Solutions and Innovations in Web-Based Technologies for Augmented Learning:
Improved Platforms, Tools, and Applications

Nikos Karacapilidis
University of Patras, Greece

Information Science REFERENCE

INFORMATION SCIENCE REFERENCE

Hershey · New York

Director of Editorial Content: Kristin Klinger
Director of Production: Jennifer Neidig
Managing Editor: Jamie Snavely
Assistant Managing Editor: Carole Coulson
Typesetter: Larissa Vinci
Cover Design: Lisa Tosheff
Printed at: Yurchak Printing Inc.

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue, Suite 200
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

and in the United Kingdom by
Information Science Reference (an imprint of IGI Global)
3 Henrietta Street
Covent Garden
London WC2E 8LU
Tel: 44 20 7240 0856
Fax: 44 20 7379 0609
Web site: http://www.eurospanbookstore.com

Library of Congress Cataloging-in-Publication Data

Solutions and innovations in web-based technologies for augmented learning : improved platforms, tools, and applications / Nikos Karacapilidis, editor.

p. cm. -- (Advances in web-based learning)

Includes bibliographical references and index.

Summary: "This book covers a wide range of the most current research in the development of innovative web-based learning solutions, specifically facilitating and augmenting learning in diverse contemporary organizational settings"--Provided by publisher.

ISBN 978-1-60566-238-1 (hardcover) -- ISBN 978-1-60566-239-8 (ebook)

1. Education--Computer network resources. 2. Internet in education. 3. Organizational learning--Computer assisted instruction. 4. Distance education. I. Karacapilidis, Nikos.

LB1044.87.S619 2009

371.33'44678--dc22

2008023189

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter IX
Supporting Evolution of Knowledge Artifacts in Web Based Learning Environments

Dimitris Kotzinos
Institute of Computer Science, FORTH-ICS and
Department of Geomatics and Surveying, TEI of Serres, Greece

Giorgos Flouris
Institute of Computer Science, FORTH-ICS, Greece

Yannis Tzitzikas
University of Crete and Institute of Computer Science, FORTH-ICS, Greece

Dimitris Andreou
Institute of Computer Science, FORTH-ICS, Greece

Vassilis Christophides
University of Crete and Institute of Computer Science, FORTH-ICS, Greece

ABSTRACT

The development of collaborative e-learning environments that support the evolution of semantically described knowledge artifacts is a challenging task. In this chapter we elaborate on usage scenarios and requirements for environments grounded on learning theories that stress on collaborative knowledge creation activities. Subsequently, we present a comprehensive suite of services, comprising an emerging framework, called Semantic Web Knowledge Middleware (SWKM), that enables the collaborative evolution of both domain abstractions and conceptualizations, and data classified using them. The suite includes advanced services for ontology change, comparison and versioning over a common knowledge repository offering persistent storage and validation.

INTRODUCTION

Classical learning theories are based either on the *knowledge acquisition* metaphor (where a learner individually internalizes a body of knowledge) or on the *social participation* metaphor (where a group of learners collaboratively appropriate a body of knowledge). Although widely accepted, these theories do not sufficiently capture innovative practices of both learning and working with knowledge (i.e., knowledge practices). Only sharing of knowledge in action, i.e. sharing the process of learning itself, is a reliable base for developing a shared cognition (seen both as a group and an individual characteristic).

Knowledge creation activities rely heavily on the use, manipulation and evolution of shared knowledge artifacts externalizing a body of (tacit or explicit) knowledge (Paavola, Lipponen & Hakkarainen, 2004). In order to capture the changing dynamics of community knowledge, team members should be able to update and keep versions of knowledge artifacts, to compare different artifacts or different versions of the same artifact, and to assess the consequences of such an evolution on other community artifacts. Evolution might require to negotiate the meaning of the artifacts at hand and to change the encoded knowledge (i.e., the conceptualization) accordingly, or might require encoding new knowledge regarding a real world object, process or phenomenon at hand.

Shared knowledge artifacts emerge in many collaborative learning and working settings. For instance, a video that records how group members carry out their tasks could be considered as a shared knowledge artifact which the group could annotate (with free text or with respect to an ontology), analyze and further discuss (e.g., for capturing tacit group knowledge). Moreover, and more interestingly, a knowledge artifact could take a more formal substance (e.g. for capturing explicit group knowledge) as in the case of conceptualizations (e.g., a data/knowledge base), or even software code developed within a group. Hereafter, we shall use the term *knowledge artifact* to refer to various forms of conceptual models created and/or shared by a group of learners or knowledge workers (which could be concept maps, annotations, ontologies, etc). Collaborative conceptual modeling, even though not always acknowledged as such, plays an essential role in the knowledge practices of quite many professional communities (e.g., in Communities of Practice: Domingue, Motta, Shum, Vargas-Vera, Kalfoglou & Farnes, 2001) as well as scientific communities, because the respective models provide means to explicate, discuss and scrutinize the stakeholders' understanding of a domain of discourse. Thereby, conceptual models are more than mere descriptions of a particular object of interest or phenomenon under investigation. As a matter of fact, they interpret and reconstruct a domain of discourse allowing both individual understanding and knowledge sharing among group members to improve.

Recent advances in semantic web technology (Berners-Lee, Hendler, Lassila, 2001) provide new and more powerful means to support communities in collaborative modeling activities by creating and reusing shared conceptualizations that guide, direct and shape their learning or working processes. However, most of these tools are still at their infancy. In this paper, we present a comprehensive collection of services, comprising an emerging framework, called Semantic Web Knowledge Middleware (SWKM), that allow support for collaborative knowledge evolution both of domain abstractions (i.e., models) and domain conceptualizations (i.e., modeling languages or ontologies) (Guizzardi, Pires and van Sinderen, 2005). In particular, we offer services for ontology change and comparison, version creation and persistent storage in a knowledge repository in a coherent way that will account for multi-user dynamic environments.

In this work, we assume that the community ontologies and their instances are represented using the RDF language, under the semantics of

(Karvounarakis, Magkanaraki, Alexaki, Christophides, Plexousakis, Scholl & Tolle, 2004); we will use the term *RDF Knowledge Base* (RDF KB in short) to denote a populated ontology, which, in practice, is a set of RDF triples. In order to support personal and group knowledge management based on multiple conceptualizations, we should be able to distinguish ontologies and instances according to the actors (individual or group) involved in their creation. To this end, we will use the notion of *namespaces* and *named graphs*, where a namespace is a collection of RDF/S classes and properties (that constitute an ontology, or part of an ontology), whereas a named graph is a collection of RDF triples (and could contain both data and schema triples, i.e., could constitute an RDF KB, or part of an RDF KB). Namespaces and named graphs will be uniquely identified using URI references. Namespaces and named graphs may depend on other namespaces or named graphs, in the sense that they may reuse elements (e.g., classes) or declarations from other namespaces or named graphs; such dependencies may need to be taken into account in certain services, as we will see in later sections.

The rest of this paper is organized as follows: initially, the usefulness of collaborative knowledge creation is justified through examples of collaborative activities and the underlying principles and interactions are presented; this leads to the formulation of a set of needs and requirements that motivate our work. Subsequently, our suite of services (Semantic Web Knowledge Middleware – SWKM) for supporting knowledge evolution on schema and data are presented and the persistent storage of the changes is discussed. Finally, the paper comments on related work and concludes.

MOTIVATION AND EMERGING REQUIREMENTS

Two scenarios derived from learning environments are presented in this section in order to establish and demonstrate the motivation that drives this work. These examples are also used as the driving force in order to obtain the needed high level requirements that are subsequently supported by the services presented in the next section.

(Re-)Constructing Arguments

There are several cases where evolution of knowledge artifacts can facilitate the learning process. As an example, consider the scenario where a group of people (students, researchers, co-workers etc), possibly with different backgrounds and/or from different fields, meet in order to reach a decision on some issue. In order to scaffold this process the group is presented with an argumentation ontology which could be inspired by similar efforts in the literature (e.g., (Gordon & Karacapilidis, 1997), (Toulmin, 1958)). Said ontology could also be used to annotate related resources. For example, a certain claim might be backed up with a link to a respective resource.

In a scientific environment (Benn, Shum & Domingue, 2005), this scenario could involve re-constructing pre-existing scientific arguments based on a set of research papers, or explicating the group members' own arguments; in a professional environment, it could involve the improvement of the design and function-ability of a company's new products and could involve members with different expertise (e.g., market-analysts, information technology experts and others). In either case, every member of the group prepares a set of resources describing his or her current understanding (view) of the given topic. The extraction engine produces an overall conceptual map, which integrates the individual views and provides a basis for the core discussions of the group.

