Import Service is used to persistently store ontological descriptions, the Update Service is used to update the metadata information on the ontologies (which is stored in the Ontology Registry Schema, which is an ontology itself and described below)

and the Query Service is used to query the metadata information stored in the Ontology Registry Schema (for retrieval purposes). The dependencies between the Registry Service and the aforementioned services are schematically depicted in Figure 6.

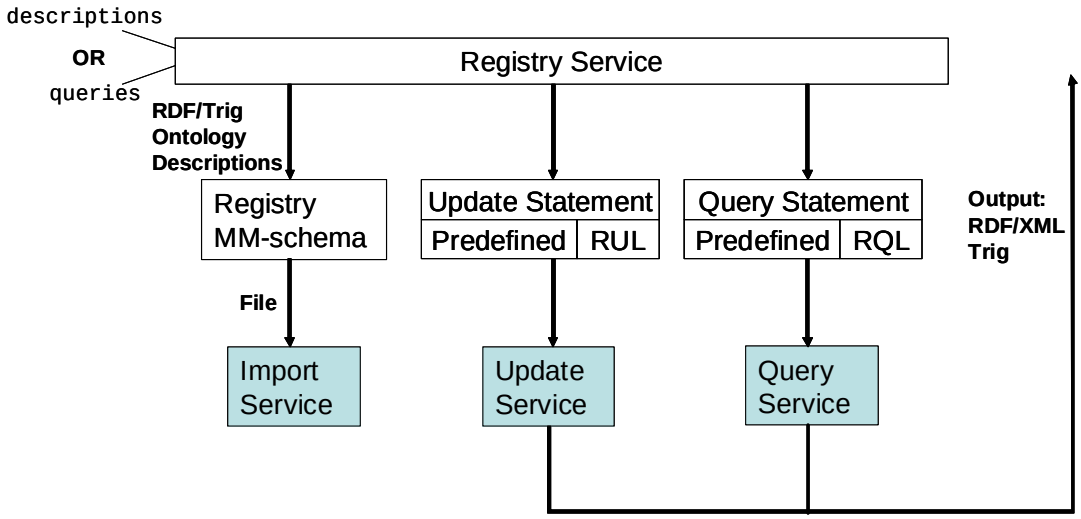


Figure 6: High-level view of the Registry Service

As already mentioned, the Registry Service is using its own ontology, encoded in RDF and following the RDF/S, in order to explicitly describe every other ontology stored in the Knowledge Repository. This ontology is called the *Ontology Registry Schema* and is described in detail later in this section (see Figure 8). For every ontology stored in the Knowledge Repository, an instance of the proper type is created and stored under the Ontology Registry Schema. The Registry is also supporting the recording of the versioning of schemas by allowing for each ontology the creation of multiple instances of the corresponding class `Version` and relating these instances to the proper instance of the class `Schema`. Thus, the metadata stored for each namespace are divided into two main categories regarding to whether their values are changing with each version (e.g., the number of classes or the related namespaces) or they are permanent characteristics of the namespace (e.g., the encoding or the URI prefix). This, in turn, imposes the rule that at least one version should exist in the Knowledge Repository for any stored namespace and its instance should be correctly related to the instance representing the namespace in the registry.

Since keeping track of versions has a significant role in the lifecycle of a schema, the registry supports a sophisticated versioning mechanism, accounting for and supporting the fact that different versions of a schema can be developed in parallel. Thus, during the lifecycle of a schema its versions can create a Direct Acyclic Graph (DAG). This means that a version might depend on more than one versions (like Version5 in Figure 7), which might be considered as merging two or more versions. Similarly, two or more versions might depend on a single one (like Version1 and Version2 in Figure 7), which might be considered as forking or parallel development. This way the maximum possible flexibility is provided and all known versioning schemes can be easily supported. The related information is provided by the Versioning Service. Apart from the versioning mechanism, the registry additionally offers the possibility to document the changes that occur on a schema when moving

from one version to the next one(s). These changes have the format of the results of the Comparison Service that compares two RDF models (see section 3.1.2).

Finally, as mentioned above, the Registry Service offers the possibility to retrieve ontology metadata information from the repository and also update the information that is already stored. In order to retrieve data from the registry, one can either type an RQL query, or use a query from a set of predefined ones. The latter type (the predefined queries) are exposed through a set of web service methods and are highly configurable by the developer of the service allowing for the necessary flexibility and taking advantage of the knowledge of the Ontology Registry Schema. Similarly, in order to update the information stored in the registry a set of implemented web methods is exposed accounting for most actions that might be needed by the user and assuring the necessary consistency of the information in the registry, imposing for example the rule of necessitating at least one version per schema; nevertheless, the user can always post updates in RUL, in which case (s)he bares also the responsibility for keeping the consistency rules.

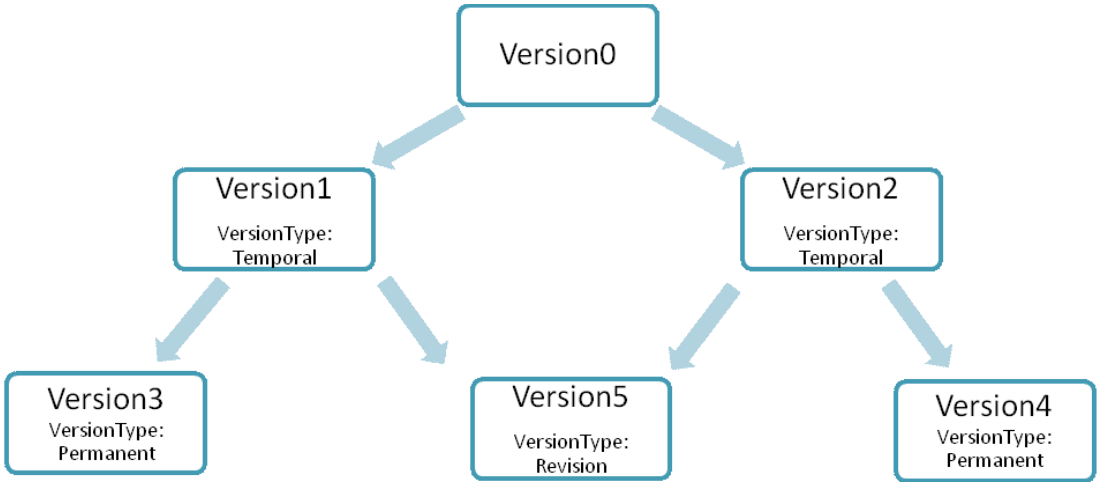


Figure 7: A possible DAG created by the versions of a schema. In this example, the first version is version0 and from it are emanating two versions, version1 and version2 that are developed in parallel. These two versions are merged by Version5. Version3 and Version4 are labelled as permanent versions, meaning that the authors do not plan to work anymore on them.

The schema of Ontology Registry consists of five basic classes: `Schema`, `Version`, `Change`, `foaf#Person` and `foaf#Organization`.

- The `Schema` class represents a stored namespace (or ontology or schema) and includes, besides the URI of the schema, information about the creator, the title, the purpose, the keywords etc. Regarding the organization of the concepts described by a namespace, the kind of their interrelations and the level of conceptualization, further classification is offered through the subclasses of class `Schema`. These subclasses are the following: `Ontology`, `Thesaurus`, `Taxonomy`, `SemanticNetwork`, `DomainOntology`, `UpperOntology`, `TaskOntology`, `CoreOntology`, `ApplicationOntology`, `FederatedThesaurus`, `FacetedThesaurus` and `NetworkedThesaurus`.

