27490

KP-LAB

Knowledge Practices Laboratory

Integrated Project

Information Society Technologies

D5.5: Specifications and Prototype of the Knowledge Repository (V.3.0) and the Knowledge Mediator (V.3.0)

Due date of deliverable: **31/01/09**
Actual submission date: **09/02/09**

Start date of project: 1.2.2006 Duration: 60 Months

Organisation legal name of lead contractor for this deliverable:
Foundation for Research & Technology – Hellas (FO.R.T.H.)

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):	Dimitris Andreou	ICS-FORTH
	Vassilis Christophides	ICS-FORTH
	Giorgos Flouris	ICS-FORTH
	Dimitris Kotzinos	ICS-FORTH
	Panagiotis Pediaditis	ICS-FORTH
	Petros Tsialiamanis	ICS-FORTH
Editor(s):	Giorgos Flouris	ICS-FORTH
Partner(s):	ICS-FORTH	
Work Package:	WP5 – Semantic Web Knowledge Middleware	
Nature of the deliverable:	Report	
Internal reviewers:	Michal Racek (POYRY) Ivan Furnadjiev/Tania Vasileva (TUS)	
Review documentation:	http://www.kp-lab.org/intranet/work-packages/wp5/result/deliverable-5.5/	

Version history

Version	Date	Editors	Description
0.1	September 25, 2008	Giorgos Flouris, Vassilis Christophides	Initialization document, tasks and responsibilities.
0.2	December 18, 2008	Giorgos Flouris	First draft ready
0.8	January 12, 2009	Dimitris Andreou, Markos Charatzas, Vassilis Christophides, Giorgos Flouris, Dimitris Kotzinos, Panagiotis Pediaditis, Petros Tsialiamanis	Incorporated comments and various corrections, second draft ready
0.9	January 16, 2009	Giorgos Flouris, Dimitris Kotzinos	Final proof-reading done, final draft ready for internal reviews
1.0	February 4, 2009	Giorgos Flouris, Dimitris Kotzinos, Vassilis Christophides	Review comments considered and incorporated in the document; final version delivered

Executive summary

This deliverable reports the technical and research development performed until M36 (January 2009) within tasks T5.2 and T5.4 of WP5 in the KP-Lab project, per the latest Description of Work (DoW) 3.2 [DoW3.2]. The described components are included in the KP-Lab Semantic Web Knowledge Middleware (SWKM) Prototype Release 3.0 software that takes place in M36. This release builds on the Prototype Release 2.0 that was presented in [D5.4].

The present deliverable includes both the specification, as well as the implementation details for the described components. The description of the features of the new functionalities is provided based on the motivating scenarios and the subsequent functional requirements. The focus and the high-level objective of the new services is the provision of improved scalability and modularity properties on the existing services, as well as improved management abilities upon conceptualizations. The implementation of the services is described by providing the related services' signatures, their proper way of use, the accepted input parameters, as well as their preconditions and effects.

Initially, we describe the Delete Service, which is a Knowledge Repository service allowing the removal of existing namespaces from the repository; such removal includes the deletion of the contents of said namespaces, as well as the deletion of any reference to the namespaces themselves that exists in the repository. This new service enhances SWKM management capabilities upon conceptualizations.

Then, the Named Graphs functionality is described, which is a new feature that allows a very flexible modularization of the information found in RDF KBs. We describe in detail the semantics of this feature, as well as the offered capabilities for querying and updating RDF KBs that include modularization information (i.e., information on named graphs) and the implications from their use.

Finally, in the context of the Knowledge Mediator, we present the Persistent Comparison Service, which is a variation of the existing (Main Memory) Comparison Service (see M24 release, [D5.3], [D5.4]); unlike the original version, the new service works exclusively on the persistent storage, guaranteeing improved scalability features.

Table of Contents

1 Introduction

In the context of KP-Lab we need to create, evolve and manage various kinds of conceptualizations supporting either KP-Lab tools' interoperation [D4.2.3II], or learners'/workers' knowledge transformation practices [DKKC08]. Consider for instance, conceptualizations that describe learners'/workers' understanding of a particular object of interest or phenomenon under investigation; such conceptualizations play essentially the role of epistemic artefacts allowing knowledge workers/learners not only to externalize their understanding of the domain/problem at hand but also to compare their underlying modelling practices. In particular, it also allows them to elicitate complementary or contradictory viewpoints. Such kinds of knowledge creation processes constitute the sparkle of the trialogical framework [TCFKMPS06], [TCFKMPS07].

As described in [D5.3], such conceptualizations are represented using *RDF/S Knowledge Bases* (*KBs*). More precisely, a conceptualization usually comprises an ontology along with corresponding instantiations of its classes and properties. An ontology is a vocabulary of terms (i.e., a taxonomy), enriched with various types of constraints, relationships and rules which can be expressed in the RDF/S data model (see [KMACPST04] for details).

In order to effectively support the life-cycle of such conceptualizations, a number of services have been developed, and are currently available for use in the context of the KP-Lab Environment and end-user tools. Such services have been described in previous deliverables ([D5.1], [D5.3]) and are included in the second prototype release of the SWKM services (V2.0), which was released in M24 and described in [D5.4]. In the present deliverable, we present the specification and implementation of some additional services developed during the last year, aiming to enhance the management facilities of RDF KBs. The new services along with refined versions of the old ones are part of the third prototype release of the SWKM services (V3.0), which is due in M36 (January 2009) and have been developed in the context of tasks T5.2 (SWKM Knowledge Access and Evolution Services) and T5.4 (SWKM Knowledge Repository).

The current deliverable includes both the specification of the new services, as well as the technical details regarding their implementation. The new services include the *Delete Service* (Knowledge Repository), the enhancement of the Query/Update Service with *named RDF/S graphs* (Knowledge Repository) and the *Persistent Comparison Service* (Knowledge Mediator). The objective behind the introduction of the new services is the provision of *improved scalability and modularity properties* on the existing ones, as well as *improved management abilities* upon existing conceptualizations.

More specifically, the manipulation and management of RDF/S KBs is based on the Knowledge Repository storage and its services: Import, Export, Query, Update [D5.1]; one missing piece of functionality is the ability to remove obsolete conceptualizations from the repository, which were imported but not used anymore. The Delete Service covers this need, as it was developed in order to allow the user to remove in a consistent way RDF/S KBs from the knowledge repository. The typical

usage of this service is the removal of an RDF/S namespace which is, for various reasons, no longer needed or wanted. The Delete Service can be used to completely remove an unwanted namespace and its corresponding instantiations from the repository; in effect, this action corresponds to "undoing" a previous import operation, as it removes all traces of the requested namespace from the repository, including the triples that belong in the namespace, the reference to the namespace from the list of namespaces in the repository, all dependency links to/from the namespace etc.

On the other hand, the ability to modularize RDF/S graphs is expected to serve several needs in KP-Lab, for example in determining the origin or ownership of information (per learner or per source of information consulted), or how a particular piece of information evolves, or in order to impose flexible access control policies on parts of an RDF KB. These needs arise by the High-Level Requirements (HLRs) of KP-Lab [D2.4]; we elaborate on this in section 3.4. The named graphs functionality covers this modularization need by allowing the user to define, store and manipulate modules of RDF/S graphs; such modules are defined using *named graphs*, or sets of named graphs (called *graphsets*). Named graphs and graphsets can be viewed as containers of triples and represent the "logical parts" (modules) of the full RDF/S graph. Named graphs and graphsets provide a very flexible and powerful way to represent modules of RDF/S graphs, which is suitable for several diverse applications, as described above. The introduction of the modules in RDF/S graphs requires suitable support both at the level of the underlying representation of triples (for storing the association of each triple with the named graphs or graphsets that it belongs to) and at the level of SWKM services (to exploit the advanced RDF/S modelling capabilities offered by named graphs). In this deliverable, we describe in detail the semantics of named graphs and graphsets, as well as the support that is currently provided in terms of querying and updating named graphs and graphset information (per DoW 3.2 [DoW3.2]).

Apart from the Knowledge Repository, this deliverable also deals with the Knowledge Mediator which includes the knowledge evolution services, i.e., Comparison, Change Impact and Versioning. One need that was identified was the scalability of these services in order to be usable over large KBs, given that the size of RDF/S KBs manipulated in KP-Lab is increasing (approximately 238% on average over the last four months). Thus, scalability of the evolution services emerges as a crucial requirement in order to successfully deploy the KP-Lab Environment and tools in real working and learning settings.

Scalability can be achieved by moving the bulk of the computational and storage needs of the services from the main memory to the persistent memory. Using this idea, and starting from the Comparison Service ([D5.3]), we developed a version of the service that works on the persistent memory (Persistent Comparison Service). This service enables to outline the differences between variations of conceptualizations expressed in the RDF/S data model in a similar way to the functionality offered by the Main Memory Comparison Service (described in [D5.3], [D5.4]). However, rather than in main memory, comparisons of RDF/S KBs are performed directly in secondary memory (i.e., the knowledge repository), and thus the implementation challenges are different. In addition, the user of the two services (Main Memory Comparison Service and Persistent Comparison Service) is faced with an interesting

trade-off between the maximum supported size of compared RDF/S KBs and the time required to compute the differences.

2 Motivation

2.1 Collaborative Modelling

Collaborative modelling constitutes an important knowledge practice in quite many professional and scientific communities, as well as in various educational settings. Models are more than mere descriptions of a particular object of interest or phenomenon under investigation, because they provide important epistemic features (i.e., for understanding of the models *per se*, as well as of the underlying modelling practices) that can trigger exploration, inquiry, and knowledge creation [PH05]. Recent advances in semantic web technology provide new and more powerful means to support collaborative modelling activities by allowing the users to externalize, share, and evolve their own models and modelling languages.

In the context of the KP-Lab project, two examples of recently developed tools [D6.6] for supporting collaborative modelling activities centred around conceptualizations expressed as RDF/S KBs [D5.3] are the Collaborative Semantic Tagging [SemTag], in which learners collaboratively annotate various content items with semantic tags (i.e., vocabulary terms), and the Collaborative Semantic Modelling [ColMol], in which learners have additionally the possibility to structure the terms of their vocabularies using various semantic relationships such as “is_A” and “has_part”. These tools are developed to cover different needs and requirements of the learners [D2.4].

In these tools, collaborative modelling, rather than being an isolated activity, is tightly integrated into everyday groups’ work practices, along with an open access and reference to various forms of employed knowledge artefacts. Moreover, knowledge workers and learners are stipulated to develop alternative conceptualizations of a particular object of interest or phenomenon under investigation and to support triangulation of the different perspectives, without losing the information regarding the origin of (and the rationale behind) each conceptualization. In order to trace the rationale of the conceptualizations’ evolution, means for comparing successive versions of conceptualizations have to be in place, as well as negotiation and argumentation mechanisms that support the exchange of ideas towards converging to a common understanding of the domain at hand. In addition, the evolution and negotiation process often implies that older versions may have to be discarded in the process of building new and more adequate conceptualizations.

2.2 Emerging Functionality

In this setting, collaborative modelling activities may take several forms. For example, users may concurrently edit some shared conceptualization, while seeing synchronously the changes performed by the whole group. Alternatively, they may edit their conceptualizations locally on their personal space and commit and merge asynchronously their conceptualizations in a shared space. In the latter scenario, the merging process may be either initiated centrally by a curator of the models or in a peer-to-peer fashion by the learners themselves. In both cases, adequate tools are

required to monitor and analyze the differences of the different conceptualizations as well as to record the rationale of the underlying modelling choices made by the knowledge workers or learners in order to argue and negotiate about the acceptance or rejection of the changes made so far. During the whole process, we may need to keep in the knowledge repository the full history of intermediate versions, or only the most "important" ones.