The group of people collaborating in this scenario need to reconstruct their argumentation in a knowledge artifact (RDF KB) using the provided argumentation ontology. The end product will be a single RDF KB representing the arguments of

the entire group. During collaboration, differences in opinions may arise which should be discussed and resolved in a synchronous manner. In order to identify and isolate those differences, the learners may need to store different versions of their argumentation and compare them using appropriate delta functions. Such dispute resolution will cause changes in the original construction of the argument, thus leading to changes in the original argumentation. Notice that, in this particular example, the changes affect only the data portion of the RDF KB rather than the schema, but this is not the case in all examples of collaboration. A slightly more complicated version of this example would arise if we have different groups of people who use different argumentation frameworks, in which case the system may need to support the storage, classification and retrieval of more than one namespaces through the use of some registry.

Evolution and Use of Multiple Ontologies and RDF KBs

In the above example, the learners were restricted to use a particular argumentation ontology in order to construct their arguments; even in the more complicated case of multiple argumentation ontologies, each learner would use a single, pre-defined ontology that would populate in order to represent his argumentation. However, in the general case, learners should be able to create and use different conceptualizations (both ontologies and related instances) to describe the underlying domain. Similarly, they should be able to describe the domain from different viewpoints and under different perspectives. This implies that the learners should not be in any way restricted to a predefined set of ontologies, but should have the ability to develop their own. Moreover, it should be possible to easily switch between changing the schema of an ontology and changing the data classified under the ontology schema.

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

In this context, a learner or group of learners should have the ability to adapt given KBs to the particular needs of the activities they are involved in. As already mentioned, this adaptation includes the evolution of both the ontology schema and the classified instances. Even though ontologies by definition provide shared conceptualizations for a domain of interest (Gruber, 1993), they also provide the means to carry out activities and hence need to be adapted to local practices and task requirements. For example, a learner might decide that a given ontology does not provide the necessary concepts for the task at hand, and hence might want to extend it. While stable and widely accepted ontologies are useful from a technical point of view, locally adapted and adaptable ontologies seem to be more apt for several applications, including applications related to collaborative learning. Furthermore, the local adaptation of the so created RDF KBs also allows the representation of different perspectives on a shared object of activity, which might help to get a better understanding of the phenomenon at hand.

The existence of different perspectives regarding the given object of activity raises also the need to compare those perspectives; such a comparison is the driving force behind the adaptation and negotiation process. In addition, the continuous mutual adaptation and negotiation would generate various intermediate versions of each RDF KB, some of which the user may want to keep; in this case, an adequate storage facility should

be provided, that would support the storage of the relationships between different versions, along with the versions themselves. Finally, the existence of several different RDF KBs (and versions of RDF KBs) raises the need for a registry facility, that would allow easy access and retrieval of RDF KBs through the use of adequate metadata that would describe said RDF KBs.

Summary of Requirements and Proposed Solution

Collaboration implies that different professional experiences, different social and cultural backgrounds, participants' individual interests and goals, as well as inherent business rules and practices (including tacit ones) will be present in the process, and may cause misconceptions and frustrating ambiguities and misunderstandings (De Leenheer & Meersman, 2007). To smoothen the effects of such differences, the shared background of the collaborating group (partners) should be continuously negotiated until common concepts, characteristics and values have been agreed upon. In this respect, ontologies and RDF KBs, being shared conceptualizations of the domain under discussion (Gruber, 1993), are useful in this process, as they provide the means to describe shared resources of semantics (De Leenheer & Meersman, 2007).

Towards this end, learners have to be aware of existing ontologies and resources, to understand their purpose and meaning but also to adapt them to their own local and temporal environment and ideas. Developing conceptualizations in an intuitive way requires overcoming the formalization barrier imposed by current ontology editors and making the creation and modification of ontologies and resources easy to both use and understand. Moreover, a controlled evolution of ontologies and resources calls for adequate protocols for negotiating the requested changes (Vrandečić, Pinto, Sure and Tempich, 2005). Controlled evolution can be supported by specifying an allowed set of changes and restricting learners to apply only these changes. Additionally, in order to trace the rational of a - controlled or not - evolution, means for comparing successive versions of ontologies and resources should be available and thus the evolution history of the ontologies and the resources should be preserved. The history of the evolution can also be used to in order to apply additional reasoning on the various versions of an ontology regardless of the fact that these versions might or not be materialized at the secondary storage level (Plessers, De Troyer, 2005).

The need to support such collaborative activities implies that RDF KBs have to be viewable, accessible and updatable by learners. View and access is necessary in order for a learner to grasp the understanding of other learners regarding the domain at hand, whereas updatability is necessary in order for the learner to be able to provide their own arguments and feedback regarding a domain of discourse.

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

The updatability requirement, which is present in both examples, is mainly supported through the provision of a service that would effectively support changes in the ontologies and the related instances hosted by a KB. Such changes should be supported automatically and transparently by the system, so that the learner does not have to deal with the technicalities and side-effects of any single change upon his KB; it should be enough for him to indicate the required changes

in a declarative way and let the service do the rest. As always the service should guarantee the consistency of the KB after the application of the computed changes.

As conceptualizations change over time, different versions of an RDF KB may need to be stored and made persistent, so a service should be in place that would keep track of such versions and their relationships. In several cases, it would make sense for a learner to compare the old version of a KB with the new one in order to see the newly submitted changes, or to compare KBs submitted by different learners; such comparisons would be extremely helpful in order for the learners to understand the differences in their argumentation or understanding of the domain, so as to bridge such differences and reach a valid RDF KB representing the opinions of the entire group as closely as possible.

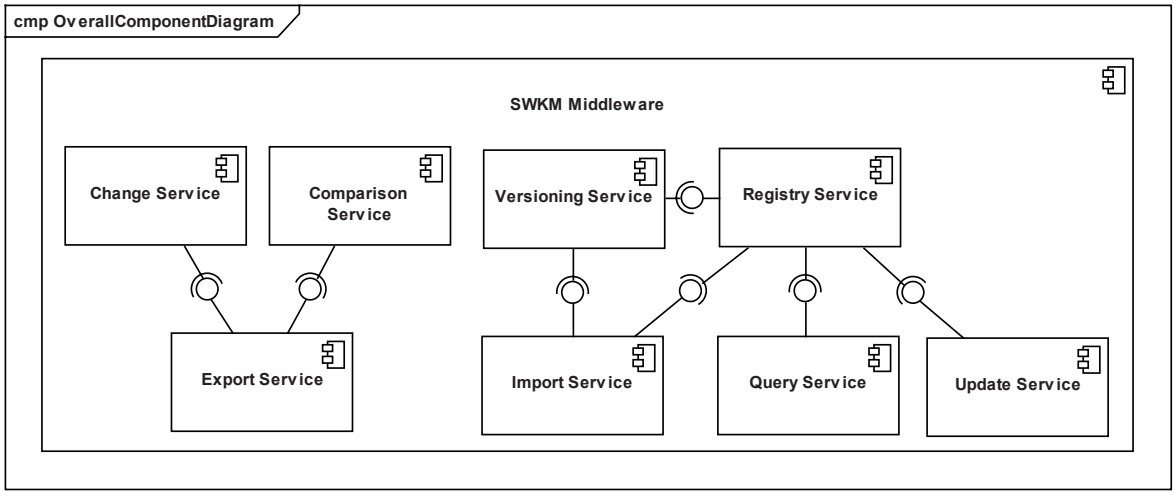
In the following section, we will describe a suite of services that supports the above needs. This suite consists of four services, namely: the *Change Impact service*, which is used to support changes and report the side-effects of a user-requested change; the *Comparison service*, which is used to compare two RDF KBs and identify their differences; the *Versioning service*, which is used to make persistent different versions of an RDF KB and to keep track of different versions; and the *Registry service* which stores and manipulates metadata information regarding ontologies and instantiations for easy access and retrieval of RDF KBs. These services make use at certain points of other basic repository services as ones that store ontologies and data in RDF KBs (Import), retrieve them as a whole (Export) or use a more fine grained approach by letting the user ask (Query) the repository or change it (Update) in a consistent way. Figure 1 illustrates each service as a component and shows the coupling of these services.

EVOLUTION SERVICES

Change Service

The *Change Service* is responsible for determining the actual changes that should occur on an ontology or the related instances (i.e., on an RDF

Figure 1. Component diagram of the SWKM



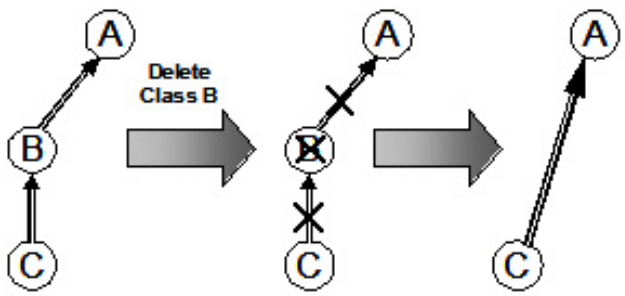
KB) in response to a change request. The actual changes are not always the same as the requested ones, as the original change request could lead to invalidities if performed straightforwardly. For this reason, the change service first attempts to apply the change request to the target RDF KB in a straightforward way; if this naïve application leads to an RDF KB that is meaningless, invalid or does not obey the RDF formation rules (Konstantinidis, Flouris, Antoniou & Christophides, 2007), then additional updates (called *side-effects*) are added to the original request to guarantee validity.