- The *Version* class is correlated to class *Schema* by the property *hasVersion* and describes attributes of a schema that might change between versions such as statistical characteristics of a schema (number of classes, number of properties, maximum length of a hierarchy). As one might see, this class also contains properties that correlate one schema to another with the relationships *import*, *extend* and *instanceOf*. Moreover, class *Version* has a property with predefined values that is used to indicate the intended uses of a version regarding its evolution during the version lifecycle. The predefined values are instances of *VersionType* class. The evolution can be seen in two ways: versions that are going to be developed in parallel and versions that are developed sequentially and depend on one another. Thus, the *VersionType* class can take the form of one of the following subclasses: *Permanent* (not to be merged in the future), *Temporal* (might be merged in the future) and *Revision* (replacing its previous versions). An example of the use of these values can be seen in Figure 7.
- The *Change* class is correlated with class *Version* through the property *changeRequest* and describes the insertions/deletions of RDF statements that have led to the creation of this version (in the form of add/delete statements like the ones produced by the Comparison Service).
- The *(FOAF#)Person* and *(FOAF#)Organization* classes from the schema *FOAF* are correlated to both classes *Schema* and *Version* through the properties *creator*, *publisher* and *contributor* respectively.

Moreover, some additional classes have been specified that are related to the language, encoding, and physical language used in the document describing a specific namespace. The main classes and properties of Ontology Registry Schema are illustrated in Figure 8. The recording of a schema namespace by the Registry Service might also include the storing into the registry not only of the instances of classes *Schema* and *Version* but also instances of the classes *Change*, *foaf#Person*, *foaf#Organization*, *Language*, *Encoding* and *PhysicalLanguage*.

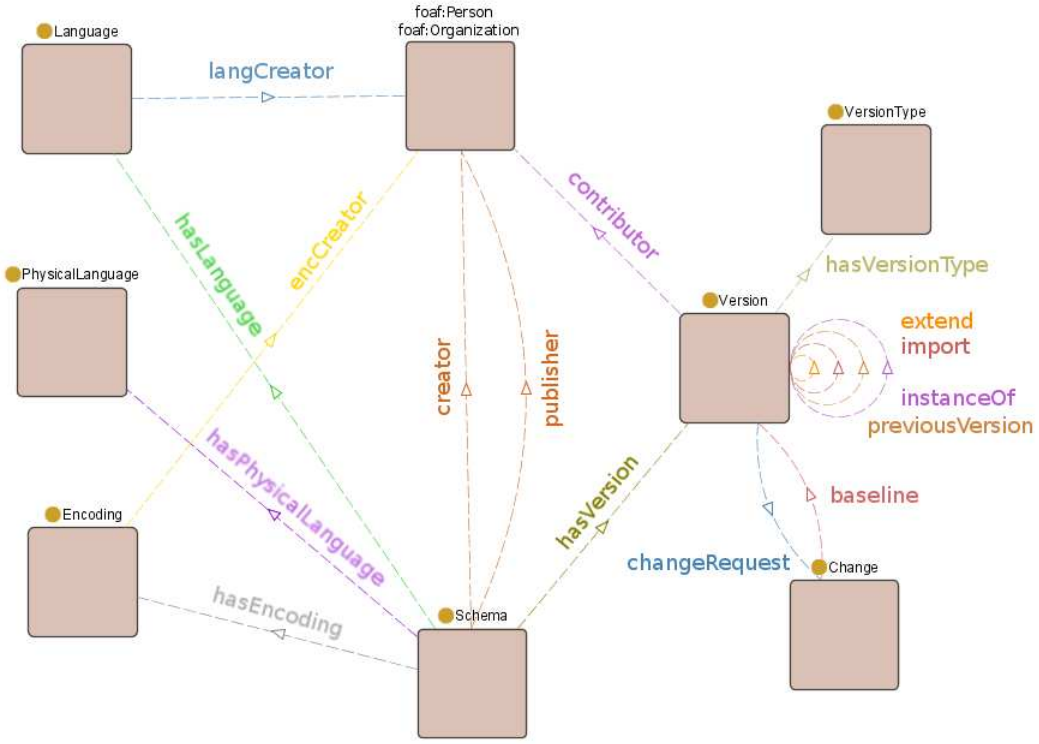


Figure 8: The Ontology Registry Schema

The Registry Service offers functionalities for:

- Storing information into the Ontology Registry Schema
- Updating information in the Ontology Registry Schema
- Retrieving information related to any object stored under the Ontology Registry Schema

As already mentioned, the methods exposed by the Registry Service are using the underlying methods offered by the Knowledge Repository, more specifically the Import, Update and Query Services. The Registry Service builds on top of these services in order to provide a more intuitive interface between the Knowledge Repository and the applications using the registry. These methods try to hide the possible complexity of producing the right (and optimized) RQL queries or RUL updates by predefining the correct ones, account for the consistency and imposing the necessary rules (which otherwise would have to be imposed manually) and exploit on the knowledge of the Ontology Registry Schema which the application need not know in detail.

So the available methods (web services) of the Ontology Registry API for inserting information into the Registry are:

```
void insertSchema(String schemaURI, String[] versionID,
  String file, Format format)
void    insertSchemaURI(String    className,    String
  instanceURI, String[] versionID)
void insertPerson(String classURI, String[] personURI,
  String[] property, String file, Format format)
```

```
void insertPersonURI(String classURI, String[]
  personURI, String[] property)
void insertOrganization(String classURI, String[]
  organizationURI, String[] property, String file,
  Format format)
void insertOrganizationURI(String classURI, String[]
  organizationURI, String[] property)
void insertInstance(String className, String str1,
  String[] str2, String file, Format format)
void insertInstanceURI(String className, String str1,
  String[] str2)
boolean existInstanceURI(ClassName className, String
  instanceURI)
```

The corresponding ones for updating information already stored in the Registry (including deletion of instances from the registry, update of the range properties with the constraint that the properties have literals as a range, etc.) are:

```
void removeInstance(String className, String
  instanceURI)
void editInstanceURI(String className, String oldURI,
  String newURI)
void insertProperty(ClassName className, String
  instanceURI, String propertyName, String[]
  propertyValue, PropertyRangeType rangeType)
void editProperty(ClassName className, String
  instanceURI, String propertyName, String oldValue,
  String newValue, PropertyRangeType rangeType)
```

Finally, the Registry Service uses the Query Service in order to retrieve data from the registry by evaluating RQL queries. The user can directly pose RQL queries through the `query()` method of the Query Service or use the method that is implemented by the Registry Service API, called:

```
String evaluatePredefinedQuery(String queryCategory,
  String queryID, String param, Format format)
```

that can be used when the application needs to use one of the predefined queries which are in turn dynamically specified by the service developer in an XML file.

3.2 Knowledge MatchMaker

The Knowledge Matchmaker module supports advanced manipulation of content items, namely *mining* and *notification* [DoWB]. It enables *concept map creation*, *clustering* and *classification* of the available information resources associated with employed ontologies; and also *notification* of changes to content items produced/consumed within a collaborating group of individual users or application programs, according to explicitly *subscribed* preferences.

3.2.1 Notification service

The objective of this service is to support individual (human) users as well as various tools or software components accessing the knowledge repository by keeping them aware of changes. This objective will be achieved by designing and implementing a *notification service* [D5.1]. In describing this service below, we use the term "users" to refer collectively both to individual users and to the various tools or software components accessing the knowledge repository.

The notification service as we conceive it relies on the following basic concepts:

- *The objects of interest*:
  In the KP-Lab project the objects of interest are content items of various kinds as far as they have a description.
- *The description of the objects of interest*:
  The description of an object of interest is composed of a set of RDF statements according to an already given RDF schema. (see Section 3.2.2).
- *The subscribers (or receivers) of notification*:
  Subscribers are those users that have submitted to the notification service a description of the content items that are of interest to them. Such a description is called a *subscription*. A user can be a "physical" person or another module of the project.
- *The subscriptions of the objects of interest*:
  A user subscription is of the same nature as the item description (i.e. a set of RDF statements.)
- *The events that fire the notification service*:
  In the KP-Lab project the knowledge repository can change in several ways:
  - insertion, deletion, or modification of a content item;
  - locking of a content item (for reading or writing purposes).

  We use the term "event" to refer to one such change together with one content item involved in the change; and we consider as "event description" the description of the corresponding content item.
- *The matching algorithm that supports the notification service*:
  This algorithm is invoked, or "triggered" by each event occurring at the knowledge repository and determines the set of subscribers to be notified of the occurrence of that event. Its basic function is to compare user subscriptions to the description of an event (based on an appropriately defined partial ordering structure) and to determine the set of subscribers to be notified of the event.