A comprehensive analysis and classification of the various dimensions characterizing collaborative modelling activities was presented in [NCLM06] and is briefly reported in the following:

- Whether the collaboration is *synchronous* or *asynchronous*, i.e., whether the knowledge workers/learners will collaborate on the same version of the conceptualization or using different local copies which are afterwards committed and merged. In the former case, adequate support for inspecting the effects and side-effects of the proposed changes on the same version of a conceptualization are required (see also Change Impact Service [D5.3]). In the latter case, support for fusing automatically different conceptualizations when the curator or the learners take the final decisions is required (see also Knowledge Synthesizer [D5.6]).
- Whether the editing is performed in a *continuous manner*, saving only the latest version (and allowing rolling back to any previous version – see Versioning Service [D5.3]), or in a *step-wise manner*, in which different "official" versions of a shared conceptualization are published, but the history of the individual edits between versions is not kept (and thus discovery of the differences between consecutive versions can be performed only *a posteriori* – see Comparison Service [D5.3]).
- Whether there is some central control (*curation*) over the contents of the produced shared conceptualization or only peer-to-peer interactions are supported. The existence of multiple versions of the shared conceptualization and/or local ones highlights the need for adequate management tools (e.g., to import, export [D5.1] or delete conceptualizations from the knowledge repository, as well as to store metadata on the conceptualizations through the use of a Registry Service [D5.3]).
- Whether there is some (semi-)automated assistance that records (*monitors*) the performed edits and, possibly, creates adequate metadata and/or logs on the changes implemented so far. In particular, metadata about the origin (e.g., a group or a curator) and the reasons why the changes were made (e.g., whether the appearance or disappearance of a concept in a new version relates to the appearance or disappearance of other concepts or of relationships among them) are crucial in order to compare the underlying modelling practices (such metadata are usually captured by a Registry Service [D5.3]).

Depending on the context, different combinations of the above dimensions stipulate different processes of collaborative knowledge creation, and result in different high-level requirements for the SWKM services and the KP-Lab end-user tools supporting such processes. For example, the Visual Model Editor (VME) and Visual Modelling Language Editor (VMLE) [D6.6] support asynchronous, step-wise collaborative editing, which could be curated or non-curated.

In the sequel, we will detail the design rationale of the new SWKM services delivered for M36. In this respect, we consider that shared or individual conceptualizations of knowledge workers and learners are represented by a series of user-defined RDF KBs, including both the schema and the data of each conceptualization.

It should be stressed that SWKM services aim to support not only the management of *user-defined ontologies* (conceptualizations) but also of the *KP-Lab system ontologies* (see [D4.2.3II] for details), thus enabling tools' interoperation. Even though such ontologies are relatively stable, changes may occur from time to time, especially when new versions of the KP-Lab end-user tools are released. The main additional need that arises from the existence of such ontologies is related to such changes. In particular, when an old version of a system ontology is replaced by a new one, the older version should be entirely deleted from the repository whereas the (usually large amount of) underlying instances need to be reclassified under the new ontology version to ensure a seamless functioning of the related services and tools.

3 High-Level Functional Requirements

3.1 Knowledge Repository

3.1.1 Removing Conceptualizations

As already mentioned, one of the needs that arise in the process of managing ontologies and RDF/S KBs is the need to withdraw an existing conceptualization that is no longer needed, or to "undo" some storage operation which was made by mistake, or using the wrong input etc. In the context of KP-Lab, this need may arise in several scenarios.

As an example, consider the case of a learner that starts developing a conceptualization in his personal space (e.g., a visual modelling language in the VMLE tool), but, in the process, he realizes that his efforts are totally out of track. In such a case, the learner might be better off starting his efforts from scratch and discarding whatever he has developed so far, rather than attempting to correct the existing conceptualization. If, however, he has already stored his conceptualization at the knowledge repository, this also includes the removal of the conceptualization from the repository.

In another context, the deletion operation may be useful in order to allow the replacement of an old ontology version by a new one. Furthermore, the ability to remove conceptualizations from the repository may be seen as a way to "undo" previous storage operations; such a need may arise, for example, when an import was made by mistake, or using the wrong set of input files (serializations).

As conceptualizations are represented by RDF/S KBs, the Delete Service is essentially used to remove RDF/S namespaces and instances classified under these namespaces. The Delete Service should work directly upon the persistent storage, and it should remove both the contents of the removed namespace (i.e., classes and properties), the classification links of the instances classified under the deleted classes

and properties, as well as any references to the namespace itself that may exist in the database.

It should be stated that erasing the contents of a namespace and the instances classified under this namespace could not be fully supported using existing services (e.g., Update and Change Impact Services). The reason is that the Update Service can only be used to remove instance resources of a namespace, so the schema cannot be deleted, whereas the Change Impact Service works on the main memory and cannot, therefore, directly delete the contents of a stored namespace. Moreover, these services can only be used to remove the *contents* of the namespace, but cannot be used to remove the *references to the namespace* in the list of the dependent namespaces, or in other records maintained by the Knowledge Repository (e.g., dependency links). Thus, by simply removing the classes and properties of a given namespace we would get an empty namespace, but the reference(s) to the namespace itself would persist in the repository.

The deletion of a namespace may cause problems related to the validity of the Knowledge Repository. For example, if the deleted namespace has some dependent namespaces or instance resources, and we proceed with its deletion in a straightforward manner, then we risk the existence of dangling references to physically deleted resources.

The easy way to avoid this problem is to state that, when deleting a namespace, all dependent namespaces and instance resources should be deleted along with it. This is not always the desirable behaviour of the service though: in many cases, we may not know what are the dependents of a namespace, or who uses them and for what reason. Thus, deleting such dependent namespaces may cause problems to other users. In such a scenario, we should not be allowed to delete any dependents, so we have no option but to abort the operation if there are any dependents. In other cases (for similar reasons), we may want to retain the data classified under the deleted namespace's classes and properties. Note that retaining, in this context, does not mean that the operation should be aborted, as in the case of dependent namespaces. Instead, there is a more clever way out of the problem, namely the *reclassification* of the data in a way that would leave no dangling references in the database. The reclassification should classify the class and property instances under the minimal superclasses and superproperties of the deleted classes and properties.

The above considerations apply both in the context of user-defined ontologies and system ontologies. Note that both types may have interdependencies (see also [D4.2.3II]), as well as data, so the above options make sense for both cases; nevertheless, we expect that for system ontologies the option of reclassifying the data will often be chosen, whereas for user-defined ontologies either the option of removing the data and the dependent namespaces, or the option of retaining them altogether will be chosen. From the above requirements, it follows that the Delete Service should support a variety of operational modes, giving the user the option to determine the desirable behaviour of the service on a per-case basis.

3.1.2 Modularity of Conceptualizations

One of the requirements that arise from the previously described scenarios (section 2) is the need to record the origin of each piece of knowledge codified by a

collaboratively developed conceptualization. Keeping track of metadata about creation and modification history, influences, ownership, as well as other *provenance* or *lineage* information (see [Tan07] for a survey) is crucial in order to make *informed judgments about the quality, integrity, and authenticity* of data and knowledge developed by a group of editors (in synchronous/asynchronous or centralized/peer-to-peer settings).

In the context of learning, this requirement could appear, for example, in the case of a learner who uses VME to create a model recording information found in several different sources (e.g., books, web pages, other co-learners etc). In many cases, it is useful not only to store the relevant information itself, but also to store the *source* of the information. The latter (source of the information) could be important in determining the *support*, or *reliability* of each piece of information in the conceptualization.

This need could be viewed as part of a more general need for modularization of conceptualizations represented as RDF/S KBs. Depending on the granularity of the logical modules and the modularization policy, modularization may be useful in different ways, e.g., by allowing learners to claim "ownership" of some part of an RDF/S KB (see HLR4.1, HLR9.1 [D2.4]), or by describing the source or modelling rationale underlying each contribution made (see HLR4.4, HLR12.1 [D2.4]). In order to support such a modularization, we should be able to associate each of the triples that compose the RDF/S KB with zero, one, or more than one module.

Note that the solution of codifying each "module" into a separate conceptualization (e.g., a separate RDF/S KB) is not enough for our purposes, because the connection between such modules, in the context of a larger conceptualization that engulfs all of them, would be lost. Instead, each module would be viewed as a separate conceptualization, a fact which does not coincide with our original intentions.

Some (simple) kind of modularization within a single conceptualization is offered by namespaces [BHLT06]. However, the modularization offered by the namespaces solves our problem only partially, as namespaces have a number of deficiencies regarding the modularization they offer. First of all, the modularization offered by namespaces is restricted to the schema level only; it is not clear where (i.e., in which namespace) a data triple belongs to. Secondly, the focus of namespaces is on modularizing the names (URIs) of classes and properties defined in ontologies; what we need here is a modularization of the conceptualization itself, which is actually composed by the triples that exist in the RDF/S KBs. Thirdly, namespaces are not flexible in their modularization abilities, since any name in the schema must belong to one, and only one, namespace; therefore, sharing of information between namespaces is not allowed.

As a consequence, we need some other mechanism that will be used to group triples into modules. Each such module (as a whole) should be a resource of its own which should be accessible, referable to, and which could, itself, be associated with some metadata information. This feature is necessary in order to decide "how credible is", or "how evolves" a piece of knowledge codified by a conceptualization.

One of the most difficult problems that arise during modularization is the fact that some parts of the information (especially non-explicit ones) cannot be clearly classified in one of the modules. As an example, consider the scenario where each module represents the origin of the information; consider also some piece of information (say z) which is not explicit, but implied by two explicit pieces of information which have different origin (say x, y, with origin G_x, G_y respectively). In this case, what is the origin of z? The only satisfactory answer would be that the origin is not any of the existing modules, but is *shared* between G_x, G_y; thus, in this case, z is assigned to more than one modules *at the same time and in a shared fashion*, so our solution should support the assignment of some information in the conceptualization to more than one modules in a shared fashion.

Note that the "shared assignment" of a triple to a set of modules is different from the multiple assignment of a triple to different modules. In the first case, the assigned triple does not belong to any one of the modules, but it belongs to all of them in a joint fashion; in the second case, the triple belongs independently to each of the modules. The above two modes of triple assignment to modules could be combined.

Of course, having defined modules that satisfy the above requirements is not, by itself, enough. We need such modules to be manageable by the various services, meaning that the underlying representation should be able to record the assignment of information into modules and that all the SWKM services should recognize and support such modules. This requirement asks for the enhancement of all existing SWKM services in a way that they will be able to understand, store, retrieve, query, update, compare etc information on the modules, as well as triples that are assigned to specific modules, taking into account the assignment information.

3.2 Knowledge Mediator

3.2.1 Scalable Comparison of Conceptualizations

Comparing individual viewpoints of a particular object of interest or phenomenon under investigation is one of the main activities towards the construction of a shared conceptualization among group members. First of all, it helps identifying how shared or individual conceptualizations evolve over time. Additionally, it may be viewed as an aid towards the negotiation and argumentation process, because it helps the learners identify the converging and conflicting parts between their viewpoints [NCLM06]; this allows the learners to focus on the points that cause disagreements and need further argumentation and negotiation.

Note that such a comparison should have the ability to take into account not only explicit but also implicit knowledge encoded in a conceptualization expressed in RDF/S (see also [ZTC07]). The need for comparing conceptualizations has been elaborated in [D5.3], where we described the Comparison Service that allows us to detect the differences between two RDF/S KBs under various modes and parameters.