As an example, consider the removal of a class, as shown in Figure 2, where the removal of class B would render all associations of this class with neighboring classes invalid. In such cases, the change service needs to determine additional change operations (side-effects) to be executed along with the original change request which would restore the validity of the KB. In our example, the most obvious set of side-effects would be to remove all invalid associations. In addition, the implicit subsumption relation between A and C that exists (implicitly, as a consequence of the other subsumptions) in the original RDF KB, need not be lost, so it is reinstated in the result, this time in an explicit manner; this is another type of side-effect, which guarantees that only information relevant to the update is lost during the change. For example, if a learner believes that all "Trucks" are "Big_Vehicles" and all "Big_Vehicles" are "Vehicles" and he decides to remove the class "Big_Vehicles" (because, e.g., the definition of the class is too fuzzy, or because it doesn't offer much to the conceptualization of the domain), then this should not cause the loss of the (implicit) information that "Trucks" are "Vehicles".

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

Having said that, the invoker of the service is given the option to restrict the considered KB, as well as the changes and their side-effects to happen in the given collection of namespaces or named graphs, plus, of course, those namespaces or named graphs that the members of this collec-

Figure 2. Change service - Removal of a class



tion depend on; it should be clear that this option may not give the best possible results, as certain side-effects may not be computed.

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

The output of the service is of the same form, i.e., a set of change operations (additions and removals), capturing all the effects and side-effects of the original change request upon the target KB (actual changes). In the example of Figure 2, the output would contain the deletion of B (direct effect), the deletion of the two IsAs (side-effect) and the explicit addition of the previously implicit IsA (side-effect). These effects and side-effects are returned to the invoker (typically, an application with a user interface that interacts with the learner), in order to be visualized and either accepted or rejected.

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

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

The above two properties are motivated by related research on *belief revision* (Gärdenfors, 1992), and they correspond to the principles of Validity and Success respectively that are studied in that field. Notice that, in many cases, there may be more than one possible actual changes (i.e., side-effects) that satisfy Success and Validity. In such cases, the service will select the action that has the minimal possible impact (i.e., the "mildest" possible effects and side-effects) upon the original RDF KB, without negating its validity. In other words, the result of the change should be "as close as possible" to the original KB, according to another belief revision principle, namely the principle of Minimal Change (Gärdenfors, 1992). One possible manifestation of this principle can be found in Figure 2, in which case it caused the explicit addition of the subsumption relation between A and C, to avoid unnecessary loss of information.

The impact of a change upon an RDF KB is measured by means of a preference ordering, which allows the service to determine the most plausible out of the different options for side-effects that restore the KB's validity (i.e., the

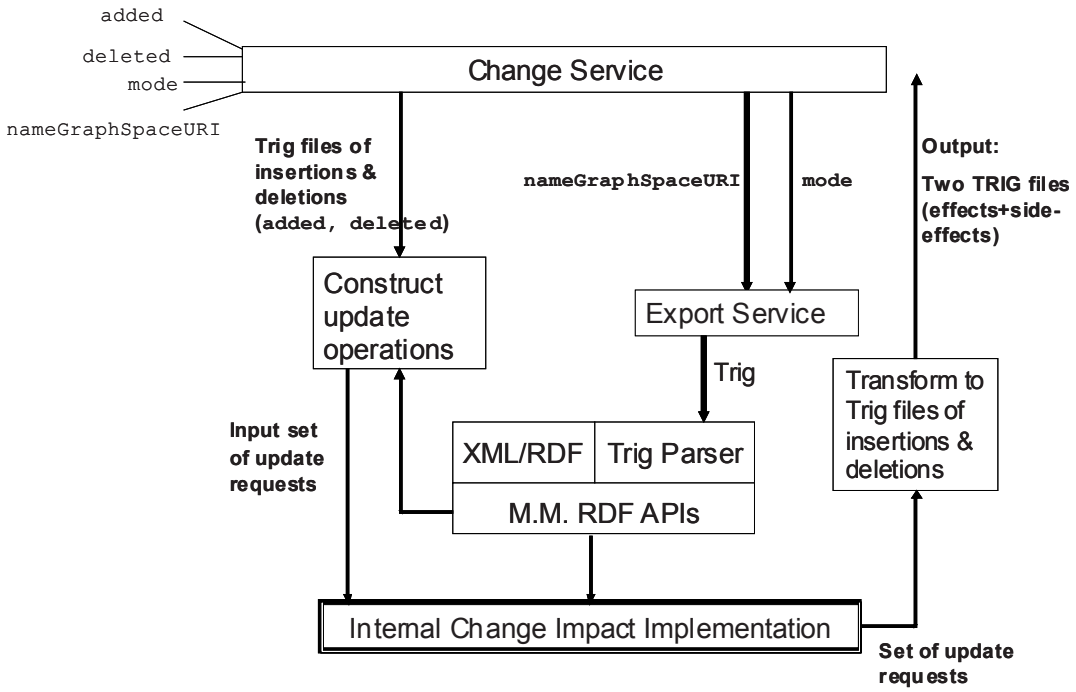
one with the minimal impact) by comparing the impact of different sorts of side-effects upon the RDF KB. Therefore, this preference ordering is a critical parameter that affects the determination of the actual change and implicitly allows us to fine-tune the behavior of the service (i.e., the returned side-effects).

As already mentioned, an update request can contain any number of simple operations (additions or removals of triples), which are not executed in any particular order, but as a whole, in a transactional and deterministic manner. This way, the entire update request is considered while determining the impact of the various potential side-effects. Notice that the set of side-effects computed in this manner may be different from the one we would get if we processed each constituent of the update operation separately.

In order for the system to guarantee the described behavior in a consistent and deterministic manner, the service implementation is backed up by a formal theory which is described in detail in (Konstantinidis, Flouris, Antoniou & Christophides, 2007). Based on this theory we have developed a general-purpose algorithm, which has been proved to exhibit the described behavior for any kind of update request (simple or complex).

This general-purpose algorithm is backed up by a set of special-purpose algorithms which calculate the proper effects and side-effects for simple operations only; this way, we are able to provide faster, special-purpose implementations of our general-purpose algorithm, which are applicable only for simple update requests (thus trading generality for performance). The special-purpose algorithms exhibit the same behavior as the general-purpose one, but are no substitute for it; recall that there is an infinite number of possible update requests, so this effort is inherently incomplete, and we will necessarily have to resort to the general-purpose algorithm for

Figure 3. High-level view of the change service



certain update requests. The process of selecting the proper algorithm (special-purpose or general-purpose) to use for a particular update request is transparent to the user: the service determines whether the given update request is supported by a special-purpose algorithm and adapts the execution sequence accordingly.

Figure 3 shows the general architecture of the web service. As shown in the figure, the change service exposes a single service which is used to apply an update request upon an RDF KB. The signature of the method is as follows:

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

The output of the above method is a pair of strings; the first string represents the RDF triples that should be added to the KB, whereas the second represents the RDF triples that should be removed from the RDF KB. Both strings encode the triples in TRIG (http://sites.wiwiss.fu-berlin.de/suhl/bizer/TriG/) format. As already mentioned, these triples include both the direct effects that were dictated by the original update request, and the ones dictated by validity considerations, i.e., the side-effects. Void additions and removals have been removed from the output.

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

The update request is specified using the string parameters `added` and `deleted`, representing the set of triples that should be added and deleted respectively from the RDF KB (i.e., the original update request). The triples should be encoded using TRIG syntax. The added and deleted triples are combined with the parsed output of the Export Service in order to determine the types of update operations that need to be executed upon the RDF KB and are ultimately fed, along with the RDF KB that was produced by the parsed output of the Export Service, to the Internal Change Impact Implementation to produce the output.

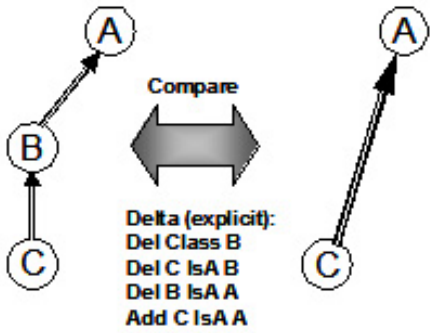
Comparison Service

The *Comparison Service* is responsible for comparing two collections of namespaces or named graphs (KBs). The result of the comparison is a "delta" (or "diff") describing the differences between the two collections of namespaces or named graphs, i.e., the change(s) that should be applied upon the first in order to get to the second (see Figure 4 for an example).