In simple terms, the basic principle of notification can be expressed as follows: for each subscription, if the subscription matches the description of the triggering event then notify all users having that subscription.

In fact, a user subscription can be seen as a conjunctive query expressing long term interests of a user for content items of a certain type – a query that the user would like to submit to the repository from time to time. The notification service on the other

hand can be seen as the functionality that does this in place of the user (so that the user does not have to submit the same query again and again), and informs the user only if the answer to the query has changed. Clearly, two or more users might have the same interests, hence the same subscription.

The basic problem of notification is how to determine efficiently the set of all users to be notified, under a high number of events and a large number of subscriptions. The matching algorithm that we have designed during the first year of the project will be implemented to answer this need.

In implementing the notification service, care will be taken so that transposing the algorithm in a different context will require minimal changes and effort. In other words, the idea is to provide an implementation as generic as possible.

Our implementation will be conducted under a number of assumptions, including the following:

1/ The form in which a subscriber receives notifications may differ from one user to another. For example, a human user might prefer to be notified via email whereas an application program will most likely be notified via RSS. The choice of a form of notification should therefore appear in the subscription. The first version of our prototype will simply produce the set of users to be notified, disregarding the form in which notification will be sent to the users concerned.

2/ User notification can be made in one of several ways:
- immediately after an event has occurred;
- after a fixed number of events have occurred (number to be specified in the subscription);
- periodically (periodicity to be specified in the subscription, e.g. weekly).
The user must indicate in the subscription in which way notification is to be done. In the KP-Lab project we shall implement the first approach (“immediate” notification). We note that the second and third approach require the storing of events until the next notification time.

3/ As all users do not have the same access rights on all content items, it is important to take into account access rights during notification. Indeed, it makes no sense to notify a user about a content item which the user cannot access. The first version of our prototype will not be concerned with access rights (i.e., every user has access rights to every content item).

The notification service is composed of 2 main web services (see figure 9):

(a) Subscription service: It is responsible for the following task:

Subscription update: It consists in registering a new subscription and unregistering or modifying an already registered subscription.

```
String RegisterSubscription(URI userId, RdfDocument
Subscription, String schemaURI, String EventType,
String[] notificationForm)
```
input:
- userId: the identifier of the user invoking this method
- Subscription: An RDF document representing the subscription of the user according to the the RDF schema specified by schemaURI.
- schemaURI : the RDF schema of the subscription
- EventType : the type of the event the user is interested in (insertion, deletion or modification of a document)
- notificationForm: the form by which the notifications will be delivered to the subscriber.

output: if the registration failed returns an error message, else returns ok.

Preconditions:
A user who is already registered (who has an identifier). The subscription is submitted as an RDF document according to a given RDF schema. The user has to specify the type of the event to which he wants to subscribe, as well as the form of the notification (for the first version of our prototype, only "RSS feeds" will be used for the notifications).

```
String ModifySubscription(URI userId,URI SubsURI,
RdfDocument NewSubscription, String schemaURI, String
EventType, String[] notificationForm)
```
input:
- userId: the identifier of the user invoking this method
- SubsURI : the URI of the old subscription (to be changed)
- NewSubscription: An RDF document representing the subscription of the user according to the the RDF schema specified by schemaURI.
- schemaURI : the RDF schema of the subscription
- EventType : the new type of the event the user is interested in (insertion, deletion or modification of a document)
- notificationForm: the form by which the notifications will be delivered to the subscriber.

output: if the registration failed returns an error message, else returns ok

Preconditions:
A user who is already registered (who has an identifier). The user has to specify only the parameters to be changed. For example if he wants to change only the subscription and keep the same event type and the same notification form, he only has to specify the new subscription.
The subscription is submitted as an RDF document according to a given RDF schema. The user has to specify the type of the event to which he wants to subscribe, as well as the form of the notification (for the first version of our prototype, only "RSS feeds" will be used for the notifications).

```
String UnregisterSubscription(URI userId, URI SubsURI)
```

input:
- userId: the identifier of the user invoking this method
- SubsURI: the identifier of the subscription the user wants to unregister.

output: if the unregistration failed returns an error message, else returns ok.

Preconditions :
A user who is already registered (who has an identifier).

(b) Notification propagation module:

This module delivers the notifications: After the matching process, this module sends the notifications to the module responsible for the delivery of notifications (an RSS aggregator for example).

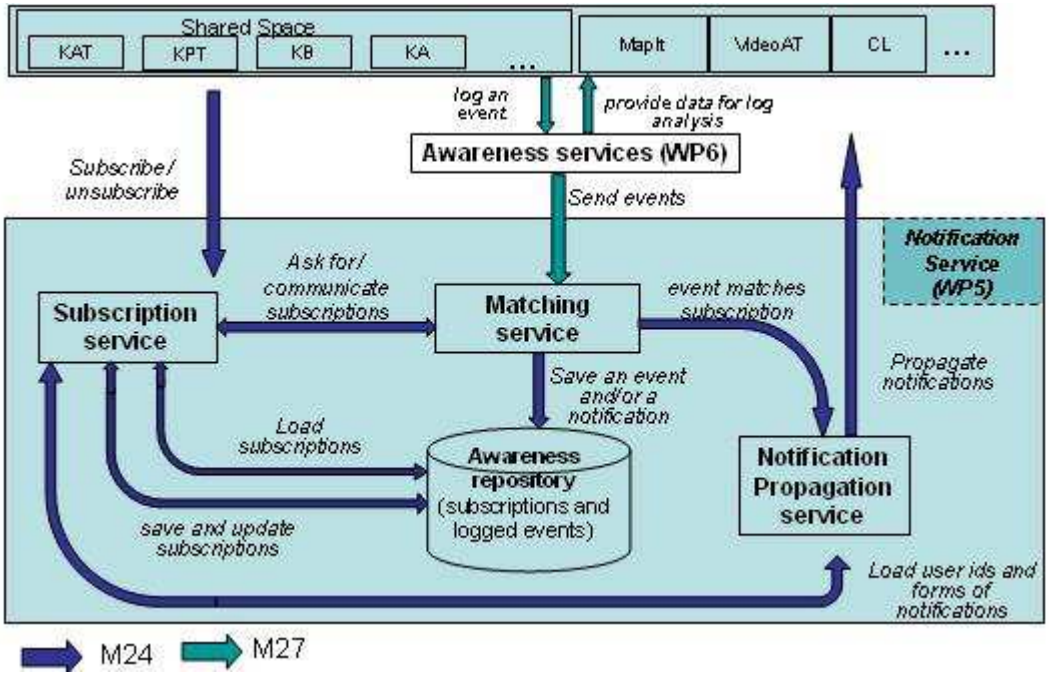
```
RSSfeed Propagate(URI DocumentId, RdfDocument Description,
String EventType)
```

input:
- DocumentId: the identifier of the document being added, modified or deleted.
- Description: An RDF document representing the desctiption of the document
- EventType : the type of the event (insertion, deletion or modification of a document)

output: the RSSstreams corresponding to the susbscriptions to be notified.

The notification service uses a repository called "**Awareness Repository**" to save the data needed to perform the notifications (see figure 9), it's main tasks are:

(a) Storage of the users subscriptions
(b) Storage of the events to be processed.



(c)

Figure 9. Notification service

3.2.2 Text Mining Services

Text mining and extraction services are designed to assist users in the process of creating or updating the semantic descriptions of KP-Lab knowledge artefacts. The semi-automatic generation of these descriptions or even of new KP-Lab ontologies relies on the textual information attached to particular artefacts as a textual content itself, or as a set of text-based metadata.