The main characteristic of the Comparison Service version implemented for the second release (V2.0) in M24 [D5.4] is that it works on the main memory. This means that the compared RDF/S KBs must be loaded in the main memory before being compared, so they have to fit into the available main memory. This approach allows

fast execution of the comparison, but, unavoidably, is constrained by the size of the available memory. Therefore, even though the Comparison Service performs well for small and medium-sized conceptualizations, it does not scale for large conceptualizations. The problem does not usually appear when one considers only the schema information of RDF/S KBs (which is often small), but it does appear when the instances are also considered.

It also emerges as a need in the context of the KP-Lab project since a large number of instances is actually starting to populate the existing KP-Lab ontologies [D4.2.3II] and their number is constantly increasing. To be more specific, the knowledge repository currently (January 2009) contains more than 25000 class instances, classified under 219 classes, and more than 125000 property instances, classified under 349 properties; this size is constantly rising: for example, during the last four months (September 2008-January 2009), the number of objects in total (classes, class instances, properties, property instances) in the knowledge repository has risen by approximately 238%. Note that this increase does not affect so much the services that work directly upon the repository (e.g., Query, Update etc), but it affects a lot the services that work on the main memory (e.g., Change Impact, Comparison etc).

To address this problem we have implemented for M36 a new version of the Comparison Service which is able to compare conceptualizations directly in the secondary memory of the knowledge repository. Even though such an implementation is slower than the original main memory implementation (because accesses to the hard disk are slower than accesses to the main memory), it is not limited by the size of the machine's main memory, but by the size of the machine's hard disk, which is expected to be much larger. The functionality and behaviour of the Persistent Comparison Service is identical to the one provided by the main memory version of the service. Therefore, the high-level functional requirements for the Persistent Comparison Service are the same as those described in [D5.3] regarding the Main Memory Comparison Service.

The persistent version of the Comparison Service is not meant to replace the original, main memory version. Instead, the existence of both implementations of the Comparison Service provides the KP-Lab system developers the opportunity to use either, depending on the setting; in particular, for small and medium-sized RDF/S KBs, they can employ the main memory implementation, which will execute the comparison more efficiently than the persistent implementation, whereas for comparing large RDF/S KBs they may employ the persistent version of the service, which is scalable and guaranteed to produce a result, even though it is not as efficient as the main memory implementation.

3.3 Summary of Functionalities

The following table summarizes the high-level requirements identified for the services and functionalities described in this deliverable, as well as the service or functionality that provides the related function.

Functionality	**Short Description**	**Related Service or Functionality**
Knowledge Repository		

Remove a conceptualization along with its dependent namespaces and instance resources	Allow the removal of a namespace from the repository, including its contents and any references to it; any dependent namespaces or instance resources should be deleted as well in the same way	Delete Service allows the deletion of namespaces, including their contents and any references to them; exact behaviour determined by the mode of operation
Remove a conceptualization only if it does not have any dependent namespaces or instance resources	Allow the removal of a namespace from the repository, including its contents and any references to it; if it has any dependent namespaces or instance resources, neither them, nor the namespace should be deleted	Delete Service allows the deletion of namespaces, including their contents and any references to them; exact behaviour determined by the mode of operation
Remove a conceptualization along with its dependent namespaces; any dependent instance resources should be reclassified	Allow the removal of a namespace from the repository, including its contents and any references to it; any dependent namespaces should be deleted, and any dependent instance resources should be reclassified	Delete Service allows the deletion of namespaces, including their contents and any references to them; exact behaviour determined by the mode of operation
Create and store modules of information	Create modules of information and store them in the repository	Named graphs and graphsets allow the definition of highly flexible modules
Find the triples, nodes or modules that satisfy a certain property	Query RDF/S KBs taking into account module information	An extension of RQL allows the execution of more sophisticated queries that can return and consider triple assignment to modules
Add/delete triples to/from modules	Update RDF/S KBs, including module information and the association of triples with modules	An extension of RUL allows the execution of more sophisticated updates that can update and consider triple assignment to modules
Manipulate modules	Create new modules or remove existing ones from the repository at will	An extension of RUL allows the execution of special updates that create and remove modules
Knowledge Mediator		

Scalable comparison	Allow the comparison of large conceptualizations, in a scalable way	Persistent Comparison Service allows the comparison of large conceptualizations; the comparison is performed at the repository level for scalability purposes

Table 1: Summary of Requirements and Functionalities

3.4 Connection with KP-Lab HLRs

In [D2.4], a number of User Tasks (UTs), Driving Objectives (DOs) and High-Level Requirements (HLRs) were defined. Here, we will describe the connection of such UTs, DOs and HLRs with the components presented in this deliverable, as well as the relation of such components with non-functional requirements in the KP-Lab project.

3.4.1 Delete Service (Knowledge Repository)

The Delete Service is associated with HLR4.4:"Users are able to save and share conceptual models (e.g. vocabularies and visual models)", which is part of DO4:"Users can describe the semantics of artefacts and their relations" and UT2:"Modifying the content of the shared artefacts individually and collaboratively". It is also associated with HLR6.3:"Users can share and integrate different visual modelling languages, ontologies and vocabularies", which is part of DO6:"Provide users with possibilities to develop and use their own conceptual models" and UT2:"Modifying the content of the shared artefacts individually and collaboratively". Both associations stems from the fact that the sharing of visual models, visual modelling languages, ontologies, or vocabularies would imply the existence of various versions of the same conceptualization; therefore, improved management capabilities of such multiple versions should exist, including the ability to remove obsolete conceptualizations.

3.4.2 Named Graphs (Knowledge Repository)

The concept of named graphs intends to support a number of functional and non-functional requirements of the KP-Lab Environment and tools. More precisely, named graphs is a generic mechanism for modularizing knowledge in such way that can be used to identify (through URIs) and establish references to sub-graphs of large RDF/S KBs and to encode additional information such as the origin of the sub-graph or the access policies related to it. One of the features of named graphs is that they allow for simultaneous memberships, so that any artefact can participate in any number of named graphs at the same time. On the usability side, named graphs allow us to improve query performance by restricting the search space only to sub-graphs of interest (so, e.g., costly and frequent queries related to the retrieval of a KP-Lab shared space could be optimized in this respect). In the sequel, we will present the main HLRs related to the functionality of named graphs.

First of all, the named graphs functionality is associated with HLR1.1:"Users can create structure and share various artefacts (e.g. sketches, various kinds of texts, video and audio-files, models as well as ontologies) in one place", which is part of DO1:"Provide a collaborative environment where users can work on shared artefacts" and UT1:"Organizing shared artefacts and collaborative tools"; in a more generic sense it is also associated with DO3:"Users are provided with support for the re-use of

shared artefacts and structures", which is part of the same UT. As described above, named graphs allow us to structure collections of artefacts (and their relationships) and handle them afterwards as unique entities, referenced by a unique URI in the RDF/S KB. Named graphs can serve any purpose since they do not carry any predefined semantics allowing the user to attach any semantics as he deems appropriate.

Along the same lines, HLR4.4:"Users are able to save and share conceptual models (e.g. vocabularies and visual models)", which is part of DO4:"Users can describe the semantics of artefacts and their relations" and UT2:"Modifying the content of the shared artefacts individually and collaboratively" can be served by the capability provided by the named graphs to encode in a flexible way (i.e., through reification) the source of each RDF/S knowledge module without imposing a particular RDFS schema (since they carry no semantics themselves) for describing the actual contents of the module; this is coupled with the capability to provide a single and unique way (URI) to reference any (visual) model enabling its seamless saving and retrieval.

Furthermore, named graphs functionality is associated with HLR9.1:"Users can track the evolution and changes of knowledge objects and find out their authors and contributors (sequences of performed steps in time, incl. versioning)", which is part of DO9:"Users are provided with history on content development and work process advancement" and UT3:"Management and organization of collaborative work processes". With named graphs, changes can be tracked, comparisons can be made and evolution can be captured in a more aggregate level (not only at the single artefact level), since named graphs can be seen as single entities.

Finally, the named graphs functionality is associated with HLR12.1:"Users can work around a shared "virtual whiteboard" view where collaborative modelling and discussion takes place", which is part of DO12:"Provide users with means to capture, reflect, discuss and model their activities and to develop new models of working" and UT5:"Investigation and development of knowledge practices". In order to better serve this HLR we will need to build upon the ability of named graphs to encode information on the source (origin) of each piece of data; this kind of information is useful during collaborative modelling and discussion. Moreover, it provides users with the ability to separate modelling or discussion sessions, refer to or comment on them as single unique entities and reuse them or their contents as they see fit.

3.4.3 Persistent Comparison Service (Knowledge Mediator)

The Persistent Comparison Service is associated with HLR4.5:"Users are able to compare and integrate different knowledge representations/visual models", which is part of DO4:"Users can describe the semantics of artefacts and their relations" and UT2:"Modifying the content of the shared artefacts individually and collaboratively". The association originates from the fact that the Persistent Comparison Service provides another, more scalable way to execute the comparison of different knowledge representations/visual models.

4 Functional and Architectural Design

4.1 Knowledge Repository

4.1.1 Delete Service

The purpose of the Delete Service is to *delete an entire namespace* (including its contents) from the underlying repository. Upon successful execution, the Delete Service removes the requested namespace (including its contents) from the database and could also possibly affect dependent namespaces and data, depending on the user's choice on the mode of operation (see [D5.4] for details on namespace and data dependencies).

Let us initially consider the simple case where the user requests the deletion of a namespace (say ns1), which has no dependent namespaces or data triples (the namespace ns1 itself may depend on other namespaces). This is the simplest case, in which the namespace's contents, as well as any references to it are removed from the database, in particular:

- All the triples that use names from ns1 are deleted from the database.
- All the names from ns1 and the references to them are removed from the database.
- All the references to ns1 are deleted from the database (e.g., ns1 is removed from the list of namespaces, all namespace dependency links that involve ns1 are removed etc).

At the end of the operation, there will be no trace of ns1 in the database. The Delete Service can, in this respect, be considered as the "complement" of the Import Service, in the sense that if we import a namespace and then delete it, the repository will return to its original state; similarly, if we delete a namespace and then re-import it, the database will return to its original state. As a result, the Delete Service allows us to "undo" import operations (and vice-versa).

The above are true so long as ns1 does not have any dependent namespaces or data triples. If a namespace ns2 (or data triple t) depends on ns1, then the dependent namespace ns2 (or data triple t) has no valid meaning without the existence of ns1, because it refers to URIs (resources) defined in ns1. As described in the previous section, dealing with such dependent namespaces and data triples can be done in different ways.

The simplest way to deal with the problem of dependent namespaces and data triples is to delete them along with the deleted namespace. However, as explained before, there are cases where this kind of action is not desirable. In such cases, the user has the option to retain dependent namespaces and data triples.

In the former case (i.e., retaining dependent namespaces), the Delete Service cannot proceed with the deletion of the original namespace, as that would render the repository to an invalid state. More specifically, the removal of the namespace's contents would create dangling references in the dependent namespaces. Thus, if there are any dependent namespaces and the user disallows their deletion, the service will fail, and nothing will be deleted.

In the latter case (i.e., retaining dependent data triples), the Delete Service can overcome the problem by *reclassifying* the instances involved in the dependent data triples, rather than deleting the dependent data triples altogether.

The reclassification is performed as follows: data originally classified under about-to-be-removed classes (or properties), will be reclassified under certain superclasses (or superproperties) of the about-to-be-deleted classes (or properties); those superclasses (and superproperties) are all the minimal superclasses (or superproperties) that are not in the namespaces that are about to be deleted. This way, all the problematic data triples are replaced with triples that would not create any dangling references.

More precisely, the steps of the aforementioned reclassification are as follows:

- Consider an individual &a which is directly classified under the classes $A_1,\ldots,A_n$. Let us assume that the classes $A_1,\ldots,A_k$ are contained in the namespace to be deleted or in its dependents.
- For each A_i (i=1,…,k), find the minimal (most specific) class(es) which will not be deleted (i.e., they are not parts of the deleted namespace or its dependents), let's say $B_{i1},\ldots,B_{im}$.
- Reclassify &a under B_{ij}.
- Remove the classification links between &a and A_i (i=1,…,k).