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

Notice that the problem of comparing two namespaces or named graphs is very different

Figure 4. Comparing two namespaces



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

Therefore, even though our comparison will be triple-based, it should be based on semantic, rather than syntactic considerations. Our research has shown that there are alternative methods for computing a semantic delta between namespaces or named graphs (Zeginis, Tzitzikas & Christophides, 2007). In particular, the implicit knowledge (i.e., the inferred triples) contained in the two namespaces or named graphs may or may not be taken into account, leading to the following four cases:

- **Delta Explicit (Δ_e):** Takes into account only explicit triples

  $\Delta_e(K \rightarrow K') = \{Add(t) \mid t \in K' - K\} \cup \{Del(t) \mid t \in K - K'\}$

- **Delta Closure (Δ_c):** Takes also into account inferred triples

  $\Delta_c(K \rightarrow K') = \{Add(t) \mid t \in C(K') - C(K)\} \cup \{Del(t) \mid t \in C(K) - C(K')\}$

- **Delta Dense (Δ_d):** Returns the explicit triples of one KB that do not exist at the closure of the other KB

  $\Delta_d(K \rightarrow K') = \{Add(t) \mid t \in K' - C(K)\} \cup \{Del(t) \mid t \in K - C(K')\}$

- **Delta Dense & Closure (Δ_{dc}):** Resembles Δ_d regarding additions and Δ_c regarding deletions

  $\Delta_{dc}(K \rightarrow K') = \{Add(t) \mid t \in K' - C(K)\} \cup \{Del(t) \mid t \in C(K) - C(K')\}$

In the above bullets, the comparison is performed between two RDF KBs K and K′. The operator C(.) stands for the consequence operator, which is a function producing all the consequences (implications) of K, i.e., all the inferred triples of K. In the example in Figure 4, only the explicit knowledge is taken into account in the comparison, so the shown result corresponds to Δ_e. If the implicit knowledge was also taken into

account, the result would be different (e.g., Δ_c, Δ_d and Δ_{dc}, would not report the addition of the [C IsA A] triple).

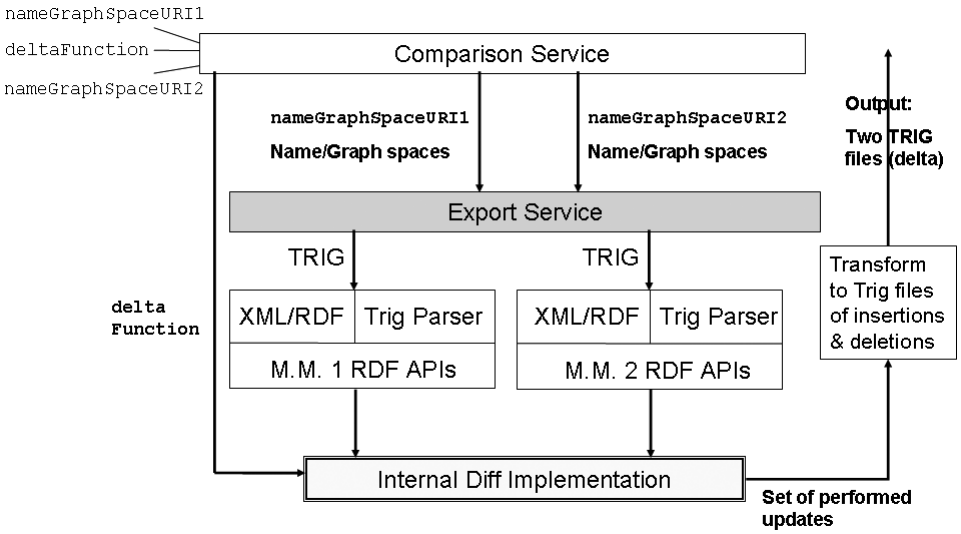
One of the main properties that we intuitively expect to hold in a comparison function is that its output, when applied upon the first RDF KB, should give the second; this property is called *correctness*. In order to study which of the four delta functions guarantees correctness, we should first determine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

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

Using this definition of update semantics, in (Zeginis, Tzitzikas & Christophides, 2007) it was shown that only certain pairs of delta functions with update semantics guarantee correctness, namely: (Δ_e,U_p), (Δ_{dc},U_{ir}) and (Δ_c,U_{ir}). Most existing comparison tools rely on the (Δ_e,U_p) pair. If we consider the update semantics U_{cs}, then the Δ_c function guarantees correctness.

Another critical consideration is related to the size of the delta; in this respect, delta dense (Δ_d) is best, compared to any other delta function (i.e., returns the smallest deltas), whereas Δ_{dc} gives smaller in size delta than Δ_c; on the other hand, Δ_{dc} and Δ_e are incomparable. Notice however that, as we saw above, Δ_d (the smallest possible delta) does not guarantee correctness unless certain conditions hold; for more details, refer to (Zeginis, Tzitzikas & Christophides, 2007).

Figure 5. The Comparison service



In the implementation of the *Comparison Service* we don't adopt any particular policy regarding the "correct" or "best" delta function; in particular, the delta function to be used is just a parameter of the service, and the invoker is assumed to understand the implications of using any particular delta function.

Figure 5 shows the general architecture of the web service of diff. As shown in the figure, the Comparison Service exposes a single service which is used to compare two collections of name or graph spaces and return their delta (diff) according to the selected delta function. The signature of the method is as follows:

```
String[] diff(String[] nameGraph-
SpaceURI1, String[] nameGraph-
SpaceURI2, String deltaFunction)
```

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

The input of the method is the two collections of the namespaces or named graphs to be compared, as well as a parameter indicating the mode of the comparison (delta function). These two collections are passed using the `nameGraphSpaceURI1[]` and `nameGraphSpaceURI2[]` parameters. Each such parameter is an array of strings, each string containing the URI of a namespace or named graph (so each of `nameGraphSpaceURI1[]` and `nameGraphSpaceURI2[]` represents a collection of namespaces and/or named graphs). It should be emphasized that the comparison is not performed upon the namespaces and named graphs in the input only, but also upon the namespaces and named graphs that they depend on. In other words, the compared conceptualizations (RDF KBs) occur by taking the union of the triples in the URIs indicated by `nameGraphSpaceURI1[]` (and `nameGraphSpaceURI1[]`) plus the triples in the namespaces and named graphs that the input namespaces or named graphs depend on. This is implemented through two independent calls to the Export Service (one for each of the compared collections), followed by the parsing of the results to produce the related data structures used by the Internal Diff Implementation.

The `deltaFunction` parameter indicates the type of the delta function to be used in the comparison (Δ_e, Δ_c, Δ_d, Δ_{dc}). This information, along with the parsed output of the Export Service are then fed into the Internal Diff Implementation to produce the output (diff) of the service.

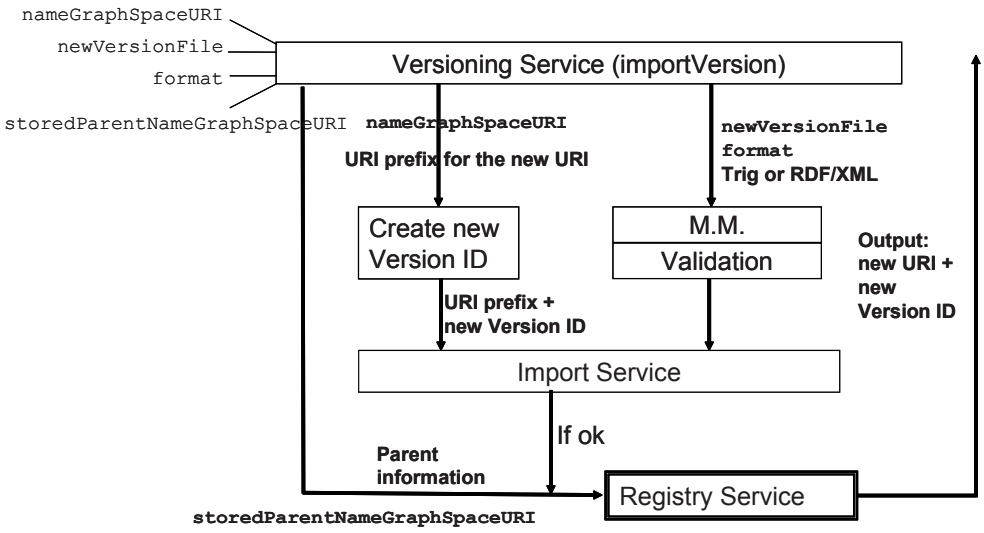
Versioning Service

The *Versioning Service* is responsible for constructing a new persistent version of a namespace or named graph, in effect allowing the creation of several versions of an ontology or its instances, while using the Registry Service (see below) to store the logical relationships between these versions, e.g., which version was created as an evolution of which pre-existing one.

The basic functionality of the Versioning Service offers versioning at the level of single namespaces or named graphs. To this end, it takes as input the information regarding the version's URI, the parent versions' URI(s) and the contents of the new version and creates a persistent version of the namespace or named graph in the given URI, with a new version ID.

