On RDF/S Ontology Evolution

George Konstantinidis[1,2], Giorgos Flouris[1],
Grigoris Antoniou[1,2], and Vassilis Christophides[1,2]

[1] Institute of Computer Science, FORTH, GREECE
[2] Computer Science Department, University of Crete, GREECE
{gconstan, fgeo, antoniou, christop}@ics.forth.gr

Abstract. The algorithms dealing with the incorporation of new knowledge in an ontology (ontology evolution) often share a rather standard process of dealing with changes. This process consists of the specification of the language, the determination of the allowed update operations, the identification of the invalidities that could be caused by each such operation, the determination of the various alternatives to deal with each such invalidity, and, finally, some selection mechanism for singling out the "best" of these alternatives. Unfortunately, most ontology evolution algorithms implement these steps using a case-based, ad-hoc methodology, which is cumbersome and error-prone. The first goal of this paper is to present, justify and make explicit the five steps of the process. The second goal is to propose a general framework for ontology change management that captures the of this process, in effect generalizing the methodology employed by existing tools. The introduction of this framework allows us to devise a whole class of ontology evolution algorithms, which, due to their formal underpinnings, avoid many of the problems exhibited by ad-hoc frameworks. We exploit this framework by implementing a specific ontology evolution algorithm for RDF ontologies as part of the FORTH-ICS Semantic Web Knowledge Middleware (SWKM).

1 Introduction

Change management is a key component of any knowledge-intensive application. The same is true for the Semantic Web, where knowledge is usually expressed in terms of ontologies and refined through various methodologies using *ontology evolution* techniques. The most critical part of an ontology evolution algorithm is the determination of *what* can be changed and *how* each change should be implemented. The main argument of this paper is that this determination can be split into the following 5 steps, which, although not explicitly stated, are shared by many ontology evolution tools:

1. *Model Selection.* The allowed changes, as well as the various alternatives for implementing each change, are constrained by the expressive power of the ontology representation model. Thus, the selection of the model may have profound effects on what can be changed, and how, so it constitutes an important parameter of the evolution algorithm.

2. *Supported Operations.* In step 2, the supported change operations upon the ontology are specified.
3. *Validity Model.* Problems related to the validity of the resulting ontology may arise whenever a change operation is executed; such problems depend on the validity model assumed for ontologies.
4. *Invalidity Resolution.* This step determines, for each supported operation and possible invalidity problem, the different (alternative) actions that can be performed to restore the validity of the ontology.
5. *Action Selection.* During this step, a selection process is used to determine the most preferable among the various potential actions (that were identified in the previous step) for execution.

Unfortunately, most of the existing frameworks (e.g., [1, 5, 8, 15]) address such ontology evolution issues in an ad-hoc way. As we will see in Section 3, this approach causes a number of problems (e.g., reduced flexibility, limited evolution primitives, non-faithful behavior etc), so evolution algorithms could benefit a lot from the formalization of the aforementioned change management process. In Section 2, we define ontology evolution and give a general overview of the state of the art in the field; this allows us to motivate our work and place it in its correct context. In Section 3, we describe four typical ontology evolution systems, namely OilEd [1], KAON [5], Protégé [8] and OntoStudio (formerly OntoEdit [15]); we show how these systems fit on the aforementioned five-step process and criticize the ad-hoc methodology that they employ to face these steps.

Section 4 introduces the general formal framework that we employ in order to model the various steps of this process. Our framework allows us to deal with arbitrary change operations (rather than a predetermined set). In addition, it considers all the invalidity problems that could, potentially, be caused by each change, and all the possible ways to deal with them. Finally, it provides a parameterizable method to select the “best” out of the various alternative options to deal with an invalidity, according to some metric. The formal nature of the process allows us to avoid resorting to the tedious and error-prone manual case-based reasoning that is necessary in other frameworks for determining invalidities and solutions to them, and provides a uniform way to select the “best” option out of the list of available ones, using some total ordering. Our framework can be used for several different declarative ontological languages and semantics; however, for implementation and visualization purposes, we instantiate it for the case of RDF, under the semantics described in [11].