As an example of this process, consider Figure 1. The deletion of namespace ns1 will cause the deletion of classes ns1#A, ns1#D. If the user asks for a reclassification of the data resources, then the resource &a, originally classified under ns1#D and ns3#C will now be reclassified under ns2#B, ns3#E and ns3#C. The classification links to ns2#B and ns3#E will be created because ns2#B and ns3#E are both minimal superclasses of ns1#D that do not belong in the namespace(s) to be deleted (ns1 in this case). The classification link to ns3#C will persist, because ns3#C is not affected by the deletion.

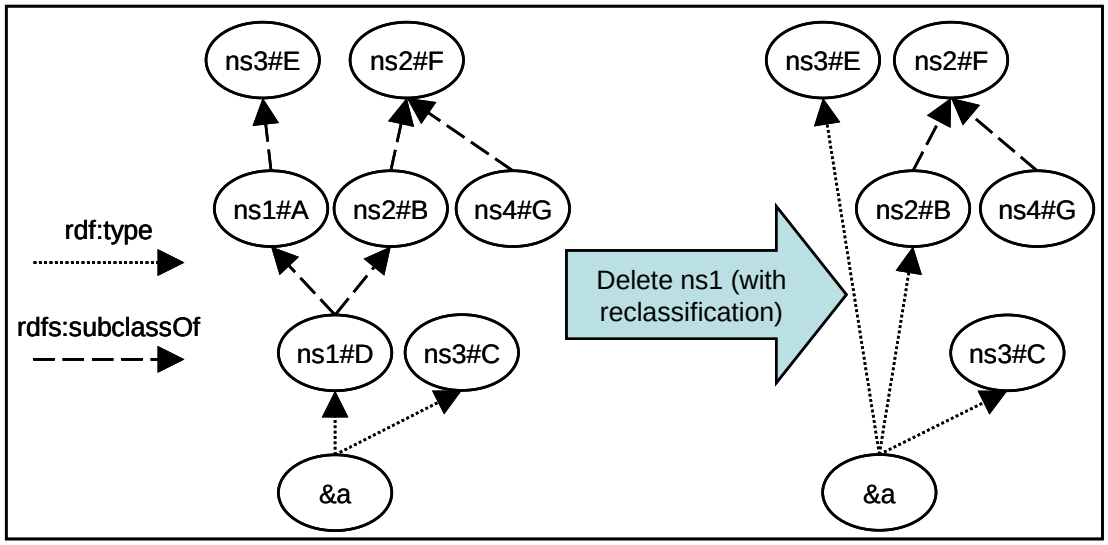

Figure 1: Reclassification in Delete

The same process as with class instances is also used for property instances which can be reclassified in the same manner. Also, note that class instance reclassification may

affect property instances as well: the deletion of a class instance causes the deletion of all associated property instances, whereas the reclassification of a class instance causes the reclassification (if possible) or deletion (if reclassification is not possible) of the associated property instances.

Note that this reclassification semantics is identical to the semantics used in RUL [MSCK05] (which is employed by the Update Service): when the deletion of a classification link is requested in the Update Service (RUL), a reclassification like the one described above takes place. In fact, this is how the Delete Service implements the reclassification process (i.e., through calls to the Update Service). The reader is referred to [MSCK05] for further details on the process.

Summarizing the above we could say that, depending on the user's selections, we have the following different modes of operation for the Delete Service:

1. *Soft Delete*: Under that mode, the Delete Service will fail if there are any namespaces or data triples which are depending on the namespace to delete. In any other case, the input namespace is deleted, along with its contents and any references to it. In the example of Figure 1, the deletion (under "soft mode") of ns4 will succeed (and will delete ns4#G and the IsA link of ns4#G to ns2#F), but any other deletion would fail (under "soft mode").
2. *Hard Delete with Reclassification*: Under that mode, the Delete Service will delete the namespace in its input, along with all its dependent namespaces (if any). All data classified under the deleted namespace (or its dependents) will be reclassified as described above. In Figure 1, one can see the effects of this mode of operation when deleting ns1. If, in the same figure, we were asked to delete ns3, on the other hand, then the operation would cause the deletion of both ns1 and ns3, as well as the reclassification of &a under ns2#B only.
3. *Hard Delete*: Under that mode, the Delete Service will delete the namespace in its input, along with all its dependent namespaces and data triples (if any). If this deletion leaves any resource unclassified, the resource is deleted altogether from the database. In the case of Figure 1, the deletion of ns3 would cause the deletion of ns1 and the deletion of the resource &a (along with all its classification links); on the other hand, the deletion of ns2 would cause the deletion of ns1 and ns4, but the resource &a would persist, and would be classified under ns3#C only.

The input to the Delete Service is the namespace to delete, along with some parameters indicating the "mode" of the operation; the output is a boolean flag indicating success or failure of the operation. More details on the related methods' signatures and the implementation of the service appear in the next section.

The Delete Service works exclusively on the persistent storage level, which means that the service does not use the Main Memory API and does not load the deleted namespace in the main memory. The service is implemented by executing adequate SQL update operations (DELETE) upon the database, making sure that all the contents of the deleted namespace(s), as well as all the references to them (e.g., dependency information) are removed.

4.1.2 Modularization Using Named Graphs

Named Graphs

In the relevant literature [CBHS05], the proposed solution to the modularization problem outlined in previous sections is the introduction of *named graphs*, which allow the decomposition of an ontology or RDF KB into logical modules. A named graph can be simply viewed as a *container of triples*, to which we have assigned a name (URI). Using this URI, we can refer to named graphs using RDF/S triples, just like we do for all types of resources. Formally, a named graph can be modelled as an assignment (function) of sets of triples to URIs (names) [Ped08].

Applying this line of thinking in our context, we will associate each named graph with one logical module in the RDF/S KB, so each module would correspond to one named graph. The concept of named graphs provides a lot of flexibility in the offered modularization for several reasons. First of all, a named graph can contain both data and schema triples. Secondly, named graphs are focused on triples, unlike namespaces whose focus is on names and nodes of the RDF/S graph. Furthermore, there is no restriction as to the number of triples contained in a named graph: it could contain any number of triples, or no triples at all. Similarly, there is no restriction as to the assignment of triples to named graphs: one specific triple could belong to one named graph, many named graphs, or no named graphs at all. Finally, the association of each named graph with a URI allows us to refer to the named graph as a whole in RDF triples, thus being able to set metadata and other information on the named graph itself. The flexibility offered by named graphs and their ability to modularize RDF/S graphs has already been exploited in the literature, especially in the context of provenance tracking and recording [CBHS05], [WN06].

Most of the named graphs (modules) that appear in the system are defined by the user (through the use of adequate RUL statements of the extended RUL presented in this deliverable). However, there is one special named graph, the DEFAULT# named graph, which is not user-defined, but system-defined, and has been introduced for backwards compatibility purposes. This named graph can be queried, updated and accessed, just like any other (user-defined) named graph. By default, it is assumed to contain all triples that have not been explicitly assigned to any particular named graph (thus, a triple belonging to “no” named graph is actually a triple belonging to the DEFAULT# named graph). Unlike user-defined named graphs, the #DEFAULT named graph exists by default in the knowledge repository and cannot be removed.

Co-ownership and Graphsets

The introduction of named graphs in our model has a number of implications, the most important one being the fact that they force us to introduce a new and more general concept (the *graphset*) in order to be able to fully exploit the modularity offered by named graphs. The need for graphsets stems from the need for explicit “shared assignment” of a triple to a set of named graphs, whose semantics is that the triple does not belong to any of the named graphs of the set in isolation, but to all named graphs of the set at the same time, in a state of *co-ownership* (see [Ped08]). This state of co-ownership appears mainly due to RDFS *inference*, but the real need that forces us to introduce graphsets stems from the fact that RDF/S graphs are *updatable*, *dynamic entities*, and our update semantics requires that inferred knowledge is a “first-class citizen” that needs to be retained, if possible, after the update.

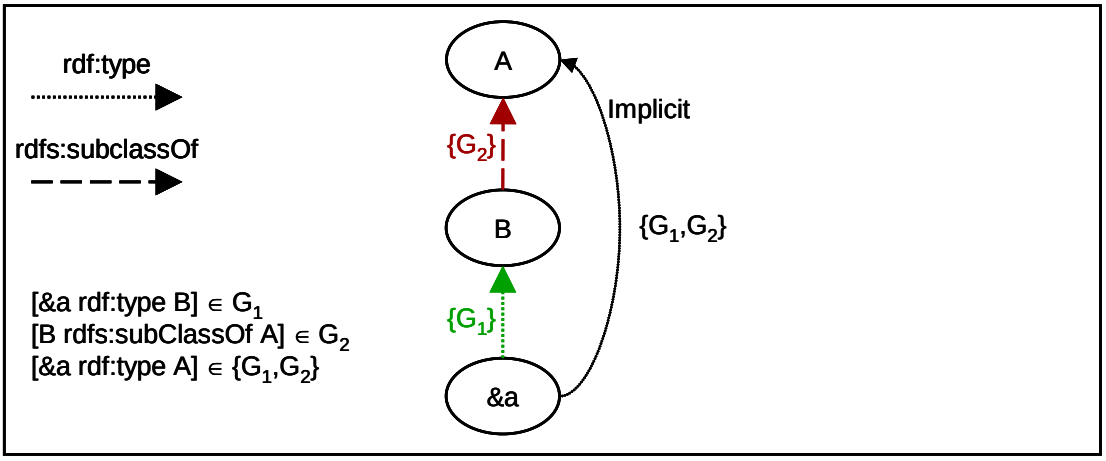

Figure 2: Graphsets in Inference

The problems that arise due to the inference mechanism are illustrated in Figure 2. In that figure, it would be a mistake to classify the (implicit) triple [&a rdf:type A] into any single named graph, because its generation is based on the existence of triples in both G_1 and G_2. The only acceptable solution is to claim that this triple *belongs to both named graphs* (G_1 and G_2) *in a shared fashion*. This "shared ownership" is represented using the notion of a graphset, which is a *set of named graphs* (namely, the set $\{G_1,G_2\}$ in our example) that the triple belongs to.

It should be emphasized that a triple belonging to G_1 and/or G_2 is different than belonging to $\{G_1,G_2\}$. In the former case, the named graphs G_1 and/or G_2 contain said triple, whereas in the latter the triple does not belong to any of G_1 and G_2 independently, but only to $\{G_1,G_2\}$ in a shared fashion. Therefore, saying that a triple belongs to multiple named graphs (or graphsets) means that a triple belongs to each of those named graphs (or graphsets) in an independent manner, whereas saying that a triple belongs to certain named graphs in a joint fashion means that it belongs to the graphset formed by those named graphs (and only in this graphset, i.e., it does not belong to the named graphs themselves). Note that the DEFAULT# named graph can also be used to form graphsets.

The need for the introduction of graphsets presents itself even more emphatically when updates are considered. During updates (in particular, deletions), we are often faced with the need to introduce in an explicit manner triples which were originally implicit. This is due to the semantics of RUL updates [MSCK05] and the fact that as much as possible of the original implicit knowledge is retained during an update. Whenever such a situation arises, we must make sure that the newly introduced triples are assigned to the correct named graphs (or graphsets).