By default, a new version ID is generated automatically by the service each time a new

Figure 6. Versioning service (importVersion)



version is requested. The service guarantees that no two versions of the same namespace or named graph will get the same version ID. The user of the service relies on the use of the full URI to refer to the desired version of the namespace or named graph, whereas the Registry Service offers the necessary functionality for accessing the different versions and querying their interrelationships, in a transparent way.

Figure 6 summarizes the functionality of the service. Initially, a new version identifier is created; this identifier will be associated with the new version and will be used to create the URI of the new version. Moreover, the contents of the new version are validated before being fed to the Import Service (along with the new URI), which will make the version persistent, under the new URI. Following the import, appropriate calls to the Registry Service guarantee that the new version is properly recorded in the registry; to this end, the information on the new version's parent(s) is necessary.

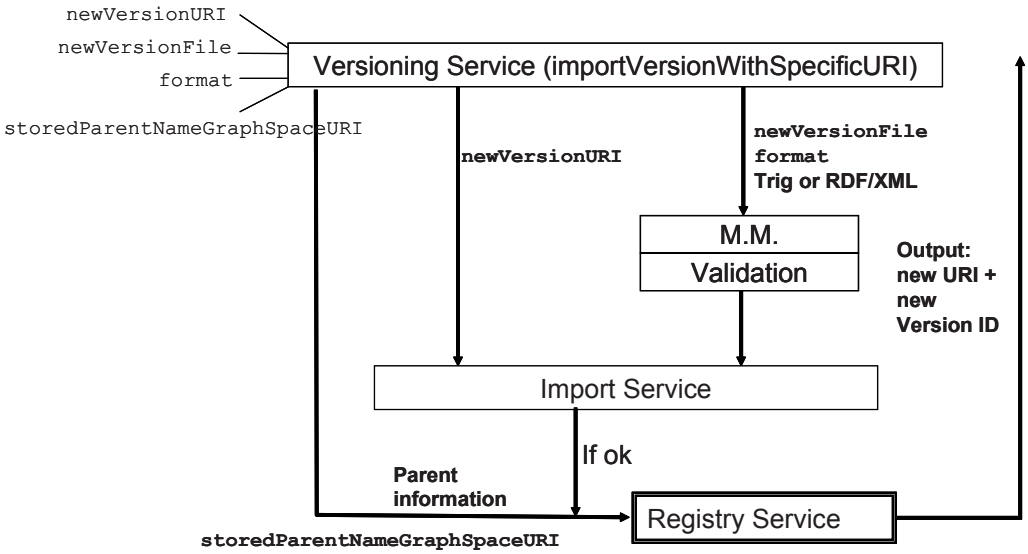
It should be emphasized that the creation of the new version does not remove the old version(s) from the repository. Since the old versions' URIs do not change, references to old versions are still valid. Changes of references to old versions are under the responsibility of the users of the service. The final output of the service is a URI that includes the URI prefix and the version ID of the new version.

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

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

The output of the above method is a string containing the full URI of the new version. This URI could be later used by the invoker in order to get the contents of the new version, through a call to the Export Service. The input consists of `nameGraphSpaceURI` parameter, which is used in the process of determining the new version's URI. The `storedParentNameGraphSpaceURI` parameter is an array of strings, each containing the URI of one of the parents of the current version. If there is no previous version of

Figure 7. Versioning service (importVersion With Specific URI)



the given namespace or named graph (i.e., if the currently created version is the first one), then there are no parents, so the array is empty. The *newVersionFile* parameter contains a string describing all the triples of the new version of the namespace or named graph, which should be stored as the content of the new version. The format of the string in `newVersionFile` could be either TRIG or RDF/XML; the exact format is determined using the `format` parameter.

In addition to the above, we provide two variations of the basic versioning service. The first variation allows the user to bypass the automatic creation of the version ID and indicate the version ID that he wants to use (see also Figure 7). The signature of the method is:

```
void importVersionWithSpecificURI(String newVersionUri, String[] storedParentNameGraphSpaceURI, String newVersionFile, String format)
```

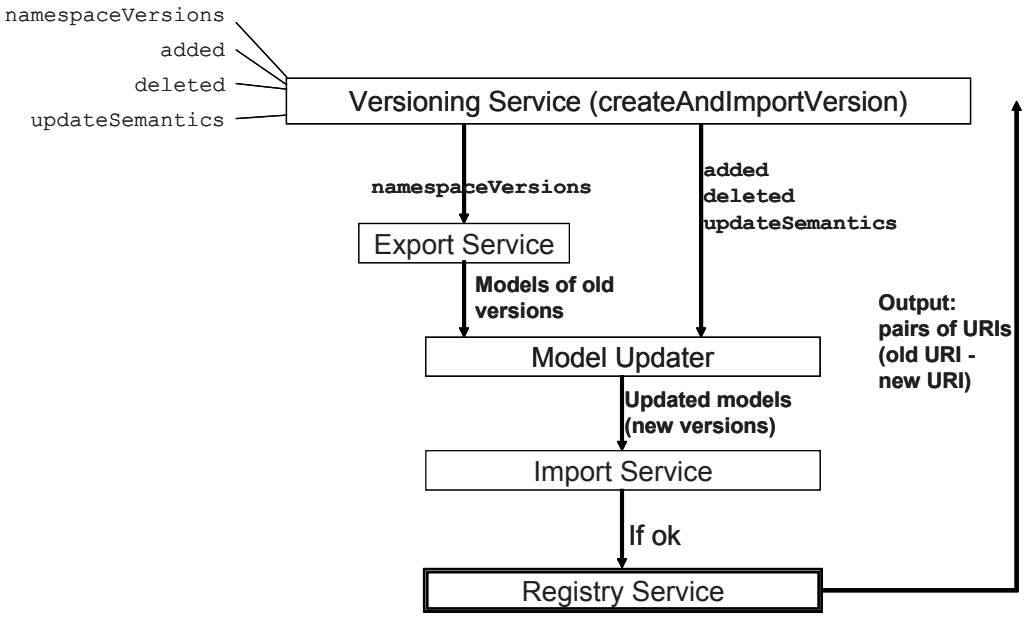
The functionality of this method is similar to the standard importVersion, the only difference being that this method will not automatically create a new version ID, but will use the one provided by the caller (i.e., the one specified by the `newVersionUri` parameter, which replaced the `nameGraphSpaceURI` of `importVersion`). In addition, this method does not provide an output (the new version ID is already known to the caller, so there is no need to return it).

The second variation is used to generate a new version from a given one; the difference with the standard case is that the contents of the new version are not explicitly specified; instead, a "delta" from the old (existing) version to the new one is provided. The signature of this method is the following:

```
String[][] createAndImportVersions(String namespaceVersions[], String added, String deleted, String updateSemantics)
```

Figure 8 depicts the related process. The parameters `added` and `deleted` contain the triples to be added and deleted respectively from the original namespace(s). The parameter `updateSemantics` determines how the additions and

Figure 8. Versioning service (createAndImportVersion)



deletions will be performed. There are two options on the update semantics to be used (see also the Comparison Service): either plain semantics (U_p), or inference and reduction semantics (U_{ir}).

The namespace(s) to be affected are given in the `namespaceVersions` parameter. In order for the method to decide to (from) which namespace(s) each triple should added (deleted), the following convention is used:

- All non-schema triples are ignored (as the method deals with namespaces only).
- Each added schema triple is added to the namespace to which the subject of the triple belongs.
- Each deleted schema triple is deleted from the namespace it belongs.
- If any change (addition or deletion) should affect (per the above convention) a namespace that is not in the given list of namespaces (`namespaceVersions`), then this particular change is ignored.

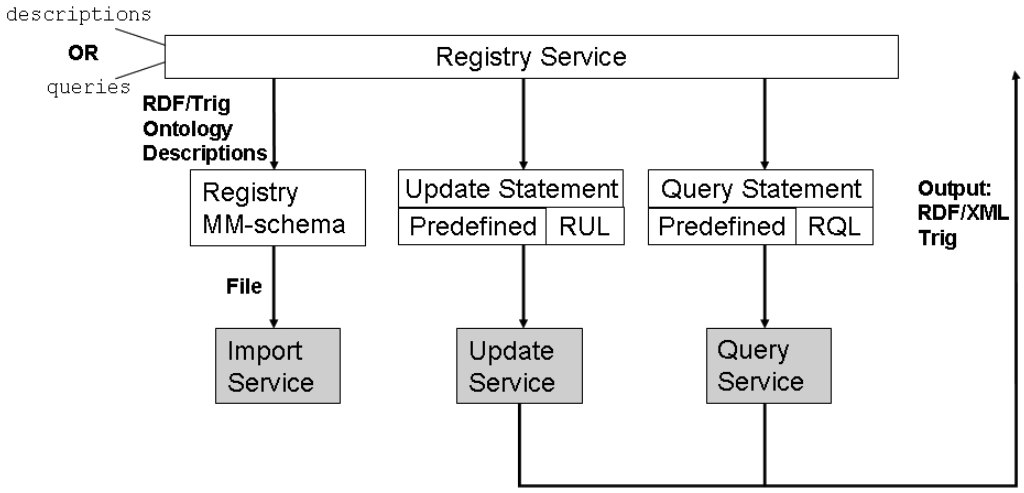
After all additions and deletions have been performed, the modified versions of the input namespaces are saved in the repository. Notice that the namespaces which are unaffected by the above changes are not saved. In the case of `createAndImportVersions`, there is no need to pass explicitly (as a parameter) the parents of the newly created versions, as these are obviously the ones in the `namespaceVersions` parameter.