Finally, in Section 5, we exhibit the merits of our framework via the development of a general-purpose algorithm for ontology evolution. This algorithm has general applicability, but we demonstrate how it can be employed for the RDF case. Then, we specialize our approach for the case of RDF and devise a number of special-purpose algorithms for coping with RDF changes (similar to the existing ad-hoc ontology evolution algorithms), which sacrifice generality for efficiency; the main advantage of such special-purpose algorithms with respect to the standard ad-hoc methodologies is that, due to their formal underpinnings

and their proven compatibility with the general framework, they enjoy the same interesting properties.

The above algorithms are currently being implemented as part of the FORTH-ICS Semantic Web Knowledge Middleware (SWKM), which provides generic services for acquiring, refining, developing, accessing and distributing community knowledge. The SWKM is composed of four services, namely the Comparison Service (which compares two RDF graphs, reporting their differences), the Versioning Service (which handles and stores different versions of RDF graphs), the Registry Service (which is used to manipulate metadata information related to RDF graphs) and the Change Impact Service (which deals with the evolution of RDF graphs). The SWKM is backed up by a number of more basic services (Knowledge Repository Services) which allow basic storage and access functionalities for RDF graphs[1]. This paper describes the algorithms we employ for the Change Impact Service of SWKM, as well as the underlying theoretical background of the service.

2 Related Work and Motivation

2.1 Short Literature Review

Ontology evolution deals with the incorporation of new knowledge in an ontology; more accurately, the term refers to *the process of modifying an ontology in response to a certain change in the domain or its conceptualization* [4]. Ontology evolution is an important problem, as the effectiveness of an ontology-based application heavily depends on the quality of the conceptualization of the domain by the underlying ontology, which is directly affected by the ability of an evolution algorithm to properly adapt the ontology both to changes in the domain (as ontologies often model dynamic environments) and to changes in the domain's conceptualization (as no conceptualization can ever be perfect) [4].

In order to tame the complexity of the problem, six phases of ontology evolution have been identified in [12], occurring in a cyclic loop. Initially, we have the *change capturing* phase, where the changes to be performed are determined; these changes are formally represented during the *change representation* phase. The third phase is the *semantics of change* phase, in which the effects of the change(s) to the ontology itself are determined; during this phase, possible problems that might be caused to the ontology by these changes are also identified and resolved. The *change implementation* phase follows, where the changes are physically applied to the ontology, the ontology engineer is informed of the changes and the performed changes are logged. These changes need to be propagated to dependent elements; this is the role of the *change propagation* phase. Finally, the *change validation* phase allows the ontology engineer to review the changes and possibly undo them, if desired. This phase may uncover further problems with the ontology, thus initiating new changes that need to be performed to improve

[1] For more details on the architecture of the SWKM, see: http://athena.ics.forth.gr:9090/SWKM/index.html

the conceptualization; in this case, we need to start over by applying the change capturing phase of a new evolution process, closing the cyclic loop.

This paper focuses on the second and third phase (change representation and semantics of change), which are the most critical for ontology evolution [10]. Notice that during the change representation phase we determine the *requested* change (i.e., *what* should be changed), whereas during the semantics of change we determine the *actual* change (i.e., *how* the change should be performed). With respect to the five-step process described in Section 1, the change representation phase corresponds to the first two steps of our framework, whereas the semantics of change phase corresponds to the last three steps.

There is a rich literature that deals with the problem of ontology evolution. In general, two major research paths can be identified [4]. The first focuses on aiding the user performing changes in ontologies through some intuitive interface that provides a number of useful editing features; such tools resemble an ontology editor (and some of them are indeed ontology editors [13]), even though they often provide many more features than a simple ontology editor would. The second research path focuses on the development of automated methods to determine the effects and side-effects of any given update request (which correspond to phases 2 and 3 of [12]); this approach often borrows ideas from the related, and much more mature, discipline of belief change [6].

The first class of tools is more mature at the moment, but the second approach seems more interesting from a research point of view, as well as more promising; for this reason, it is gaining increasing attention during the last few years [4]. The two research paths are complementary, as results from the second could be applied to the first in order to further improve the quality of the front-end editing tools; similarly, automated approaches are of little use unless coupled with tools that address the practical issues related to evolution, like support for multi-user environments, transactional issues, change propagation, intuitive visual interfaces etc (i.e., the remaining four phases of [12]).