Although the knowledge artefacts can be stored in various forms (e.g., textual documents, conceptual maps, video sequences, images, etc.), they often contain textual information directly in its content, or indirectly in metadata or textual annotations given by users. The textual description is analysed using different text mining techniques. As a result of the text mining analysis, relevant concepts from the KP-Lab ontologies are suggested to the users during the formal description (i.e., annotation) of knowledge artefacts. Moreover, unsupervised text mining techniques – concept map creation and clustering – can be used to find some unseen concepts and relations in the set of analyzed textual resources and to group (cluster) the resources according to their content. These may lead to, e.g., the suggestion to upgrade existing KP-Lab ontologies, as the knowledge of a user group evolves.

The fundamental tasks for the envisioned text mining services are *concept map creation*, *clustering* and *classification* of knowledge artefacts. Classification groups a given set of artefacts into predefined or ad hoc categories. Concept map creation automatically extracts significant terms from textual resources and converts them to a structure of concepts and their relations. In addition, the derived text mining tasks, such as *keyword extraction / summarisation* and *information extraction*, can also be used by KP-Lab tools to create an initial dictionary for ontologies and to extract the values of various metadata properties.

The functionality and the algorithms used for the specified text mining tasks were already briefly outlined in [D5.1] and are described in more detail in the following sections.

3.2.2.1 Pre-processing of texts

Basic text mining tasks, i.e., text classification, clustering and concept map creation, need to manipulate textual documents in a specific form (e.g., the “bag of words” representation, vector space model, etc.). The pre-processing phase is responsible for transforming data into the appropriate form. It consists of several language-dependent NLP (natural language processing) steps that provide annotations of the plain-text resources.

For the purposes of concept map creation, clustering and classification of knowledge artefacts in the KP-Lab, we decided to employ unified modules for *tokenization* (splitting input text to individual tokens), *stemming* (or more sophisticated *lemmatization* in morphologically rich languages), *elimination of stop words*, and *POS* (part-of-speech) *tagging*. Other advanced NLP techniques such as chunking, WSD (word-sense disambiguation) or the full syntactic analysis are used by individual modules (e.g., they are crucial for some methods of concept map creation but not for the classification).

The pre-processing of texts is handled by GATE – General Architecture for Text Engineering [GATE]. GATE is an infrastructure for developing and deploying software components that process human language.
GATE helps in three ways:

1. by specifying an architecture, or organisational structure, for language processing software;
2. by providing a framework, or class library, which implements the architecture and can be used to embed language processing capabilities in diverse applications;
3. by providing a development environment built on top of the framework made up of convenient graphical tools for developing components.

The pre-processing component, which provides common functionality for concept map creation, clustering and classification tasks, is implemented as a pipeline of processing resources on top of the GATE engine. Additional language processing resources, that are necessary for the concept map creation service, integrate language-dependent tasks such as parsing, keyword extraction, co-occurrence statistics and semantic-distance computation. Figure 10 shows an example of NLP methods applied in the pre-processing step of the automatic concept map creation.

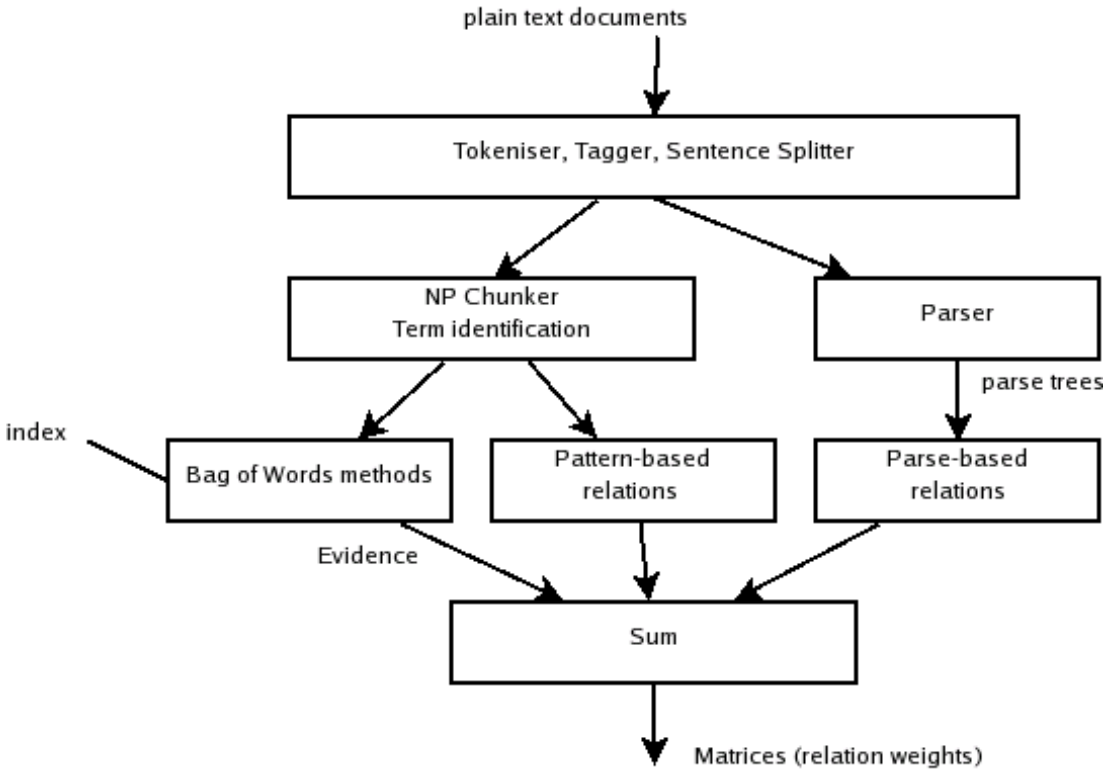


Figure 10. Pre-processing for the automatic concept map creation

To access the knowledge artefacts and their textual descriptions, we take advantage of the Gateways to the Knowledge Repository and Content Repository [D4.2.2]. The results of pre-processing (e.g., vector models of texts) as well as dictionaries and settings for NLP analysis methods are stored in the *Mining Object Repository* [D5.1]. This repository contains all the data of text mining services that requires permanent

storage; in addition to the mentioned data, there are also training sets and classification models, as well as external settings for classification, clustering and concept map creation services. Mining object URIs or *moURI*s are provided as the identifiers of the stored data.

Let us summarize the general part of the functionality implemented in the services for concept map creation, clustering, and classification (see the following sections for detailed schemata):

1. Retrieve the textual content and metadata of knowledge artefacts from the KP-Lab Knowledge Repository and Content Repository.
2. Extract the plain text from the retrieved data (which can be stored in various formats, encodings, etc.).
3. Apply the NLP analysis methods to process the input texts, e.g., parse the text into elementary words (tokens), eliminate the words of less impact on the text meaning (so-called stop-words, i.e., very frequent words, prepositions, etc.), eliminate the declination alternatives (by means of stemming, POS tagging, etc.), convert the text to a set of weighted terms. Weights correspond to the relative frequency of a particular word in the text and express its relevance or contribution to the overall text content.
4. Produce the weighted term-document matrix, which is the input for further processing.
5. Save the term-document matrix into the Mining Object Repository.
6. Return a mining object URI (moURI) of the data.

3.2.2.2 Clustering and Automatic Creation of Concept Maps

The clustering task enables finding clusters in an input set of artefacts (based on their textual content and/or metainformation). As opposed to classification, the clustering task does not require a training phase. The resulting clusters of artefacts are, in general, unnamed but they can be labelled, e.g., by the most common words in textual data. Unsupervised machine learning algorithms for partitional clustering are considered in the Knowledge MatchMaker, namely the K-means algorithm and its derivatives [MacQueen 1967].

The task of an automatic creation of concept maps identifies the most significant terms (representing concepts) and identifies relations among them. A set of artefacts provided by the user is processed first. The service can then identify concept candidates. The user can also specify a set of seed concepts and ask the service to find relation candidates, as well as the type of the relation. The full concept map can be generated in the form of a named graph.

The clustering and concept map services provide the following functionality:

- Pre-process documents (textual parts of the knowledge artefacts) by means of the methods described in section 3.2.2.1, produce an internal representation and store it into the Mining Object Repository. A moURI (mining object URI) is provided as an output, which can be subsequently used for accessing the data.
- Delete the pre-processed data from the Mining Object Repository which will not be needed any more.