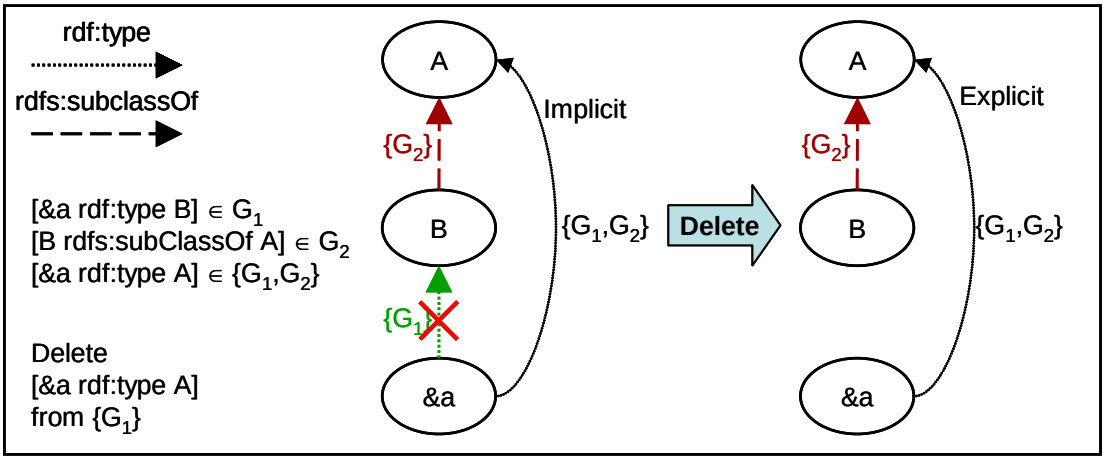

Figure 3: Graphsets in Change

In the example of Figure 3, the newly introduced (originally implicit) triple [&a rdf:type A] should be assigned to the graphset $\{G_1,G_2\}$, as described above. The introduction of this triple is due to RUL semantics [MSCK05], per which, implicit triples which will lose their support due to a deletion operation are explicitly introduced in the RDF graph. The assignment of the explicitly introduced triple to $\{G_1,G_2\}$ is due to the fact that both named graphs "cooperated" in the inference of the original triple [&a rdf:type A]. Note that this assignment means that graphsets do not only contain implicit triples (as in the case of Figure 2), but may contain explicit triples as well. This feature is extremely important, as it allows the user to *explicitly declare certain triples to belong to graphsets*, in effect stating that said triples belong to several named graphs in a shared fashion, rather than to any individual named graph.

It can be easily inferred by the above that a graphset containing just one named graph (e.g., $\{G_1\}$) is in fact the same as the named graph G_1. Therefore, a named graph can be seen as a special type of graphset. For ease of presentation, in the following we will only refer to graphsets, assuming that named graphs are the graphsets whose definition contains just one named graph.

Further, note that the modularization flexibility exhibited by named graphs is also extended to graphsets. More specifically, a graphset may contain any number of triples, even zero, whereas a triple may belong to any number of graphsets (including individual named graphs), without any limitations in this respect. The only limitation of graphsets (with respect to named graphs) is that they are not assigned a URI, only an internal ID; thus, one cannot directly refer to a graphset in an RDF/S triple (e.g., in order to assign metadata information to the graphset). However, graphsets can be referred to indirectly, via their constituent named graphs.

A further implication of these facts is that graphsets are first-class citizens in our model, i.e., they are considered of equal value as named graphs, since they can be assigned triples in an explicit manner, just like we did with named graphs in the simplified model. Also, the notion of named graphs is still supported, because a named graph G is identified with the graphset {G}, i.e., a named graph can be considered as a special type of graphset. For more details on the semantics of graphsets and their properties, the reader is referred to [Ped08]. In the same work, the

interested reader may find the details of the formalization underlying the definition of graphsets (and named graphs).

Named Graphs and Graphsets in SWKM

The introduction of graphsets in our model solves the problem of modularization of RDF/S graphs. Nevertheless, being able to fully exploit the functionality offered by graphsets requires their full support in the entire knowledge management process; such a support should include support at the level of the serialization of RDF/S graphs with graphset information, as well as extensions of the existing main memory and database representations to capture and store graphset information, and should also involve all the existing SWKM services, whose semantics are obviously affected by the enrichment of the standard RDF/S model with the new notion.

The present deliverable concerns the support currently integrated (and implemented) in the Query (RQL) and Update (RUL) services, as described in [DoW3.2]. However, a few notes on the level of support provided by the other services and components of SWKM are in order.

At the level of the serialization of RDF/S graphs, there is currently no well-defined (or accepted) standard supporting graphset information. On the other hand, the TRIG format, which is supported by the SWKM platform, inherently supports the serialization of named graphs (the RDF/XML format does not). Thus, if we want to serialize an RDF Graph that contains named graph information, this can only be done using the TRIG format.

At the level of the database representation, graphset support has been implemented as an optional feature in the HYBRID representation, meaning that the new persistent storage is backwards compatible with the older services (which didn't support graphset information). The main memory model, on the other hand, does not support graphsets.

The Import and Export services of the Knowledge Repository can easily be upgraded to support graphsets; note however that the implementation of such services presupposes the existence of a serialization format that supports graphset information, which is currently not available. The support for importing and exporting named graphs (which is supported by the TRIG serialization) has been included in the current versions of the Import and Export Services.

Finally, as far as the rest of the SWKM services are concerned (Knowledge Mediator, Knowledge MatchMaker, Knowledge Synthesizer, Analytical and Knowledge Mining Services) there has been no provision, at the moment, for supporting graphsets. In fact, the introduction of graphsets in such services would present several technical difficulties and would introduce changes in their semantics. In addition, at the moment, there does not seem to exist the need for such a support in any of the Working Knots.

On the other hand, the Query (RQL) and Update (RUL) services have been extended in order to fully support querying and updating RDF/S graphs with graphset information. Such a support is far from trivial, as it involves certain non-trivial extensions in both the semantics and the syntax of the query and update languages (RQL/RUL). Moreover, the extended query and update languages (RQL/RUL) should

be backwards compatible with their older versions, and should, at the same time, support querying and updating graphset information, as well as triples in RDF/S graphs that contain graphset information. The details of the syntax of the new query and update languages (RQL/RUL), as well as the full details on their semantics, can be found at [Ped08]; here, we will give an overview of the new functionalities and some basic explanation of their semantics.

Note that the introduction of support for graphsets in the existing Query and Update services does not affect their reliability or performance; queries and updates that do not involve named graphs would be interpreted and executed using the original query and update algorithms and would return the same results (since the services are backwards compatible). However, the new, enhanced versions of the services allow the execution of more complicated queries and updates (i.e., involving graphset information).

Querying Named Graphs and Graphsets

At the level of queries, the introduction of graphset information allows the execution of more sophisticated and complicated queries. In the original RQL, one could ask for the triples (or nodes) that satisfy a certain property; in the new RQL, one can ask for the triples, nodes, or graphsets that satisfy a certain property. The property which the triple, node, or graphset, is required to satisfy may be determined taking into account graphset information.

The above flexibility allows the user to ask for the information contained in a specific graphset or which graphset contains a specific piece of information; the answers to both queries can be filtered using filtering conditions regarding the triples and/or the graphsets involved. Thus, the new functionality allows querying for different things, such as: the graphset(s) in which specific triple(s) belong to; the graphset(s) that satisfy a certain property (e.g., containing a triple); the existing named graphs in the system; the triples that belong in certain graphset(s); the triples or nodes that satisfy a certain property; the triples or nodes that satisfy a certain property within one or more graphsets, or within the graphsets that satisfy a given property; or the triples or nodes that satisfy a certain property, as well as the graphsets that participate in the satisfaction of said property. For backwards compatibility purposes, when no graphset information is provided in a query, then all graphsets are considered by default (note that this is the behaviour of the original RQL, where graphsets are not supported).

The above types of queries can be also combined to form more complicated queries. Some examples of queries, their syntax and their expected response by the system for the RDF KB shown in Figure 4 can be found in the following table. For more details on the extended RQL, refer to [Ped08].

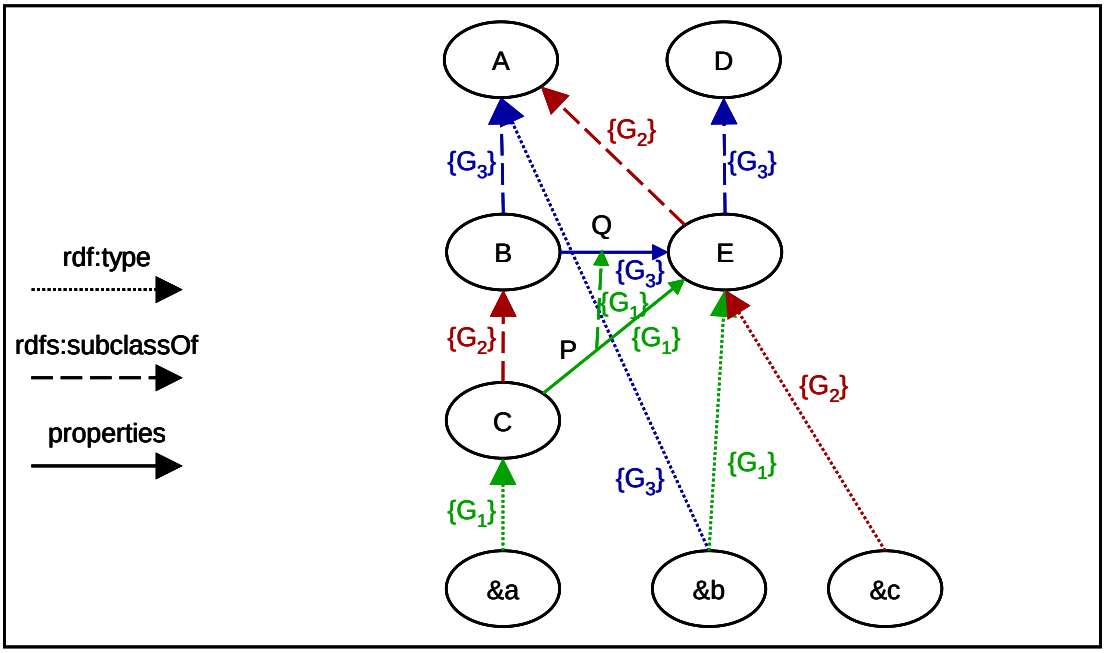

Figure 4: Querying with Graphsets

Description	Syntax	Expected Result (Figure 4)
Find all graphsets that define &b as an instance of A	SELECT g FROM g::A{x} WHERE x=&b	$\{G_1,G_2\}$ $\{G_3\}$
Find all instances of A in {G2}	SELECT x FROM g::A{x} USING NAMEDGRAPH g=&G2	&c
Find all graphsets in which C is a subclass of A	SELECT g FROM g::A{;$X} WHERE $X=C	$\{G_2,G_3\}$
Find all subclasses of A and the graphsets in which they are defined	SELECT $X,g FROM g::$X WHERE $X < A	B,$\{G_3\}$ C,$\{G_2,G_3\}$ E,$\{G_2\}$
Find all subproperty relations and the graphsets they belong to	SELECT @X,@Y,g FROM g::@X;@Y	P,Q,$\{G_1\}$

Table 2: Examples of RQL Queries

Let us suppose that the RDF/S graph in Figure 4 was developed by a learner in order to record the information found in different sources (books, web pages, other co-learners, etc) as well as the source of each information. Then, each named graph in the figure would represent one source of information. Using the above queries, the learner would be able to find all the sources that support some given information (query #1, #3), or the information of a certain form that can be found in a given source (query

#2), or the information of a certain form that can be found in any source, as well as the source that supports each such information (queries #4, #5).

Updating Named Graphs and Graphsets

At the level of updates, the new feature is that triples can now be added to (and deleted from) graphsets. Note that the new, extended RUL (and, consequently, the new Update Service) only supports the updating (adding/removing) of data triples, just like the original RUL (and Update Service).