The output of the method indicates the new URI of each namespace that was given in the input. In particular, the output is an array containing one pair for each input namespace; such a pair consists of the old and the new URI of the input namespace under question. This way, the caller can associate the URI of each old version with the URI of the respective new version.

Registry Service

The role of the *Registry Service* is to record and manage metadata information about ontologies, schemas or namespaces stored in the knowledge

Figure 9. High-level view of the registry service



repository. Furthermore, the registry offers the possibility to keep track of the development lifecycle of a schema through the support of storing versions (in a quite general sense, as we will see below), their metadata and the relationships among them. Both schema and version information follow the Ontology Registry Schema that is stored in the knowledge repository and is appropriately instantiated for each schema and version stored. Applications using the registry have the possibility to update and retrieve information about the already recorded schemas and their versions by using the available service methods. More specifically, the registry provides certain searching facilities, supports editing functions that modify stored information about ontologies or add new ones by using a query/update service based mechanism and supports a versioning mechanism in order to maintain the changes of ontologies in the registry.

The Registry Service is implemented as a web service and the different functionalities offered by it are implemented as web methods, which depend on, and use, the services provided by the knowledge repository, namely the Import, Update and Query Services. These dependencies are depicted in Figure 9.

The Registry Service is using its own ontology, encoded in RDF and following the RDF/S, in order to explicitly describe every other ontology stored in the Knowledge Repository. This ontology is called the *Ontology Registry Schema* and is described below (see Figure 10). For each of the ontologies stored in the Knowledge Repository, an instance of the proper type is created and stored under the Ontology Registry Schema. The Registry is also supporting the recording of the versioning of schemas by allowing each ontology to create multiple instances of the corresponding class `Version` and relate these instances to the proper instance of the class `Schema`. Thus, the metadata stored for each namespace are divided into two main categories regarding to whether their values are changing with each version (e.g., the number of classes or the related namespaces) or they are permanent characteristics of the namespace (e.g., the encoding or the URI prefix). This, in turn, imposes the rule that at least one version should exist in the Knowledge Repository for any stored namespace and its instance should

be correctly related to the instance representing the namespace in the registry.

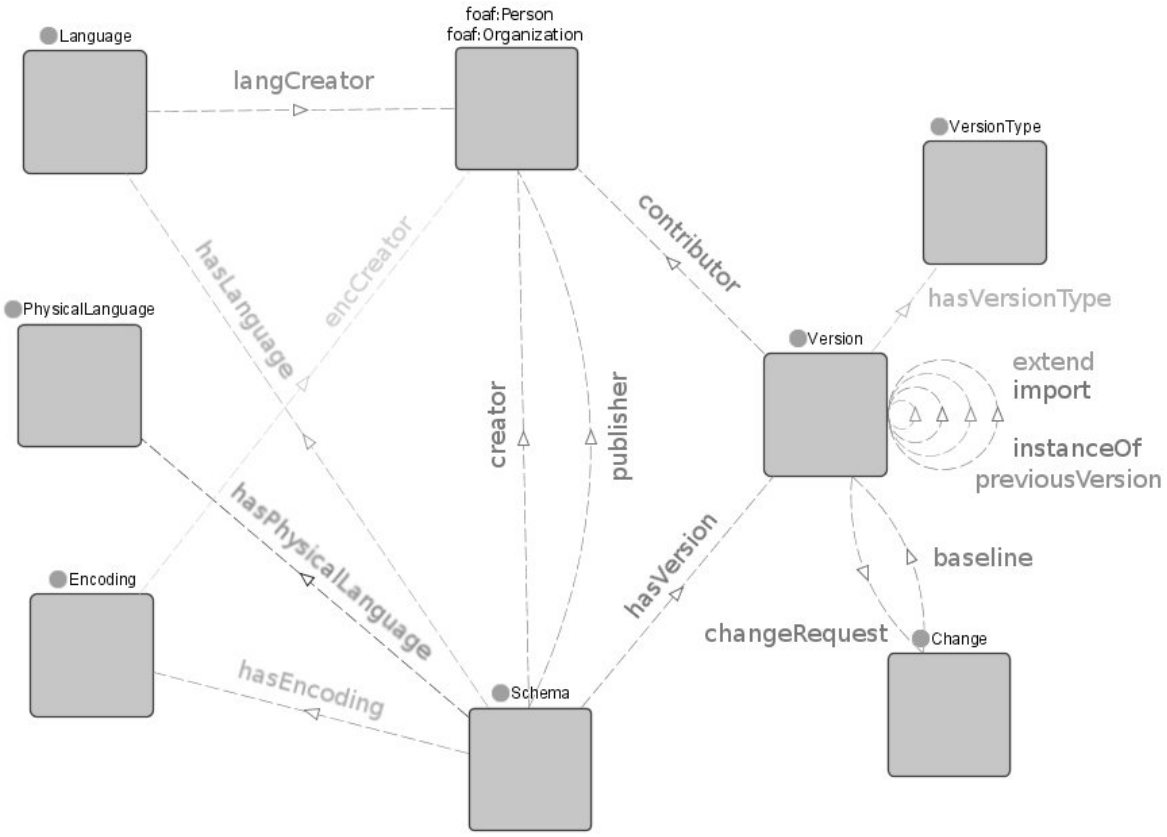
Since keeping track of versions has a significant role in the lifecycle of a schema, the registry supports a sophisticated versioning mechanism, accounting for and supporting the fact that different versions of a schema can be developed in parallel. Thus, during the lifecycle of a schema its versions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

More specifically, the schema of Ontology Registry consists of five basic classes: *Schema, Version, Change, foaf#Person* and *foaf#Organization*.

- The `Schema` class represents a stored namespace (or ontology or schema) and includes, besides the URI of the schema, information about the creator, the title, the purpose, the keywords etc. To allow more fine-grained classification, the `Schema` class is further specialized via a number of subclasses, namely: `Ontology`, `Thesaurus`, `Taxonomy`, `SemanticNetwork`, `DomainOntology`, `UpperOntology`, `TaskOntology`, `CoreOntology`, `ApplicationOntology`, `FederatedThesaurus`, `FacetedThesaurus` and `NetworkedThesaurus`.
- The `Version` class is used to allow the storage of different versions of the same schema. It is correlated to class `Schema` by the property `hasVersion` and describes attributes of a schema that might change between versions such as statistical characteristics of a schema (number of classes, number of properties, maximum length of a hierarchy). As one might see, this class also contains properties that correlate one schema to another with the relationships `import`, `extend` and `instanceOf`. Moreover, class `Version` has a property with predefined values that is used to indicate the intended uses of a version regarding its evolution during the version lifecycle. The predefined values are instances of `VersionType` class. The evolution can be seen in two ways: versions that are going to be developed in parallel and versions that are developed sequentially and depend on one another. Thus, the `VersionType` class can take the form of one of the following subclasses: `Permanent` (not to be merged in the future), `Temporal` (might be merged in the future) and `Revision` (replacing its previous versions).
- The `Change` class is correlated with class `Version` through the property `changeRequest` and describes the insertions/deletions of RDF statements that have led to the creation of this version (in the form of add/delete statements like the ones produced by the Comparison Service).
- The `(FOAF#)Person` and `(FOAF#)Organization` classes from the schema `FOAF` are correlated to both classes `Schema` and `Version` through the properties `creator`, `publisher` and `contributor` respectively.

In addition to the above classes, some additional ones have been specified as well; these are related to the language, encoding and physical

Figure 10. Ontology registry schema



language used in the document describing a specific namespace. The main classes and properties of Ontology Registry Schema are illustrated in Figure 10.

The Registry Service offers functionalities for:

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

More specifically, the Registry Service offers the possibility to retrieve and update ontology metadata information from the repository. In order to retrieve data from the registry, one can either type an RQL query, or use a query from a set of predefined ones. The latter type (the predefined queries) are exposed through a set of web service

methods and are highly configurable by the developer of the service allowing for the necessary flexibility and taking advantage of the knowledge of the Ontology Registry Schema. Similarly, in order to update the information stored in the registry a set of implemented web methods is exposed, accounting for most actions that might be needed by the user and assuring the necessary consistency of the information in the registry, imposing for example the rule of necessitating at least one version per schema; nevertheless, the user can always post updates in RUL, in which case (s)he bares also the responsibility for keeping the consistency rules.