- Identify concept candidates and rank them according to the estimated relevance, extract defining contexts for the terms
- Given a set of concepts, find related concepts from the documents provided by the user. Return a ranked list of candidate relations together with their types.
- Build the concept map, generate the named graph and store it to the Knowledge Repository.
- Find clusters in the specified set of documents (the set is given by artefactURIs)

To support the possible division of the user roles, namely the setting in which one user collects and pre-processes a set of relevant materials and others use the data to build own conceptualization later on, the concept map creation service defines two phases – the initial data collection and pre-processing and the actual extraction of concept/relation candidates or the automatic creation of concept maps.

According to the division, the concept map creation consist in the *Prepare4Mining service* (that should be invoked first) and the actual *ConceptMapCreation* service. Both services are implemented as web services and use the Mining Object Repository to store and retrieve mining objects.

The **Prepare4Mining service** exposes the following methods for creation, modification, and removal of mining objects:

```
String createMo( String[] settings,
                 String[] artefactURIs,
                 String namedGraphURI)
```

input:

settings: specification of mining parameters

artefactURIs: a training set, i.e. an array of URIs of semantically annotated artefacts (retrieved from the SWKM Knowledge Repository)

namedGraphURI: a seed conceptualization in the form of the named graph

output:

moURI: URI of the prepared mining object

```
void modifyMo (String moURI,
                 String[] settings,
                 String[] artefactURIs,
                 String namedGraphURI)
```

input:

moURI: URI of the mining object to be modified

settings: specification of mining parameters

artefactURIs: a training set, i.e. an array of URIs of semantically annotated artefacts (retrieved from the SWKM Knowledge Repository)

namedGraphURI: a seed conceptualization stored in the SWKM Knowledge Repository

```
void deleteMo(String moURI)
```
input:

moURI: URI of the mining object to be removed from the repository

The Prepare4Mining service is implemented as a web service and the different functionalities offered by it are implemented as web methods. Figure 11 shows internal procedures for creation of a mining object within the Prepare4Mining service. Blue boxes represent references to existing KP-Lab services, yellow boxes the newly designed SWKM services.

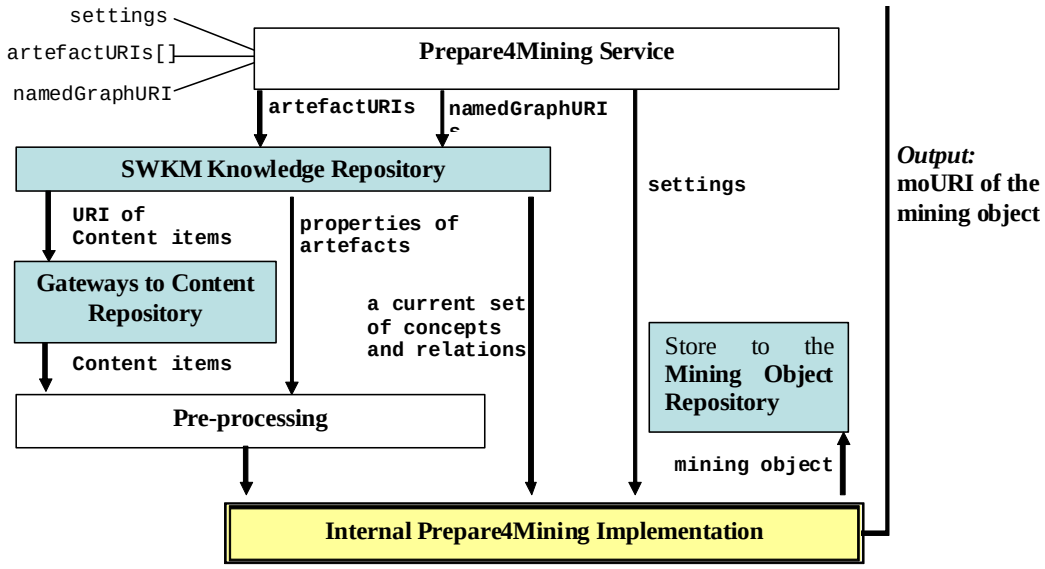


Figure 11: The Prepare4Mining Service

ConceptMapCreation service provides the following methods for (semi-)automatic building of concept maps:

```
String[] findConceptCandidates( String moURI,
                                String[] settings)
```
input:

moURI: URI of the mining object returned by the previous call of the Prepare4Mining service

settings: restrictions on the resulting list of concept candidates

output:

a ranked list of extracted concept candidates and extracted defining contexts. A score (0.0 – 1.0) is assigned to each candidate according to the estimated relevancy. A temporary moURI is generated for each concept candidate.

```
String[] findRelationCandidates( String moURI,
                                 String[] settings,
                                 String[] concepts)
```
input:

moURI: URI of the mining object returned by the previous call of the Prepare4Mining service
settings: restrictions on the resulting list of relation candidates
concepts: a set of moURIs of concepts from which the relations should lead

output:

a ranked list of most relevant relations; types of the relations (such as “is-a”, “part-of”, …) are also provided

`String buildConceptMap(String moURI, String[] settings)`

input:

moURI: URI of the mining object returned by the previous call of the Prepare4Mining service
settings: restrictions on the resulting concept map

output:

URI of the named graph representing the created concept map stored in the Knowledge Repository

The ConceptMapCreation Service is also implemented as a web service that exposes its functionality via the given web methods. Figure 12 shows the internal procedures of the ConceptMapCreation services, focusing on the buildConceptMap method.

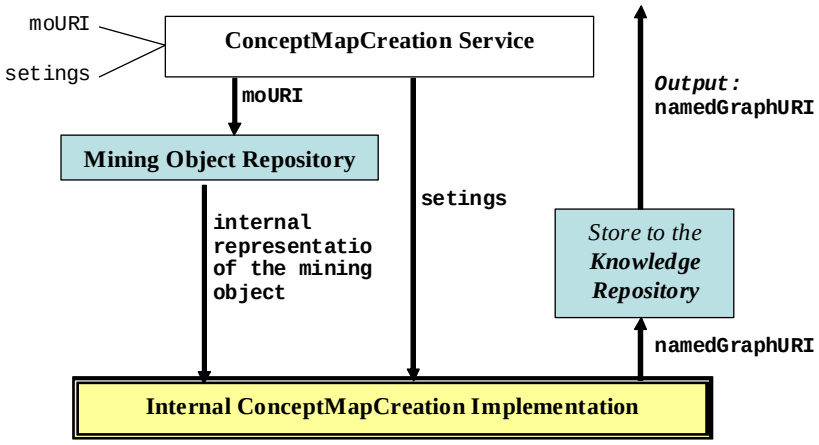


Figure 12: The ConceptMapCreation service

Clustering service provides the following method for clustering artefacts:

`String[] findClusters(String moURI, String[] settings)`

input:

moURI: URI of the mining object returned by the previous call of the Prepare4Mining service

settings: restrictions on the resulting clusters

output:

a set of sets of cluster labels that identify the clusters in the given set of artefacts based on their textual content

The Clustering Service is also implemented as a web service that exposes its functionality via the given web method. Figure 13 summarizes the internal procedure of the Clustering service.

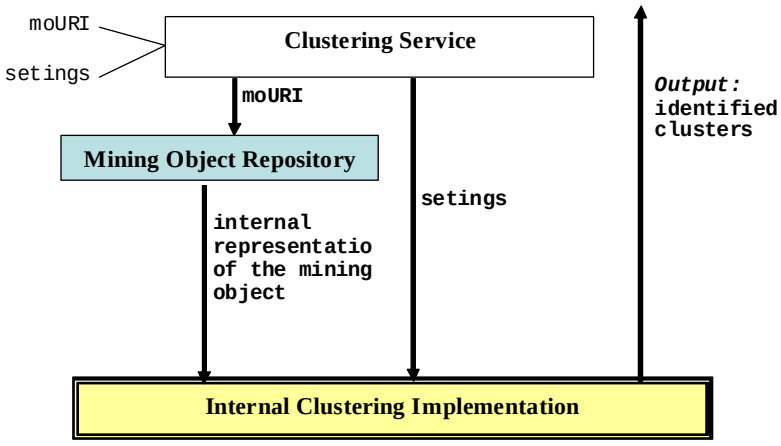


Figure 13: The Clustering service

The ConceptMapCreation and Clustering services themselves have no user interface for their functions. It is assumed that the services are invoked by other KP-Lab tools, e.g., by the Shared Space, and that their functionality will be used in the context of those tools.