2.2 Motivation

Unfortunately, the above complementarity is not sufficiently exploited. Automated approaches (second research path) seem, in general, detached from real problems and are not easily adaptable for use in an ontology evolution tool; to our knowledge, there is no implemented tool that uses one of the algorithms developed by such approaches. On the other hand, editor-like tools (first research path) do not provide enough automation and employ ad-hoc methodologies to deal with the problems raised during an update operation; such ad-hoc methodologies cause several problems that are thoroughly discussed in Section 3.

Our approach is motivated by the need to develop a formal framework that will lead to an easily implementable ontology evolution algorithm. We would like our approach to enjoy the formality of the second class of tools, and use this formality as a basis that will provide guarantees related to the behavior of the implemented system, thus avoiding the problems related to the ad-hoc nature of existing practical methodologies. This paper is an attempt towards this end. In

this respect, the work presented here lies somewhere between the two research paradigms described above, sharing properties with both worlds.

More specifically, our approach could be viewed as belonging to the second class of works, in the sense that it results to a formal, theoretical model to address changes. This model is based on a formal framework that is used to describe the process of ontology evolution as addressed by current editor-like tools (so it is also related to the first class of works), and allows us to develop an abstract, general-purpose algorithm that provably performs changes in an automated and rational way for a variety of languages, under different parameters (validity model and ordering). Like other works of the second research path above, our work is focused on the "core" of the ontology evolution problem, namely the change representation and semantics of change phases. Issues related to change capturing, implementation of changes, transactional issues, change propagation, visualization, interfaces, validation of the resulting ontology etc are not considered in this paper.

On the other hand, our approach could be viewed as belonging to the first class of tools, in the sense that it results to an implemented tool, namely the Change Impact Service of the SWKM. Our general-purpose algorithm can be applied for any particular language and set of parameters that is useful for practical purposes; for the purposes of SWKM we set these parameters so as to correspond to the RDF language under the semantics described in [11]. Fixing these parameters also allows us to better present our approach, as well as to evaluate and verify its usefulness towards the aim of implementing an ontology evolution tool. In addition to the implementation of the general-purpose algorithm, our formal framework allows the development (and implementation) of special-purpose algorithms which are more suited for practical purposes; such algorithms provably exhibit the same behavior as the general-purpose one, so we can have formal guarantees as to their expected output. For reasons explained in Section 5, both the general-purpose and the special-purpose algorithms are implemented for the Change Impact Service of SWKM.

3 Evolution Process in Current Systems

In this section, we elaborate on the five steps we described in Section 1 and describe how some typical ontology evolution tools ([1, 5, 8, 15]) fit into this five-step process. In addition, we point out the problems that the ad-hoc implementation of these tools causes, and show how such problems could be overcome through the use of a formal framework, like the one described in Section 4.

3.1 Model Selection and Supported Operations

Obviously, the first step towards developing an evolution algorithm is the determination of the underlying representation model for the evolving ontology; this is what we capture in the first step of our 5-step process. Most systems assume a language supporting the basic constructs used in ontology development, like

class and property subsumption relationships, instantiation relationships and domain and range restrictions for properties.

The selection of the representation model obviously affects (among other things) the operations that can be supported; for example, OntoStudio [15] does not support property subsumption relations so all related changes are similarly overruled. Further restrictions to the allowable changes may be introduced by various design decisions, which may disallow certain operations despite the fact that they could, potentially, be supported by the underlying ontology model. For example, OntoStudio does not allow the manipulation of implicit knowledge, whereas OilED [1] does not support any operation that would render the ontology invalid (i.e., it does not take any actions to restore validity, but rejects the entire operation instead). The determination of the allowed (supported) update operations constitutes the second step of our 5-step process.