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

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

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

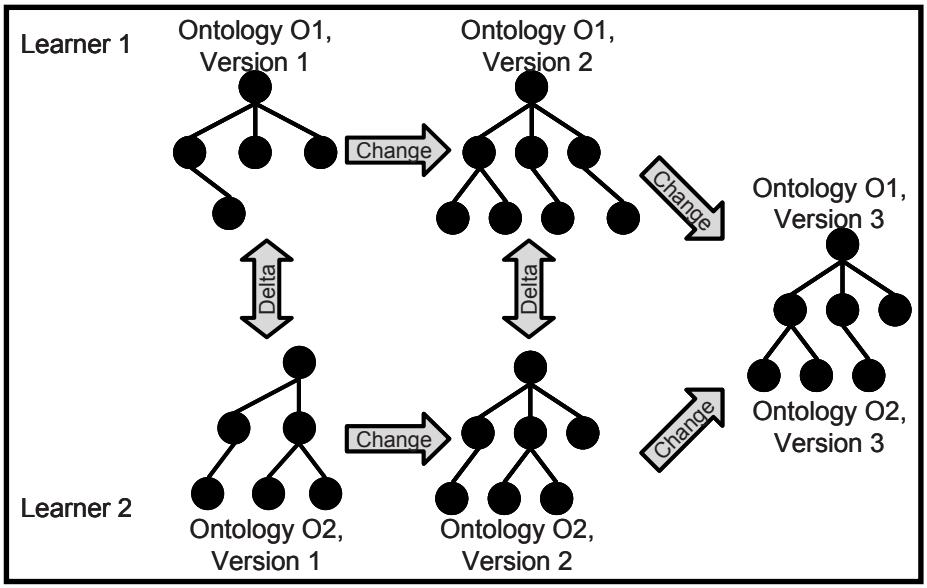
The corresponding ones for updating information already stored in the Registry (including deletion of instances from the registry, update of the range properties with the constraint that the properties have literals as a range, etc) are:

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

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

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

Figure 11. An Example of learners' collaboration



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

USAGE EXAMPLE

Consider two learners participating in a collaborative knowledge creation session (illustrated in Figure 11). Initially, each learner has to individually and separately understand the same domain of interest and develop a personal conceptualization (ontology), either from scratch or by refining an existing one, thus reflecting his own viewpoint on the domain at hand. Following that, the learners should know their modeling differences in order for each of them to understand the other; this can be achieved through the *Comparison Service*.

The role of this service is to compare two ontologies and report their differences in the form of a "delta"; this "delta" can equivalently be seen as a set of changes that should be applied upon the first ontology in order to get the second and vice versa. Notice that this problem is not trivial, as the comparison should be based on semantic, rather than syntactic considerations. The main complicating issue is related to the fact that an ontology carries within implicit knowledge. For example, if it is declared that the concept "Bottlenose Dolphin" is a sub-concept of "Dolphin", which, in turn, is a sub-concept of "Mammal", then it is implied that "Bottlenose Dolphin" is a sub-concept of "Mammal". The latter is inferred (implicit) knowledge, which does not appear anywhere in the ontology. This issue leads to four different comparison (delta) functions, depending on whether (and to what extent) inferred knowledge is taken into account, which were described in the previous section.

The determination of the differences in the learners' conceptualizations (through the Comparison Service) should allow the learners to firstly clarify and then bridge their disagreements, through discussion and negotiation, as well as to adapt their ontologies by applying changes that would cause the two ontologies to converge towards a more consensus state. This adaptation

is done using the *Change Service*, which is a critical service since it allows learners to review and evolve their conceptualizations.

The need for such a service stems from the fact that the original change request could lead to invalidities if performed straightforwardly (e.g., if we remove a concept, we cannot have any property or subsumption relationship that points to the removed concept); thus, the actual changes performed are not always just the same as the requested ones and the main role of the Change Service is to determine what the actual changes (in response to a given update request) should be. The general idea is to initially apply the update request upon the ontology in a straightforward, but controlled, way and then to correct any incurred invalidities through the use of carefully selected additional changes (*side-effects*) which are executed along with the original request. For example, if a concept is removed, the side-effects could include the removal of all subsumption relationships that point to the removed concept, handling the data classified under this concept, etc. An update operation defined by applying the effects and side-effects (i.e., the output) of this process upon the original ontology provably complies with the principles of Success, Validity and Minimal Change, as detailed in the previous section. One important feature of the service is that the actual changes determined by it are not directly applied upon the ontology, but are returned to the learner (or a collaborating group of learners) so that these changes can be visualized and accepted or rejected according to a certain negotiation policy.

The process of calculating the differences between two ontologies and then applying changes in order to converge to a more compatible conceptualization is an iterative process that could require several steps before learners agree on one or more possible alternative ontologies. It should be noted that although consensus might be the ultimate goal, it is not necessarily reached in all cases.

As learners update their ontology during a collaborative knowledge creation session, it often makes sense to keep particular snapshots (versions) of the ontology, for many reasons like backup, rollback, history of interaction and collaboration, keeping track of the learners' progress, marking particular milestones of the process, etc. This is fulfilled by the *Versioning Service* which has to construct persistent versions of changed ontologies upon request, while using the *Registry Service* to keep track of the logical relationships between such versions, i.e., which version was created as an evolution of which pre-existing one etc. The Registry Service allows the learners to keep track of the development lifecycle of an ontology through the support of storing, retrieving and updating versions, their metadata and the relationships among them.

RELATED WORK

Significant work in the area of evolution has been carried out the last few years. It has taken various forms from Belief Revision (Gärdenfors, 1992) and its application so as to support ontological changes (Flouris, 2007), to Ontology Maturing (Braun, Schmidt, Walter, Zacharias, 2007); and from Emergent Semantics operators (Tzitzikas et al, 2007) to simple (or more complicated) Ontology Editors, like Protégé (Noy, Fergerson, & Musen, 2000). All these works share the fact that they are dealing with ontology evolution and less with data evolution which can be considered a byproduct of the evolution process; thus, ontology evolution can be defined as the incorporation of new knowledge in an ontology or more accurately, as the process of modifying an ontology in response to a certain change in the domain or its conceptualization. Ontology change has also been studied in terms of managing the ontology lifecycle; the work of (Novacek et al., 2007) gives examples of such a lifecycle and studies how it can be supported, by allowing for versioning and subsequent merging

and alignment of the produced versions back to a single ontology. A survey on the various different types of ontology change can be found in (Flouris, Manakanatas, Kondylakis, Plexousakis and Antoniou, 2008).

The work presented here is based on algorithmic implementations of the evolution and comparison. Thus, a formal proof of the validity of the results can be presented (Konstantinidis, Flouris, Antoniou & Christophides, 2007), (Zeginis, Tzitzikas & Christophides, 2007) whilst the flexibility of the system is guaranteed, which means that there is no need to predefine (evolution) strategies, as, e.g., in KAON (Gabel T., Sure Y. & Voelker J, 2004), or to get only one solution per invalidity, as, e.g., in Protégé (Noy, Fergerson, & Musen, 2000). A detailed analysis of how these works compare with ours can be found at (Konstantinidis, Flouris, Antoniou and Christophides, 2007).

Apart from the theoretical work on ontology change, systems that support such processes have recently emerged. For instance, the well known ontology editor Protégé provides a semi-automatic tool for ontology merging and alignment called PROMPT (Noy, & Musen, 2003); in PROMPT, some of the change tasks are performed automatically while others need user intervention. In addition, it allows the detection of inconsistencies and the suggestion of solutions. As mentioned though, Protégé is suggesting only one solution to resolve each inconsistency. Additionally, Protégé offers the Changemanagement plugin (Noy, Chugh, Liu & Musen 2006) for storing changes along with annotations on them. At the same level is SWOOP (Kalyanpur, Parsia, Sirin, Cuenca-Grau, Hendler, 2005), which allows for collaborative editing and annotation of ontologies. Both tools are actually ontology editors, so they differ greatly from SWKM which is a framework built as a collection of web services. DILIGENT (Kalyanpur, Parsia, Sirin, Cuenca-Grau, Hendler, 2005) and ONKI (Tempich, Pinto, Sure, Staab, 2005) is more of an ontology engineering methodology, rather than a full framework, and it is lacking support for changes on the dependent ontologies; thus, it cannot be straightforwardly used to support knowledge processes and cannot be directly compared to our work.

Closer to SWKM are frameworks like KAON and Hozo (Kozaki, Sunagawa, Kitamura and Mizoguchi 2007). Hozo is more of a standalone application at the moment, and provides many of the ontology evolution features required. On the other hand, KAON, similarly to SWKM, provides a collection of services. But in the evolution arena KAON is using predefined strategies to decide on the evolution that will be chosen (i.e., which will the associated side effects be); this approach, at some cases, exhibits invalid or non-uniform behavior (Konstantinidis, Flouris, Antoniou and Christophides, 2007). Finally OilED (Bechhofer, Horrocks, Goble, and Stevens, 2001) and OntoStudio (Sure, Erdmann, Angele,Staab, Studer, and Wenke, 2002) provide limited functionality; for example, OilED is completely missing an invalidity resolution mechanism. Thus, SWKM is, to the best of our knowledge, the only framework that provides a complete support for semantic evolution.