It is easier in the analysis that follows to view the new RUL as dealing with *quadruples*, i.e., triples associated with graphset information. As in the original RUL, the full expressive power and patterns of RQL can be used to specify the triple(s) to be added and/or deleted to/from the respective graphset(s), i.e., the quadruples to add and/or delete; this feature guarantees the flexibility of the language in terms of being able to specify with accuracy the quadruple(s) to be added/deleted. The semantics of executing an update operation are complicated by the fact that more housekeeping is required in order to determine the side-effects and specify the triples' assignments to graphsets following an update. An additional feature of the new RUL is that we can manipulate named graphs and graphsets, by allowing additions and deletions of entire named graphs from the repository.

More specifically, the addition of a new data triple, associated with a graphset, proceeds, as usual, by adding the new triple associated with the graphset that the RUL statement requires; if no graphset information is specified, then the DEFAULT# graph is assumed; if the quadruple exists already (i.e., the required triple exists and is associated with the graphset that the insert statement requires), then the operation is void and no insertion takes place.

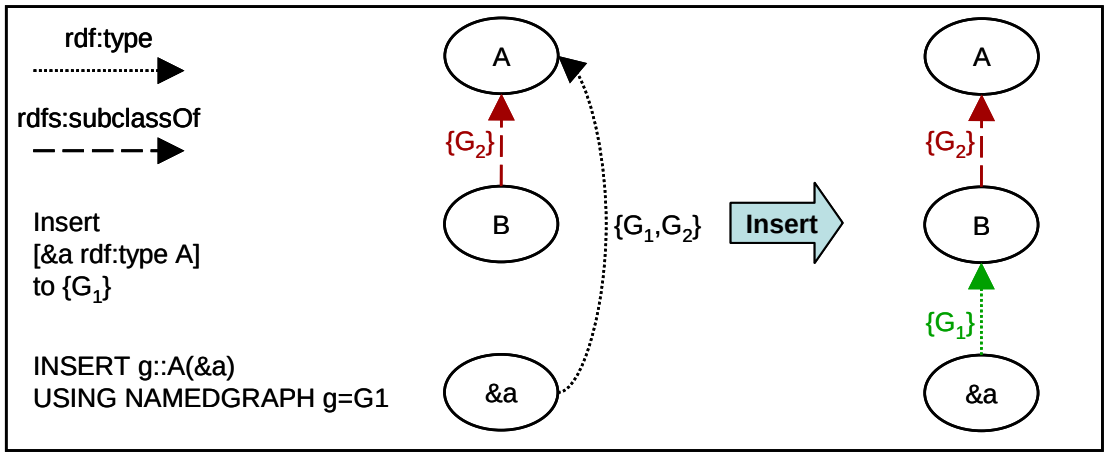

Figure 5: Insert Operation (1)

Following the addition of the quadruple, redundancy elimination takes place, as usual, the only difference being that redundancy elimination should now take into account the graphset information as well (i.e., it works on the level of quadruples, rather than the level of triples). For example, in Figure 5, the triple [&a rdf:type A] is redundant, because it belongs to $\{G_1,G_2\}$, whereas in Figure 6, the same triple belongs to $\{G_1\}$, so it is not redundant and is kept after the update.

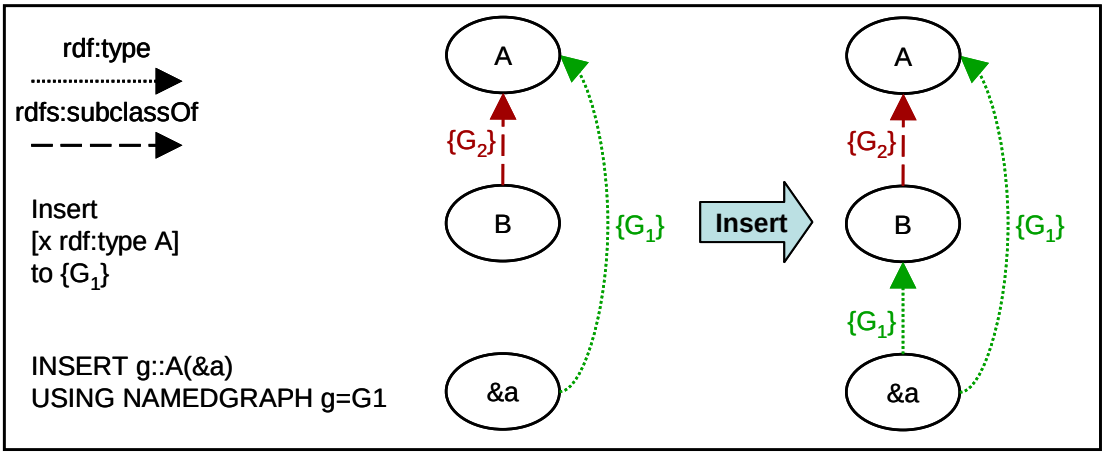

Figure 6: Insert Operation (2)

The deletion of a quadruple is more complicated. Unlike insertion, deleting a triple without specifying the graphset from which to delete it implies that the triple is deleted from all the graphsets that it appears in; if the triple does not appear in any graphset, the operation is void. Following the deletion of the triple from the graphset(s) as required, the following actions must be taken:

1. Verify that the about-to-be-deleted quadruple will not re-emerge as an implication of the remaining quadruples. If this is the case, then remove all data quadruples participating in the implication of the about-to-be-deleted quadruple. Note that, again, graphset information plays a critical role; for example, in the operation shown in Figure 7, the triple [&a rdf:type C], belonging in $\{G_1\}$, must persist, because it is not involved in any implication causing the generation of the deleted triple ([&a rdf:type B] in $\{G_1\}$); this is not the case in Figure 8, where the operation is different, and said triple must be removed.
2. All (implicit) triples that are implied by the about-to-be-deleted quadruples, or by the quadruples detected in step 1, must persist, unless they are included in the list of triples to be deleted that was identified by the delete statement itself, or in step 1. For this reason, we must explicitly add such triples, carefully assigning them the correct graphset information. For example, in Figure 7, we add the triple [&a rdf:type A] in $\{G_1,G_3\}$; in Figure 8, we add the triple [&a rdf:type A] in $\{G_1,G_2,G_3\}$, as well as the triple [&a rdf:type D] in $\{G_1,G_3\}$ (the latter is caused by the removal of [&a rdf:type C] from $\{G_1\}$).
3. After all class instantiations have been fixed, per steps 1 and 2, we must make sure that all explicit or implicit property instances are deleted or kept as necessary. In particular, each *explicit property instance* which is originating from (or leading to) one of the instances whose instantiation is affected by steps 1, 2, is checked for validity; if its source (or target) is not correctly classified (explicitly, or implicitly, and regardless of the graphset information) under the domain (or range) of the respective schema property, then the property instance is removed. Moreover, each *implicit property instance* which is originating from (or leading to) one of the instances whose instantiation is affected by steps 1, 2, is checked for validity; if its source (or target) is correctly classified (explicitly, or implicitly, and regardless of the graphset information) under the domain (or range) of the respective schema property, then the property instance should be kept, so, given that it is currently implicit, it is added (unless its addition would cause a redundancy).

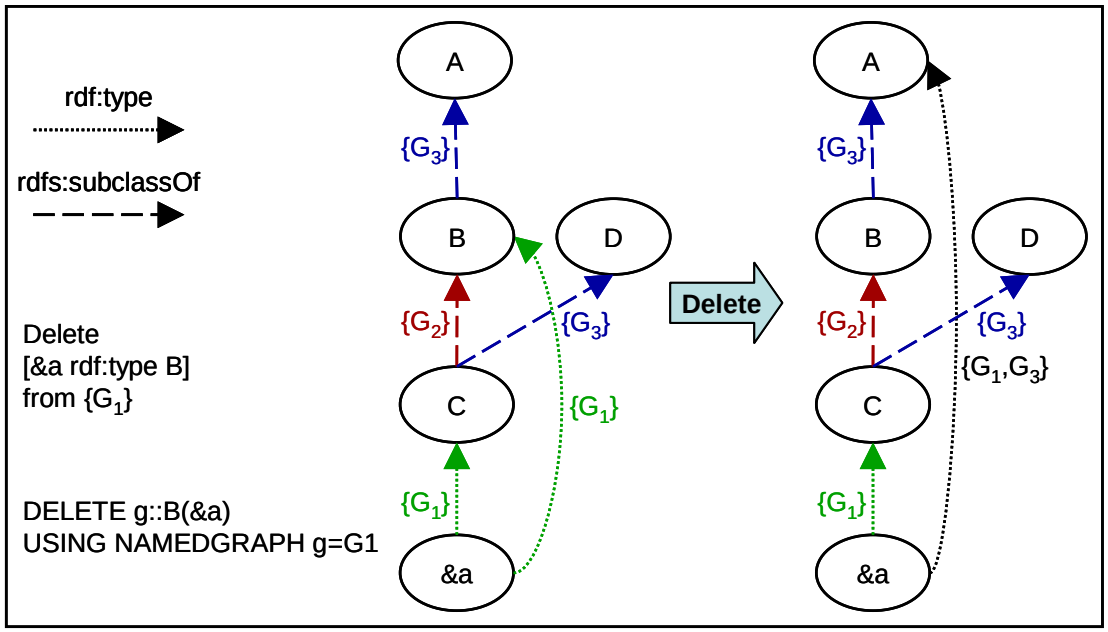

Figure 7: Delete Operation (1)

The above steps (especially #3) are quite complicated; the general idea is the same as in the original RUL, however, the critical difference that complicates the problem is the fact that we are now dealing with quadruples, rather than triples. The reader is referred to [MSCK05], [Ped08] for details on the semantics of the original and the extended RUL respectively.

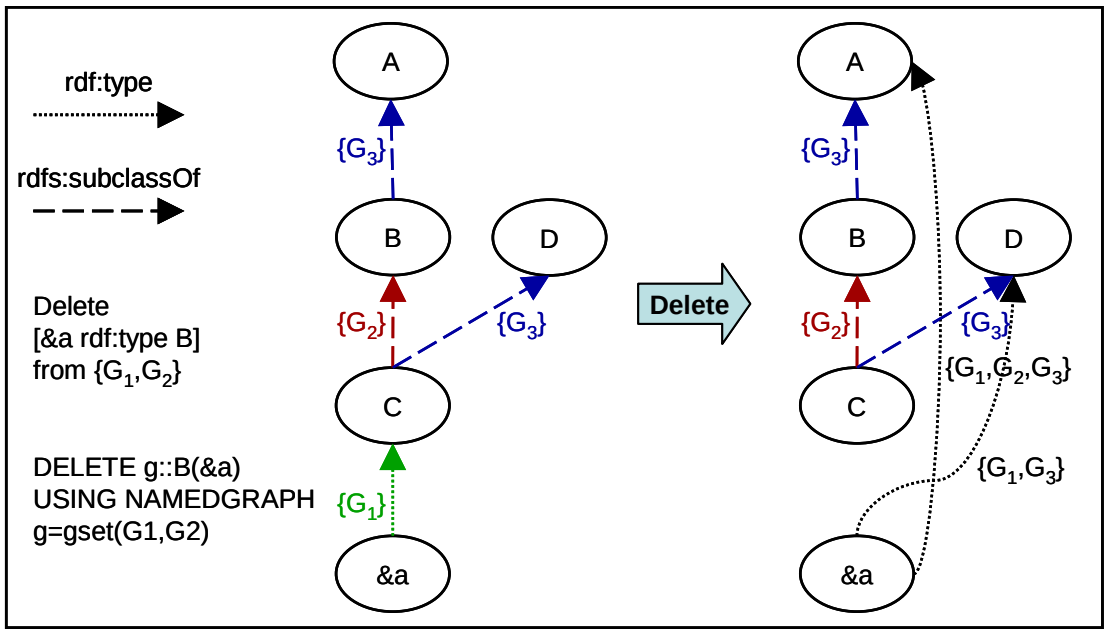

Figure 8: Delete Operation (2)

As already mentioned, the new RUL allows us also to explicitly manipulate named graph and graphset information. In particular, it is possible to:

1. *Insert a new named graph*, whose semantics is that a new, empty named graph (and the respective graphsets) are added in the repository.