According to [12, 13], change operations can be classified into elementary (involving a change in a single ontology construct) and composite ones (involving changes in multiple constructs), also called atomic and complex in [14]. Elementary changes represent simple, fine-grained changes; composite changes represent more coarse-grained changes and can be replaced by a series of elementary changes. Even though possible, it is not generally appropriate to use a series of elementary changes to replace a composite one, as this might cause undesirable side-effects [12]; the proper level of granularity should be identified in each case. Examples of elementary changes are the addition and deletion of elements (concepts, properties etc) from the ontology. There is no general consensus in the literature on the type and number of composite changes that are necessary. In [12], 12 different composite changes are identified; in [9], 22 such operations are listed; in [14] however, the authors mention that they have identified 120 different interesting composite operations and that the list is still growing! In fact, since composite operations can involve changes in an arbitrary number of constructs, there is an infinite number of them. Although composite operations can, in general, be decomposed into a series of elementary ones, for ad-hoc systems this is not of much help, as the decomposition of a non-supported operation into a series of supported ones (even if possible) should be done manually.

The above observations indicate an important inherent problem with ad-hoc algorithms, namely that they can only deal with a predefined (and finite) set of supported operations, determined at design time. Therefore, any such algorithm is limited, because it can only support some of the potential changes upon an ontology, namely the ones that are considered more useful (at design time) for practical purposes, and, thus, supported.

3.2 Validity Model and Invalidity Resolution

It is obvious that a user expects his update request to be executed upon the ontology. Thus, it is necessary for the resulting ontology to actually implement the change operation originally requested, i.e., that the actual changes performed upon the ontology are a superset of the requested ones; this requirement will be called *success*.

The naive way to implement an update request upon an ontology would be to simply execute the request in a set-theoretic way. That would guarantee the satisfaction of the above principle (success); nevertheless, this would not be acceptable in most cases, because the resulting ontology could be invalid in some sense (e.g., if a class is removed, it does not make sense to retain subsumption relationships involving that class). Thus, another basic requirement for a change operation is that the result of its application should be a *valid* ontology, according to some validity model. This requirement is necessary in order for the resulting ontology to make sense.

Both the above principles are inspired by research on the related field of belief revision [2, 6], in which they are known as the Principle of Validity and Principle of Success respectively. The Principle of Success is well-defined, in the sense that we can always verify whether it is satisfied or not. The Principle of Validity however, depends on some underlying validity model, which is not necessarily the same for all languages (ontology models) and/or ontology evolution systems. Thus, each system should define the validity model that it uses. For example, do we accept cycles in the IsA hierarchy? Do we allow properties without a range/domain, or with multiple ranges/domains? Such decisions are included in the validity model determined in step 3 of our 5-step process. Notice that the validity model has a different purpose than the ontology model: the ontology model is used to determine what constructs are available for use in an ontology (e.g., IsAs), whereas the validity model determines the valid combinations of constructs in an ontology (e.g., by disallowing cyclic IsAs).

Determining how to satisfy the Principles of Success and Validity during a change operation is not trivial. The standard process in this respect is to execute the original update request in a naive way (i.e., by executing plain set-theoretic additions and deletions), followed by the initiation of additional change operations (called *side-effects*) that would guarantee validity. In principle, there is no unique set of side-effects that could be used for this purpose: in some cases, there is more than one alternatives, whereas in others there is none. The latter type of updates (i.e., updates for which it is not possible for both Success and Validity to be satisfied) are called *infeasible* and should be rejected altogether. For example, the request to remove a class, say C, and add a subsumption relationship between C and D at the same time would be infeasible, because executing both operations of the composite update would lead the ontology to an invalid state (because a removed class C cannot be subsumed by another class) and it can be easily shown that there is no way (i.e., side-effects) to restore validity without violating success for this update. The determination of whether an update is infeasible or not, as well as of the various alternative options (for side-effects) that we have for guaranteeing success and validity (for feasible updates) constitutes the fourth step of our 5-step process.

Let us consider the change operation depicted in Figure 1(a), where the ontology engineer expresses the desire to delete a class (B) which happens to subsume another class (C). It is obvious that, once class B is deleted, the IsAs relating B with A and C would refer to a non-existent class (B), so they should

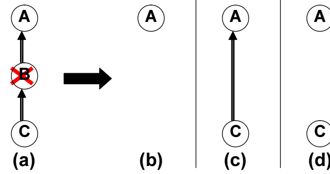

Fig. 1. Three alternatives for deleting a class

be removed; the validity model should capture this case, and attempt to resolve it. One possible result of this process, employed by Protégé [8], is shown in Figure 1(b); in that evolution context, a class deletion causes the deletion of its subclasses as well. This is not the only possibility though; Figures 1 (c) and (d), present other potential results of this operation, where in (c), B's subclasses are re-connected to its father, while in (d), the implicit IsA from C to A is not taken into account. KAON [5], for example, would give either of the three as a result, depending on a user-selected parameter.