Additionally to the generic Knowledge Management platforms, some e-learning frameworks make use of such evolution capabilities in order to support learning in emerging environments. There have already for some time existed guidelines for service based e-learning platforms. IMS Abstract Framework (Guangzuo, 2004) and ELF (http://www.elframework.org/) are two examples of such guidelines; they both provide a loose description of the basic components that are necessary to build an evolving e-learning system. Moreover, OKI (Open Knowledge Initiative - http://www.okiproject.org/) separates the development of e-learning platforms based on it into layers of services. SWKM services can be used with these

initiatives in order to provide support for the necessary evolution of ontologies and data in an emerging learning environment.

CONCLUSION

This paper presented SWKM, a suite of advanced services for managing knowledge artifacts. All artifacts (either ontologies or resources) are stored in a common knowledge repository, while their evolution is supported through services allowing the retrieval, update, comparison and versioning of such artifacts. It is important to note that the validity of the knowledge repository is verified (and guaranteed) after every executed operation against one or more knowledge artifacts. One distinctive characteristic of the evolution and comparison algorithms is that they allow for the adoption of different strategies on the fly, depending on how the implicit knowledge that the ontologies carry within is exploited.

SWKM offers the aforementioned capabilities in terms of (web) services, which can be seamlessly employed by anyone interested to support evolution, comparison and versioning of knowledge artifacts described using RDF. This allows for widespread usage, since there are no development barriers attributed to programming languages or operating environments that someone might need to overcome.

Although the last few years there are various tools available that support the Semantic Web paradigm, to the best of our knowledge, none offers the breadth of choices of SWKM and the coherence of the functionalities described here.

ACKNOWLEDGMENT

This work was partially supported by the EU Integrated Project KP-Lab (FP6-2004-IST-4). The KP-Lab Integrated Project is sponsored under the 6th EU Framework Programme for Research and Development. The authors are solely responsible for the content of this article. It does not represent the opinion of the KP-Lab consortium or the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

REFERENCES

Bechhofer, S., Horrocks, I., Goble, C., & Stevens, R. (2001). OilEd: A reason-able ontology editor for the Semantic Web. *Proceedings of the Joint German/Austrian Conference: Advances in Artificial Intelligence* (KI-01).

Benn, N., Shum, B.S., & Domingue, J. (2005). Integrating scholarly argumentation, texts and community: Towards an ontology and services. *Tech Report KMI-05-5*, http://kmi.open.ac.uk/publications/pdf/kmi-05-5.pdf

Berners-Lee T., Hendler J., Lassila O. (2001, May): The Semantic Web: A new form of Web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, *17.*

Braun S., Schmidt A., Walter A., Zacharias V. (2007). The ontology maturing approach for collaborative and work integrated ontology development: Evaluation results and future directions, *Proceedings of the International Workshop on Emergent Semantics and Ontology Evolution*, 12 Nov. 2007, Bexco, Busan Korea.

De Leenheer, P., & Meersman, R. (2007). Towards community-based evolution of knowledge-intensive systems. In *Proceedings of Ontologies, Databases, and Applications of Semantics*.

Ding, Y., & Fensel, D. (2001). Ontology library systems: The key to successful ontology re-use. In *Proceedings of the 1st International Semantic Web Working Symposium (SWWS'01).*

Domingue, J., Motta, E., Shum, S.B., Vargas-Vera, M., Kalfoglou, Y., & Farnes, N. (2001). Supporting ontology driven document enrichment within communities of practice. *In Proceedings of the 1st International Conference on Knowledge Capture (K-CAP-01)*, (pp. 30-37), New York, USA: ACM Press.

Flouris, G. (2007) On the evolution of ontological signatures. *Proceedings of the Workshop on Ontology Evolution (OnE-07.*.

Flouris G., Manakanatas D., Kondylakis H., Plexousakis D., & Antoniou G. (2008). Ontology change: Classification and survey. *Knowledge Engineering Review (KER)*, to appear.

Gabel T., Sure Y., & Voelker J, (2004) KAON-Ontology Management Infrastructure. *SEKT informal deliverable, 3(1)*

Gärdenfors, P. (1992). Belief revision: An introduction. In Gärdenfors, P. (ed). *Belief Revision*, (pp. 1-20), Cambridge University Press.

Gordon, T.F., & Karacapilidis, N. (1997). The zeno argumentation framework. *In Proceedings of the 6th International Conference on Artificial Intelligence and Law,* ACM Press, New York.

Gruber, T.R. (1993). A translation approach to portable ontology specifications. *Available at: http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html*

Guangzuo C. (2004). OntoEdu: Ontology-based education grid system for E-Learning. *GCCCE2004*, HongKong.

Guizzardi, G., Ferreira Pires, L., & van Sinderen, M. (2005). Ontology-based evaluation and design of domain-specific visual modeling languages. *Proceedings of the 14th International Conference on Information Systems Development*, Karlstad, Sweden.

Kalyanpur, A., Parsia, B., Sirin, B., Cuenca-Grau, B., & Hendler, J.(2005) Swoop: A 'Web' ontology editing browser, *Journal of Web Semantics, 4*(2), 144-153

Karvounarakis G., Magkanaraki A., Alexaki S., Christophides V., Plexousakis D., Scholl M., & Tolle K. (2004). RQL: A functional query language for RDF. *The Functional Approach to Data Management: Modelling, Analyzing and Integrating Heterogeneous Data*, (pp. 435-465). P.M.D.Gray, L.Kerschberg, P.J.H.King, A.Poulovassilis (eds.), LNCS Series, Springer-Verlag.

Konstantinidis, G., Flouris, G., Antoniou, G., & Christophides, V. (2007). Ontology evolution: A framework and its application to RDF. In *Proceedings of the Joint ODBIS & SWDB Workshop on Semantic Web, Ontologies, Databases (SWDB-ODBIS-07).*

Kozaki K., Sunagawa E., Kitamura Y., & Mizoguchi R. (2007). A framework for cooperative ontology construction-based on dependency management of modules. *Proceedings of the International Workshop on Emergent Semantics and Ontology Evolution*, 12 Nov. 2007, Bexco, Busan Korea.

Novacek V., Handschuh S., Maynard D., Laera L., Kruk S. R., Volkel M., Groza T., & Tamma V. (2007). D2.3.8v1 Report and prototype of dynamics in the ontology lifecycle, *Deliverable 2.3.8 of the Knowledge Web FP6-507482 Project available at: http://knowledgeweb.semanticweb.org/semanticportal/sewView/frames.jsp*

Noy, N., Fergerson, R., & Musen, M. (2000). The knowledge model of protégé-2000: Combining interoperability and flexibility. *Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management: Methods, Models, and Tools (EKAW-00)*, 17–32

Noy, N.F., & Musen, M.A. (2003.) The PROMPT suite: Interactive tools for ontology merging and mapping. *International Journal of Human-Computer Studies*, *59*(6), 983-1024

Noy N., Chugh A., Liu W. and Musen M. (2006) A Framework for Ontology Evolution in Collaborative Environments. *In: 5th International Semantic Web Conference*, Athens, GA, USA

Plessers P., & De Troyer O. (2005). Ontology change detection using a version log, *Proceedings of the 4th International Semantic Web Conference (ISWC-05)* Galway, Ireland 6-10 November 2005.

Sure Y., Erdmann M., Angele J., Staab S., Studer R., & Wenke, D. (2002). OntoEdit: Collaborative ontology development for the Semantic Web. *Proceedings of the 1st International Semantic Web Conference (ISWC-02).*

Tempich, C., Pinto, H.S., Sure, Y., & Staab, S. (2005) An argumentation ontology for dIstributed, loosely-controlled and evolvinG engineering processes of ontologies (DILIGENT). In: *The 2nd European Semantic Web Conference*, Greece, 241-256

Toulmin, S. (1958). *The uses of argument.* Cambridge: Cambridge University Press,

Valo, A., Hyvonen, E., & Komurainen, V. (2005). A tool for collaborative ontology development for the Semantic Web, in: *Proc. of International Conference on Dublin Core and Metadata Applications 2005*, Madrid, Spain

Tzitzikas Y., Christophides V., Flouris G., Kotzinos D., Markkanen H., Plexousakis D., & Spyratos N. (2007), Emergent knowledge artifacts for supporting trialogical E-Learning, *International Journal of Web-Based Learning and Teaching Technologies* (accepted).

Vrandečić, D., Pinto, S., Sure Y., & Tempich, C. (2005). The DILIGENT Knowledge Process. *Journal of Knowledge Management*, *9*, 85-96.

Zeginis, D., Tzitzikas, Y., & Christophides, V. (2007). On the foundations of computing deltas between RDF models. In *Proceedings of the 6th International Semantic Web Conference (ISWC-07).*