3.2.2.3 Classification

The classification task is used in order to automatically organize a set of knowledge artefacts into predefined categories. The predefined categories are the concepts of an existing ontology, which are selected to semantically annotate the artefact. The ontology (or RDF KB) is supposed to be collaboratively created by learners within the Shared Space, possibly using the assistance of the Clustering and Concept Map Building services. Ontologies, as well as the knowledge artefacts (including their properties and annotations) are stored in the SWKM Knowledge Repository and are accessible by Knowledge Mediator services. The textual content of the artefacts can be retrieved from the Content Repository according to the URI of proper content item, stored as a property of the artefacts in the SWKM Knowledge Repository.

Classification is a supervised machine-learning method based on a training set of already semantically annotated artefacts. The internal *mining objects* are created from the annotations and textual descriptions of the artefacts included in the training set. The mining objects (sometimes also referenced as *classification model*) contain binary representation of term-document matrixes, text indexes, plain text extractions, and a set of parameters (weights, rules, etc. – based on the used algorithm) created in the process of training. The mining objects are used for the classification of unknown examples (artefacts).

The following algorithms are considered to be used for classification: simple term matching, kNN, SVM, Winnow, Perceptron, Naive Bayes (multinomial and binomial), boosting, decision rules, and decision trees (various combinations of growing and pruning methods) [Lewis 1998, Quinlan 1996, Yang 2001].

The classification service will be implemented as an extension of the JBowl library [Bednar et al 2005] and will provide the following functionality:

- Create a training data set from documents (knowledge artefacts containing a textual description) already categorised to a pre-defined set of categories. The textual descriptions of the documents are pre-processed (using the pre-processing methods described above) and transformed into a term-document matrix. The classification service indexes the training data set and stores it into the Mining Object Repository.
- Create classification objects, based on the selected algorithm and on a given training data set.
- Enable modifications (tuning) of the existing classification objects, by changing the texts and/or categories in the training data set, as well as by editing the settings of the algorithm or switching to another algorithm.
- Provide statistics on the existing classification objects, by means of standard measures as precision and recall. Enable to create, index, and store a separate testing data set (composed also from categorised documents) that can be used for more exact examination of the quality of the classification process.
- Provide verification and validation of the existing classification. The classification objects are no longer valid if a portion of training data set (e.g., the term-document matrix or the set of pre-defined categories) was modified. In this case, re-indexing of the objects is needed to make them valid again.
- Classify a set of unknown documents (knowledge artefacts) to the same categories that were used for training. The output of this function is a set of weighted categories for each of the classified documents.

Based on the outlined functionality, two phases of the classification can be specified:

1) *Creation and maintenance of classification objects*, based on a given training set of already classified documents (i.e. annotated artefacts).
2) Actual *classification* of unknown documents (artefacts).

According to this division, the classification service is composed of two main sub-services: *TrainClassifier service* and *Classify service*. Both sub-services are implemented as web services and use the Mining Object Repository to store and retrieve classification objects and settings needed to perform the classification.

TrainClassifier service exposes the following methods for creation, modification, and removal of classification objects:

`String moURI createClassifier(String settings, String[] artefactURIs)`
input:

settings: a specification of classification algorithm and its settings. This algorithm will be used for the creation of the classification object.
artefactURIs: a training set, i.e. an array of URIs of semantically annotated artefacts (retrieved from the SWKM Knowledge Repository).

output: moURI: URI of the created classification object.

`void modifyModel(String moURI, String[] settings, String[] artefactURIs)`
input:

moURI: URI of a classification model to be modified.
settings: a specification of classification algorithm and its settings, as well as a mode of modification (i.e. replace training set or add to existing training set).
artefactURIs: a training set, i.e. an array of URIs of semantically annotated artefacts.

`void deleteModel(String moURI)`
input:

moURI: URI of a classification object to be removed from the repository.

The TrainClassifier service is implemented as a web service and the different functionalities offered by it are implemented as web methods. Figure 14 depicts internal procedure for creation of a classification model within the TrainClassifier service. Blue boxes are references to existing KP-Lab services, yellow boxes reference to newly designed SWKM services.

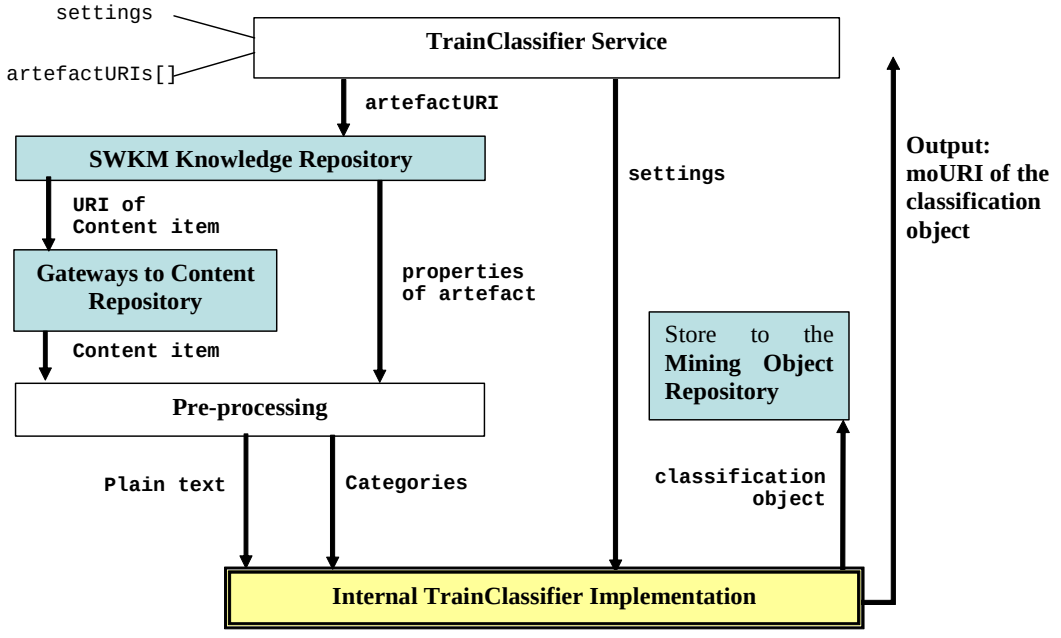


Figure 14: The LearningClassification Service

Classify service provides the following method for classification of artefacts:

```
String classify(String moURI, String[] artefactURIs,
    String format)
```

input:

moURI: URI of selected classification object.

artefactURIs: array of the artefacts to be classified. The artefacts are retrieved from the SWKM Knowledge Repository.

format: the format of output string. It can be TRIG or RDF/XML.

output: The output string contains a) an URI of the classified artefact, b) a category to which the artefact was classified, and c) a weight (score) of this particular classification. The format of the output string could be either TRIG or RDF/XML; the exact format is determined using the *format* parameter.

The Classify Service is also implemented as a web service that exposes its functionality via the Classify web method. Figure 15 summarizes the internal procedure of the Classify service.