2. *Delete all data contents of a named graph*, whose semantics is that all data triples in the given named graph are deleted. Optionally, one can also delete the data contents (i.e., all the data triples) in all the graphsets in which the given named graph participates in.
3. *Delete an empty named graph from the system*, whose semantics is that a named graph, and all graphsets that contain said named graph, are removed from the system. Note that this operation will fail if the named graph itself, or any of the graphsets that include it, contain any triples (data or schema).
4. *Force delete a named graph*, whose semantics is that the data contents (i.e., the data triples) of the named graph and the graphsets that it participates in are deleted (like in operation #2), and then, the named graph itself and the graphsets that it participates in are deleted, if empty (like operation #3). Note that this operation will fail (roll-back) if the named graph itself, or any of the graphsets that include it, contain any schema triples.

It should be noted that the above operations can be applied only to user-defined named graphs; they cannot be applied on the system-defined #DEFAULT named graph.

The above operations (insert, delete, operations on named graphs) can be arbitrarily combined forming more complicated update requests to be handled by the Update Service. Furthermore, insert and delete operations that were supported by the original RUL are supported by the enhanced version too; however, replace operations with graphsets have not been defined, even though replace operations without graphset information are still supported.

In the aforementioned example of the learner who records some information on a domain and the related sources, the updating features would allow, for example, the insertion of a piece of information (data triple) along with its related sources (i.e., the sources where the information was found in), or the removal of the association of a certain piece of information (data triple) with specific sources (meaning that we no longer believe that the particular piece of information was provided by the particular sources), or the removal of a certain piece of information (data triple) from the RDF/S graph entirely (irrespectively of the source associations). In addition, the operations on graphsets allow the manipulation of sources, such as the introduction of a new source (as a preparation for adding information related to that source), or the deletion of some source (which could be empty, or could be deleted along with its data contents, if not empty).

Implications from the Introduction of Named Graphs

As already mentioned, the introduction of named graphs provides the user with a lot of powerful features and allows a very flexible modularization of RDF/S graphs. However, named graphs come also with a number of implications, outlined here.

First of all, the notion of *"ownership"* that presents itself once modules are introduced, also causes the need for *"co-ownership"*. This need is due to the RDFS inference mechanism (which causes implicit triples to be "co-owned"), as well as by the RUL semantics (which causes explicit triples to be "co-owned"). Apart from necessary, this feature is also very useful, because it allows us also to state explicitly that some triple is "co-owned" by more than one named graphs. In order to support

this feature, the notion of graphsets was introduced as a first-class citizen in our model.

The second major implication of the introduction of named graphs (and, consequently, graphsets) is that the user should be able to query and update them. Both features require the expansion of the existing RQL and RUL languages, which is described in detail in [Ped08] (and in this deliverable).

More specifically, querying in the presence of graphsets means that a new, and more powerful version of the RQL language (and the Query Service) should be introduced in order to allow querying graphsets (and named graphs), as well as triples that contain graphset (and named graph) information. During querying, graphset information should be taken into account, and the query language should support the feature of being able to detect those named graphs (or graphsets) that contributed in determining the query answer.

Updating in the presence of graphsets is even more difficult. A new RUL (and Update Service) is required, which, as expected, builds upon the new RQL language. The new RUL should support the updating of graphset (and named graph) information, as well as the updating of triple information in the presence of graphsets (and named graphs). The main complications that arise in the specifications of the semantics of the new RUL stem from the fact that inference and redundancy elimination in the presence of graphsets is more complicated and takes into account where (i.e., in which graphsets) the involved triples are assigned to.

A further implication from the introduction of named graphs is related to their serialization; at the moment the TRIG format (only) supports the serialization of named graphs (but not of graphsets).

Finally, fully exploiting the modularization power offered by named graphs (and graphsets) implies that all SWKM services should support them. At the moment, we have provided support for named graphs (and graphsets) at the level of the knowledge repository (persistent storage). Support at the level of the main memory model is pending. The support of named graphs by the Knowledge Mediator and Knowledge MatchMaker services is inherently difficult and will be addressed in the future, depending on whether such a need will be expressed by the partners in the context of some Working Knot. Support for named graphs in the Import and Export services of the Knowledge Repository is relatively easy (and has been implemented), but support for graphsets presupposes the existence of a well-defined serialization format that supports graphsets (which is not available at the moment). Finally, support at the level of the Query and Update services has been already integrated, as described above. This deliverable only covered the support of named graphs by the Query and Update services, as described in the latest DoW [DoW3.2].

4.2 Knowledge Mediator

4.2.1 Persistent Comparison Service

The specifications and general design of the Persistent Comparison Service is, in most respects, similar to the design of the main memory version of the service that was

described in detail in [D5.3]. Here, we will repeat the main points from [D5.3], outlining the differences between the two versions of the service. The reader is referred to [D5.3] and [ZTC07] for more details.

The Persistent Comparison Service is responsible for comparing two collections of namespaces already stored in the repository and computing their delta in an appropriate form. Unlike the Main Memory Comparison Service, the comparison is made directly on the repository, without an intermediate phase of loading the two collections of namespaces into the main memory. The result of the comparison is a "delta" (or "diff") containing the differences between the two collections of namespaces, i.e., the change(s) that should be applied upon the first in order to get to the second (see Figure 9 for an example).

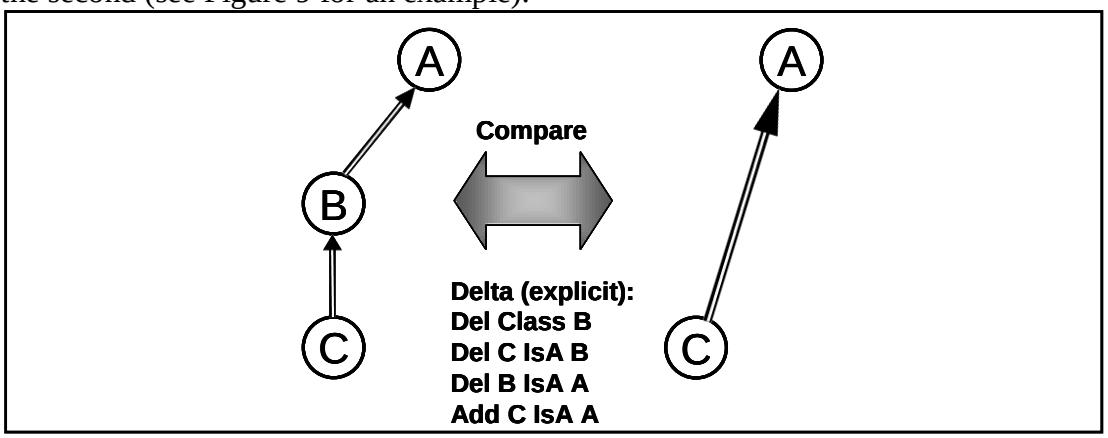

Figure 9: Comparing Two Namespaces

The comparison is based on semantic, rather than syntactic considerations (see [D5.3]), so our service is based on the comparison of the triples contained in the namespaces. All four of the different methods for computing a semantic delta between namespaces that were discussed in [ZTC07] and implemented in the Main Memory Comparison Service [D5.3] are implemented in the Persistent Comparison Service as well. More specifically, depending on whether the implicit knowledge (i.e., the inferred triples) contained in the two collections of namespaces is, or is not, taken into account, we have the following four modes of operation:

- **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 into account both explicit and 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 set of triples (namespace or collection of namespaces) K, i.e., all the triples that exist explicitly or implicitly in K.

The output of the Persistent Comparison Service in each of the different modes of operation (deltas) is identical to the output of the main memory version of the service for the respective mode (delta). Therefore, all the observations and comments made in [D5.3] regarding the property of *delta correctness*, the *size* of the various deltas and the *different update semantics* that could be used to apply a delta, hold for the persistent version of the service as well.

Just like its main memory counterpart, the input to the Persistent Comparison Service is two collections of namespaces for comparison, and the delta function that should be used for the comparison. The dependent namespaces of the compared ones (in the input) are also considered in the comparison in both versions of the service (see [D5.3] for details).

The format of the output of the service is identical to the one produced by the main memory Comparison Service, namely, a pair of strings that represent the delta of the two (collections of) namespaces. In particular, the first string of the pair represents the RDF triples that exist in the second collection of namespaces but don't exist in the first, whereas the second represents the triples that exist in the first but not in the second. The serialization of said triples in the output string is done using the TRIG format.

5 Implementation

5.1 Overview and Preliminaries

The general architectural decisions related to the SWKM platform have not changed since the latest M24 release (V2.0), and are described in detail in [D5.4]. In short, the services are being deployed as web services, which use an RDBMS server as a backend and the SWKM client as an API for contacting them. Here, we will describe the (few) parts of the architecture that have changed since the M24 release, namely the enhancements related to the SWKM client and the new automatic installation tool that we developed for the SWKM platform.

5.1.1 The SWKM Client

As an aid for contacting the various SWKM services, an SWKM client is provided, which is a collection of java classes and interfaces that can be used to contact the SWKM web services in a concise and natural way. A `client` instance is an access point to several interfaces in "gr.forth.ics.rdfsuite.services", which group the operations of each web service; these are the following:

- Importer accessed by `client.importer()`
- Exporter accessed by `client.exporter()`
- QueryHandler accessed by `client.query()`
- UpdateHandler accessed by `client.update()`
- DiffGenerator accessed by `client.diffGenerator()`
- ChangeImpact accessed by `client.changeImpact()`
- VersionManager accessed by `client.versionManager()`
- Registry accessed by `client.registry()`
- Delete accessed by `client.deleter()`

- Debug accessed by `client.debug()`

Note that there have been no changes with respect to the existing services. The new Delete Service is being accessed through a dedicated interface (`client.deleter`). The new Persistent Comparison Service has not been associated with a dedicated interface, but shares the interface of the Main Memory Comparison Service; a boolean parameter is used to determine whether the Main Memory Comparison Service or its persistent counterpart should be used (`false` for the main memory version, `true` for the persistent one). Note that, the determination of the version to use is made during the initialization of the DiffGenerator; thus, if the user wants to use both versions, he should initialize two instances of DiffGenerator. Finally, the new versions of the Query and Update service are supported using the same interfaces as the original ones; note that the changes needed to support the named graphs functionality are at a lower level, namely at the level of the RQL and RUL languages, thus no changes are required at the level of the function calls.

Each web service is modeled as a Java interface. These interfaces reside in `swkm-services-api.jar` , in the package "gr.forth.ics.rdfsuite.services". The mapping of interface names and services is given below:

Web Services	WSDL URL Paths	Interfaces
Query Service	/query?wsdl	QueryHandler
Update Service	/update?wsdl	UpdateHandler
Import Service	/importer?wsdl	Importer
Export Service	/exporter?wsdl	Exporter
Versioning Service	/versioning?wsdl	VersionManager
Comparison Service	/diffGenerator?wsdl	DiffGenerator
Change Impact Service	/changeImpact?wsdl	ChangeImpactAnalyzer
Registry Service	/registry?wsdl	Registry
Delete Service	/deleter?wsdl	Deleter

Table 3: Web Services and Interfaces

As before, the only new interface required (with respect to the M24 release) is for the Delete Service; the Persistent Comparison Service shares the interface of the Main Memory Comparison Service, whereas the new, enhanced versions of the Query and Update services use the interfaces of the original version.