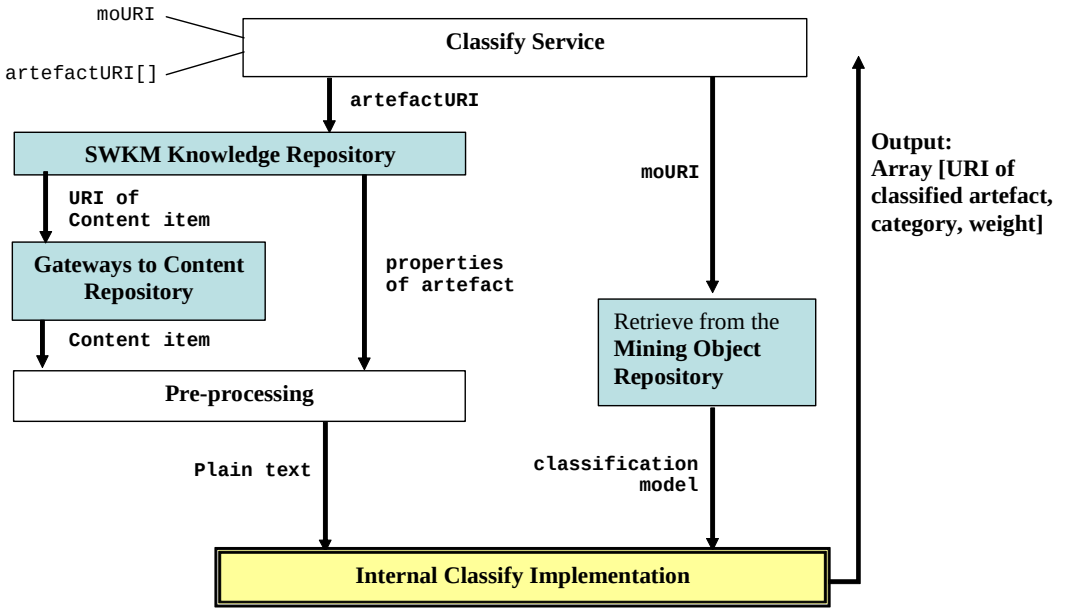


Figure 15: The Classify Service

The classification services itself have no user interface for these methods. It is assumed that the services are used in other KP-Lab tools, e.g. in the Shared Space (see example in Appendix) and the classification functionality will be used in the context of those tools. However, the *Mining Engine Console* is envisioned as a web-based application that exposes classification (as well as some clustering) functionality for KP-Lab users. It will enable to manage the Mining Object Repository, maintain classification models together with training and testing sets, view statistical reports for particular classification tasks, etc. The prototype of the Mining Engine Console was already developed and is available at http://kplab.fei.tuke.sk:8080/tmweb/admin/.

4 Conclusions and Future Work

The deep-level specification for the second release (M24) of the Knowledge Mediator and Knowledge Matchmaker components responsible for advanced manipulation with the knowledge stored in the SWKM was presented in this deliverable. Particularly, the *change*, *comparison*, *versioning* and *registry* services of the Knowledge Mediator component as well as the notification and text mining services of the Knowledge Matchmaker component were described along with the proposed functionality for each service, based upon the motivating scenarios and the subsequent functional requirements.

According to the [DoWB], the implementation of these components and services is planned to be delivered in M24. This deliverable, together with the previous deliverables [D5.1] and [D5.2], provide the specification that is sufficient for the implementation of the components and their integration with other KP-Lab tools.

Bibliography

[AMR06] Allert, H., Markkanen, H., Richter, C. (2006). Rethinking the Use of Ontologies in Learning. *Proceedings of the Joint International Workshop on Professional Learning, Competence Development and Knowledge Management - LOKMOL and L3NCD, Crete, Greece,* 8-18.

[Bednar et al 2005] Bednár, P., Butka P., Paralič, J.: Java Library for Support of Text Mining and Retrieval. In Proc. from the Czech-Slovak scientific conference Znalosti (Knowledge) 2005, Stará Lesná, Slovakia, 2005, pp. 162-169.

[Belhaj Frej et al 2006] Belhaj Frej, H., Rigaux, Ph., Spyratos, N.: User Notification in Taxonomy Based Digital Libraries (Invited Paper), ACM SIG-DOC Conference on the Design of Communication, Myrtle Beach SC, U.S.A., Oct 18-20, 2006.

[Belhaj Frej et al 2007] Belhaj Frej, H., Rigaux, Ph., Spyratos, N.: Fast User Notification in Large-Scale Digital Librairies: Experiments and Results, ADBIS 2007: Eleventh East-European Conference on Advances in Databases and Information Systems, Varna, Bulgaria, Sep 29 - Oct 03, 2007.

[BSD05] Benn, N., Shum, B.S., Domingue, J. (2005). Integrating Scholarly Argumentation, Texts and Community: Towards an Ontology and Services. *Tech Report kmi-05-5*, http://kmi.open.ac.uk/publications/pdf/kmi-05-5.pdf

[CSMWK] End User Requirements for Collaborative Semantic Modelling. Internal Report for the Working Knot on Collaborative Semantic Modelling, version 0.6, 30.07.2007.

[D2.1] KP-Lab project deliverable 2.1; http://www.kp-lab.org/intranet/work-packages/wp2/deliverable-2.1/)

[D2.2] KP-Lab project deliverable 2.2; http://www.kp-lab.org/intranet/work-packages/wp2/deliverable-2.2/)

[D4.2.2] Kp-Lab project Deliverable 4.2.2. Technical Framework Architecture Dossier - Release 2; http://www.kp-lab.org/intranet/work-packages/wp4/result/d4-2-2/

[D5.1] Specification of the SWKM Architecture (V1.0) and Core Services. KP-Lab project Deliverable D5.1, July 2006.

[D5.2] Prototype (V1.0) of the Knowledge Mediator, Repository and Manager. KP-Lab project Deliverable D5.2, December 2006.

[D8.1] Scenarios and User Requirements for KP-Labs in Education. KP-Lab project Deliverable D8.1, July 2006.

[DLM07] De Leenheer, P., Meersman, R. Towards Community-based Evolution of Knowledge-intensive Systems. In Ontologies, Databases, and Applications of Semantics, 2007.

[DF01] Ding, Y., Fensel, D. Ontology Library Systems: The Key to Successful Ontology Re-Use. In Proceedings of the 1st International Semantic Web Working Symposium (SWWS'01), 2001.

[Dom98] Domingue, J. Tadzebao and WebOnto: discussing, browsing, and editing ontologies on the Web. In Gaines, B., Musen, M. (eds): Proceedings of the 11th Workshop on Knowledge Acquisition, Modelling and Management, 1998.

[DoWA] Description of Work 2.1 Months 13–30, Part A. KP-Lab Consortium.

[DoWB] Description of Work 2.1 Months 13–30, Part B. KP-Lab Consortium.

[FFR96] Farquhar, A., Fikes, R., Rice, J. The Ontolingua server: Tools for collaborative ontology construction. Technical Report. - Stanford KSL 96-26, September 1996.

[Gar92] Gärdenfors, P. Belief Revision: An Introduction. In Gärdenfors, P. (ed). Belief Revision, pages 1-20, Cambridge University Press, 1992.

[GATE] GATE - General Architecture for Text Engineering - http://www.gate.ac.uk

[GK97] Gordon, T.F., Karacapilidis, N. The Zeno argumentation framework. In Proceedings of the 6th International Conference on Artificial Intelligence and Law, ACM Press, New York. 1997.

[Gru93] Gruber, T.R. A Translation Approach to Portable Ontology Specifications. 1993. Available at: http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html

[HHS99] Heflin, J., Hendler, J., Luke, S. SHOE: A Knowledge Representation Language for Internet Applications. Technical Report CS-TR-4078 (UMIACS TR-99-71), Department of Computer Science, University of Maryland at College Park, 1999.

[KMACPST04] Karvounarakis, G., Magkanaraki, A., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M., Tolle, K. RQL: A Functional Query Language for RDF. *TheFunctional Approach to Data Management*, pages 435-465, 2004.

[KFAC07] Konstantinidis, G., Flouris, G., Antoniou, G., Christophides, V. Ontology Evolution: A Framework and its Application to RDF. In Proceedings of the Joint ODBIS & SWDB Workshop on Semantic Web, Ontologies, Databases (SWDB-ODBIS-07), 2007.