5.1.2 Installation and Configuration

Details on the configuration, installation, optimization etc of the SWKM platform can be found at [D5.4]. A new feature, with respect to [D5.4], is the development of an easy-to-use, automatic installer that can be used in order to avoid the cumbersome installation process described in [D5.4]; this installer can be found at http://athena.ics.forth.gr:9090/SWKM.

The installer can be used to setup SWKM itself, as well as all the related services and applications that SWKM depends on (such as glassfish, postgres, etc), as necessary, depending on the services existing in the underlying system. There are two versions of the installer (both can be found at the aforementioned URL, http://athena.ics.forth.gr:9090/SWKM): the first version downloads the entire bundle

locally and executes the installation from there, whereas the second downloads only the basic executable and the rest is downloaded during the installation on a need-to-have basis. The reader is referred to http://athena.ics.forth.gr:9090/SWKM for further details on the installer.

5.2 Knowledge Repository

5.2.1 Delete Service

Signature

- `java.lang.Boolean delete(`
  `java.lang.String uri)`
- `java.lang.Boolean deleteWithDependents(`
  `java.lang.String uri,`
  `java.lang.Boolean data)`

Description

The purpose of these operations is to delete a namespace from the repository (parameter `uri`). The `delete` method corresponds to the "soft delete" mode of operation; depending on the value of the parameter `data`, the `deleteWithDependents` method corresponds to either the "hard delete with reclassification" (when `data` is `false`) or "hard delete" mode of operation (when `data` is `true`).

Both methods will return `true` if the deletion was completed successfully; they will return `false` if the deletion failed for some reason (indicative causes include: existence of data classified under the namespace to be deleted while in "soft delete" mode, dependent namespaces in "soft delete" mode, database connection problems, non-existence of the URI to be deleted etc).

Preconditions

The namespace with URI `uri` should exist in the repository. For the "soft delete" mode of operation, the service will succeed (return `true`) only if the input URI does not have any dependents or data classified under it.

Effects

After the successful execution of the operation, the triples contained in the deleted namespace no longer exist in the database. Moreover, the URI of the namespace itself is deleted from the list of namespaces in the database, and all references to such a namespace are deleted as well. This is true for all the namespaces that are deleted, and includes either the input namespace only, or the input namespace along with its dependents (depending on the mode of operation). All data classified under the deleted namespaces is either deleted or reclassified, depending on the mode of operation. If the operation fails, none of the deletions is committed and the database remains in its original state. For details on the different modes of operation, refer to section 4 of this deliverable.

5.2.2 Named Graphs

The implementation of the (enhanced) Query and Update services that support graphsets (and named graphs) was based on the implementation of the original Query and Update services. The only changes required are at the level of the individual services' implementation, which should be able to handle the more complex

(enhanced) RQL queries and RUL updates that are necessary to support graphsets (and named graphs).

Therefore, the enhancement of these services does not affect the methods used as contact points for the services; the signatures, descriptions, preconditions and effects of the related methods (`query`, `queryMultiple`, `update`, `updateMultiple`) are identical to those presented in [D5.4] and are omitted from this deliverable.

The only difference between the two versions is that the new services can also handle the enhanced queries and updates related to graphsets (which would fail under the old versions); attempting to execute, for example, a query involving graphsets with the old service would result in a failure to execute the query, as the old version cannot interpret a query that involves graphsets. On the other hand, the new methods are backwards compatible with the old ones, since the queries (and updates) that are not related to graphsets are handled in the same way in the new services.

5.3 Knowledge Mediator

5.3.1 Persistent Comparison Service

The implementation of the interface calling the Persistent Comparison Service was based on the interface that calls the Main Memory Comparison Service; as already mentioned, a boolean variable during the initialization of `DiffGenerator` (of the SWKM client) determines whether the main memory or the persistent version of the service should be used. As a result, the signature, description, preconditions and effects of the related method are very similar to the ones described in [D5.4]. In this section, we describe the (few) differences that have been introduced due to the inclusion of the persistent version of the service; for additional details, the reader is referred to [D5.4].

Signature

```
Delta diff(
java.util.List<java.lang.String> namespacesOrGraphspaces1,
java.util.List<java.lang.String> namespacesOrGraphspaces2,
DeltaFunction deltaFunction)
```

Description

The Comparison Service is responsible for comparing two collections of namespaces already stored in the repository and computing their delta in an appropriate form. The compared RDF KBs (or ontologies) are determined by the parameters `namespacesOrGraphspaces1`, `namespacesOrGraphspaces2` (see [D5.4] for details), whereas the parameter `deltaFunction` determines the delta function to use (see [ZTC07], [D5.3] for details).

The expected input, output, preconditions and behaviour of the algorithm is identical to the ones described in [D5.4], regardless of whether the initialization required the main memory or the persistent version of the service to be used; the two services have been designed so as to produce the same results, so the only thing that is affected is the performance and the scalability of the service, as described in the previous section, because the original version of the service works on the main memory, whereas the persistent version works on the persistent storage.

Preconditions

The preconditions for the service are identical to the preconditions of the Main Memory Comparison Service, outlined in [D5.4].

Effects

None.

6 Conclusion

In this deliverable we described some new services and functionalities which are included in the new M36 release (V3.0) in the KP-Lab project. These services and functionalities are the following:

- The *Delete Service*, which is a new service allowing the deletion of namespaces from the repository. The Delete Service has been implemented as part of the SWKM Knowledge Repository.
- The *named graphs feature*, which allows flexible modularization of the information found in RDF/S KBs. This functionality is expected to be used in various interesting ways within the project. At the present deliverable we only described in detail the support for named graphs that has been integrated in the Query and Update Services of the SWKM Knowledge Repository, per DoW 3.2 [DoW3.2].
- The *Persistent Comparison Service*, which is used to compare, in a scalable way, conceptualizations, in a manner similar to the Main Memory Comparison Service described in [D5.3], [D5.4]. The Persistent Comparison Service, like its main memory counterpart, is part of the SWKM Knowledge Mediator.

We described in a detailed fashion each of these services and functionalities, based on certain motivating scenarios and the subsequent functional requirements. In addition to the abstract description of their functionality, we also gave technical details on their implementation, how they can be accessed, and how each parameter of the related method calls affects the functionality of the respective service or feature.

In addition to those services and functionalities, we developed an enhancement of the streaming capabilities of the existing TRIG parser, as part of our activities related to the Knowledge Repository (see also [DoW3.2]). Under the new implementation, the input TRIG file is read in a streaming manner, thereby reducing the space requirements and improving the performance of the Import Service. This enhancement has been included in the new release, but it does not affect the usage of the service in any way, because it is an internal change.

Finally, it should be noted that the Persistent Change Impact Service (see [DoW3.2]), which was designed to improve the scalability of the original, main memory version of the Change Impact Service [D5.3] by executing the changes directly upon the persistent storage, has not been developed and is not included in this deliverable. The reason is that the problem turned out to be much more difficult than expected and additional work is required for an adequate specification and implementation of the service; in addition, up to now, we did not find requirements for such a service in any of the Working Knots. The development of such a service can be reconsidered later on, if such a requirement appears within the project. Note that persistent updates upon data is already supported using the Update Service [D5.1], whereas main memory updates upon both data and schema are supported using the Change Impact Service [D5.3], [D5.4].

7 Bibliography

[BHLT06] T. Bray, D. Hollander, A. Layman, R. Tobin. Namespaces in XML 1.0 (Second Edition). W3C Recommendation, 2006. Available at: http://www.w3.org/TR/REC-xml-names/

[CBHS05] J. Carroll, C. Bizer, P. Hayes, P. Stickler. Named Graphs, Provenance and Trust. In Proceedings of the 14th International World Wide Web Conference (WWW-05), 2005.

[ColMol] End User Requirements for Collaborative Semantic Modelling. KP-Lab internal document, v.0.6, August 2007.

[D2.4] Driving Objectives and High-level Requirements for KP-Lab Technologies. KP-Lab project Deliverable D2.4, November 2008.

[D4.2.3II] Annex II: "TLO and Ontologies Engineering in the KP-Lab Platform" of the D4.2.3: "KP-Lab Platform Architecture Dossier - Release 3". KP-Lab project Deliverable D4.2.3, Annex II, June 2008.

[D5.1] Specification of the SWKM Architecture (V1.0) and Core Services. KP-Lab project Deliverable D5.3, July 2006.

[D5.3] Specification of the SWKM Knowledge Evolution, Recommendation, and Mining services. KP-Lab project Deliverable D5.3, November 2007.

[D5.4] Prototype (V2.0) of the SWKM Knowledge Mediator, MatchMaker and Manager. KP-Lab project Deliverable D5.4, March 2008.

[D5.6] Specifications for the Knowledge Matchmaker (V.2.0), the Knowledge Synthesizer (V.1.0) and the Analytical and Knowledge Mining Services (V.1.0). KP-Lab project Deliverable D5.6, January 2009.

[D6.6] M33 Specification of End-user Applications. KP-Lab project Deliverable D6.6, December 2008.

[DKKC08] M. Doerr, A. Kritsotaki, D. Kotzinos, V. Christophides. Reference Ontology for Knowledge Creation Processes. KP-Lab Internal Document (currently in draft status), December 2008. Available at: http://www.kp-lab.org/intranet/work-packages/wp4/t4-4-services-management/t4-4-3-creation-and-support-of-semantic-models-based-on-pedagogical-models-created-in-the-project/material-of-the-heraklio-reference-model-workshop/Reference%20Ontology%20for%20Knowledge%20Creation%20Processes.doc/view

[DoW3.2] Description of Work 3.2 Months 25–42. KP-Lab Consortium, July 2008.

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

[MSCK05] M. Magiridou, S. Sahtouris, V. Christophides, M. Koubarakis. RUL: A Declarative Update Language for RDF. In Proceedings of the 4th International Semantic Web Conference (ISWC-05), 2005.

[NCLM06] N. Noy, A. Chugh, W. Liu, M. Musen. A Framework for Ontology Evolution in Collaborative Environments. In Proceedings of the 5th International Semantic Web Conference (ISWC-06), 2006.

[Ped08] P. Pediaditis. Querying and Updating RDF/S Named Graphs. Master thesis, Computer Science Department, University of Crete, 2008.

[PH05] S. Paavola, K. Hakkarainen. The Knowledge Creation Metaphor – An Emergent Epistemological Approach to Learning. In Science Education, 14(6), pages 535-557, 2005.

[SemTag] Specifications for Annotating Knowledge Objects with Semantic Tags. KP-Lab internal document, October 2007. Available at: http://www.kp-lab.org/intranet/design-teams/wk-management-and-analysis-of-complex-knowledge-structures/semantic-tagging/annotating-knowledge-objects-with-semantic-tags/AnnotatingObjectsWithSemanticTags-specifications-v1.doc/view

[Tan07] W.-C. Tan. Provenance in Databases: Past, Current, and Future. Bulletin of the IEEE Computer Society, Technical Committee on Data Engineering, 2007.

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

[TCFKMPS07] Y. Tzitzikas, V. Christophides, G. Flouris, D. Kotzinos, H. Markkanen, D. Plexousakis, N. Spyratos. Emergent Knowledge Artefacts for Supporting Trialogical E-Learning. In International Journal of Web-Based Learning and Teaching Technologies (IJWLTT), 2(3), pages 16-38, 2007.

[WN06] E. Watkins, D. Nicole. Named Graphs as a Mechanism for Reasoning About Provenance. In Frontiers of WWW Research and Development - APWeb, 2006.

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