[Lewis 1998] Lewis, D. D.: Naive (Bayes) at forty: the independence assumption in information retrieval. Machine learning: ECML-98, 10th European conference on machine learning. 1998, pp. 4-15.

[MacQueen 1967] MacQueen, J. B.: Some Methods for classification and Analysis of Multivariate Observations, Proceedings of 5th Berkeley Symposium on

Mathematical Statistics and Probability, Berkeley, University of California Press, 1967 1:281-297

[ONTSRV] Ontology Server research:
http://www.starlab.vub.ac.be/research/dogma/OntologyServer.htm#index

[Quinlan 1996] Quinlan, J. R.: Learning first-order definitions of functions. Journal of Artificial Intelligence Research, 1996, 5: 139-161.

[SEMSRCH] Bauters, M. et al: Semantic search draft requirements. Draft of requirements for semantic search in Shared Space M21 specifications. Internal Report for the Working Knot on Project and Content Management, version 0.1, 23.07.2007.

[Smrz et al 2007] Smrž, P., Paralič, J., Smatana, P., Furdík, K.: Text Mining Services for Trialogical Learning. In Proc. from the Czech-Slovak scientific conference Znalosti (Knowledge) 2007, Ostrava, Czech Republic, February 2007, pp. 97-108, ISBN 978-80-248-1279-3.

[Sta03] Stahl, G.. Meaning and Interpretation in Collaboration. In: Wasson, B., Ludvigsen, S., Hoppe, U. (eds.): *Designing for Change (pp. 523-553).* Dordrecht: Kluwer. 2003.

[STPBL] Bauters, M. et al: Semantic tagging according to PBL vocabulary requirements. Draft of requirements for semantic tagging in Shared Space M18 specifications. Internal Report for the Working Knot on Project and Content Management, version 0.5, 20.07.2007.

[Tou58] Toulmin, S. The Uses of Argument. Cambridge: Cambridge University Press. 1958.

[TCFKMPS06] Tzitzikas, Y., Christophides, V., Flouris, G., Kotzinos, D., Markkanen, H., Plexousakis, D., Spyratos, N. (2006). Emergent Knowledge Artifacts for Supporting Trialogical E-Learning. *Proceedings of the TEL-CoPs'06: 1st International Workshop on Building Technology Enhanced Learning solutions for Communities of Practice, Crete, Greece*, 162-176.

[Yang 2001] Yang, Y.: A Study on Thresholding Strategies for Text Categorization. Proceedings of SIGIR-01, 24th ACM International Conference on Research and Development in Information Retrieval, 2001, pp. 137-145.

[ZTC07] Zeginis, D., Tzitzikas, Y., Christophides, V. On the Foundations of Computing Deltas Between RDF Models. In Proceedings of the 6th International Semantic Web Conference (ISWC-07), 2007.

APPENDIX

A1. Example: Classification in the Shared Space

A student wants to create a new content item for the final report using the form in the Shared Space. He or she at first uploads a document file into the Shared Space and then specifies the metadata for the new item in the form for creation of the content item. The student can specify metadata like title or description, and add one or more tags from the predefined vocabulary to semantically annotate the new knowledge artefact (Figure A1-1).

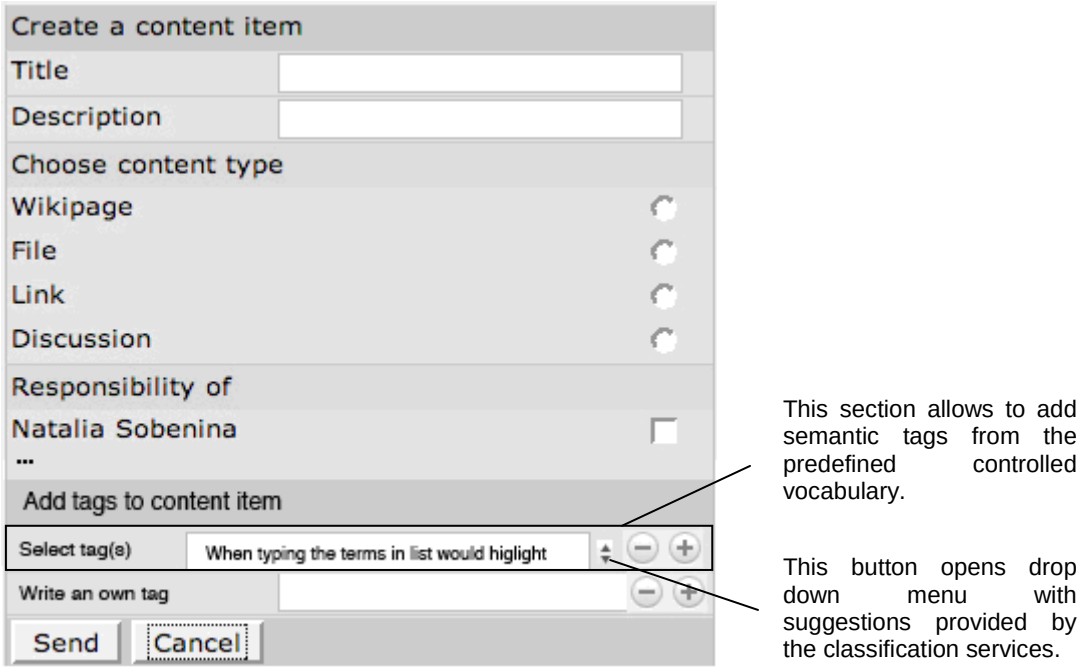


Figure A1-1. Create new content dialog with semantic tagging

After the user has uploaded the document file, the Shared Space stores the file in the content repository and sends the content URL to the classification service. The Classification Service will request new content item from the repository, analyze its text content and/or structure and apply various classification models. The result of the classification is a set of vocabulary terms suggested to the user. Each term included in the result can have additionally assigned real-valued score, which denotes the confidence that the content should be annotated with the given term.

One possibility on how to represent terms from the vocabularies in the Share Space dialog is to use a drop down menu (Figure A1-2). Suggestions for semantic annotations provided by the classification service are presented in a separate section of this menu. The user can browse the results ordered according to the confidence score, as well as to browse other terms of the vocabulary to supplement or correct suggestions.

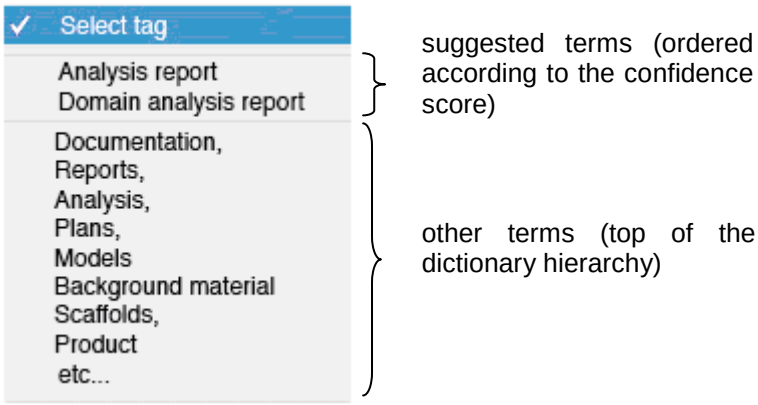


Figure A1-2. Drop-down menu for semantic tagging with suggested terms for PBL vocabulary

Various dictionaries can be specified for metadata tags including dictionaries for document type as is specified in PBL vocabulary [STPBL] or domain specific dictionaries to describe document subject. The subset of the tags and corresponding dictionaries supported by the classification service is the subject of ongoing research and depends mainly on the accuracy of the implemented models.

The previous case is an example of a single classification when the only new item is classified for semantic annotation. It is possible that a client of the classification service, i.e., the Shared Space application, sends a set of content items to be classified by the Classification Service. For example, the user can select content items in his/her shared space and then use the classification service to additionally classify all these items according to the selected controlled vocabulary. With this “batch” classification, the user can dynamically create temporal views of his/her shared space. The result of the classification can then be permanently stored as a list of semantic annotations of the classified items and can be used later to visualize the shared space.