In this particular example, both KAON and Protégé detect the invalidity caused by the operation and actively take action against it; however, the validity model employed by different systems may be different in general. Moreover, notice that an invalidity is not caused by the operation itself, but by the combination of the current ontology state and the operation (e.g., if B was not in any way connected to A and C, its deletion would cause no problems). Therefore, in order for a mechanism to propose solutions against invalidities, both the ontology and the update should be taken into account. Notice that the mechanism employed by Protégé, in Figure 1, identifies only a single set of side-effects, while KAON identifies three different reactions. This is not a peculiarity of this example; the invalidity resolution mechanism employed by Protégé identifies only a single solution per invalidity; this is not true for KAON and OntoStudio.

3.3 Action Selection

Since, in the general case, there are several alternative ways (i.e., sets of side-effects) to guarantee success and validity, we need a mechanism that would allow us to select one of the alternatives for implementation (execution). This constitutes the last component of an evolution algorithm (step 5). Such a mechanism is "pre-built" into systems that identify only a single possible action, like Protégé, but can be also parameterizable. KAON, for example, provides a set of options (called *evolution strategies*) which allow the ontology engineer to tune the system's behavior and, implicitly, indicate what is the appropriate invalidity resolution action for implementation per case. OntoStudio provides a similar customization over its change strategies.

Notice that our preference for the result of an operation reflects in a preference among the possible side-effects of the operation. For instance, if we prefer the result of Figure 1 (c), we can equivalently say that we prefer the (explicit)

addition of the (implicit) subsumption relation shown in (c) together with the deletion of the two initial IsAs as a side-effect to this operation, over the deletion of the two initial IsAs and class C, shown in (b), or just the deletion of the two IsAs, as in (d). Therefore, the evolution process can be tuned by introducing a preference ordering upon the operations' side-effects that would dictate the related choice (evolution strategy). Given that the determination of the alternative side-effects depends on both the update and the ontology, there is an infinite number of different potential side-effects that may have to be compared. Thus, we are faced with the challenge of introducing a preference mechanism that will be able to compare any imaginable pair of side-effects.

It is worth noting here the connection of this preference ordering with the well-known belief revision Principle of Minimal change [2] which states that the resulting ontology should be as "close" as possible to the original one. In this sense, the preference ordering could be viewed as implying some notion of relative distance between different results and the original ontology, as identified by the preference between these results' corresponding side-effects.

3.4 Discussion

To the best of authors' knowledge, all currently implemented systems employ ad-hoc mechanisms to resolve the issues described above. The designers of these systems have determined, in advance (i.e., at design time), the supported operations, the possible invalidities that could occur per operation, the various alternatives for handling any such possible invalidity, and have already pre-selected the preferable option (or options, for flexible systems like KAON) for implementation per case; this selection (or selections) is hard-coded into the systems' implementations.

This approach causes a number of problems. First of all, each invalidity, as well as each of the possible solutions to each one, needs to be considered individually, using a highly tedious, manual case-based reasoning which is error-prone and gives no formal guarantee that the cases and options considered are exhaustive. Similarly, the nature of the selection mechanisms cannot guarantee that the selections (regarding the proper side-effects) that are made for different operations exhibit a faithful overall behavior. This is necessary in the sense that the side-effect selections made in different operations (and on different ontologies) should be based on an operation-independent "global policy" regarding changes. Such a global policy is difficult to implement and enforce in an ad-hoc system.

Such systems face a lot of limitations due to the above problems. For example, OilED deals only with a very small fraction of the operations that could be defined upon its modeling, as any change operation that would be triggering side-effects is unsupported (e.g., the operation of Figure 1 is rejected). In Protégé, the design choice to support a large number of operations has forced its designers to limit the flexibility of the system by offering only one way of realizing a change; in OntoStudio, they are relieved of dealing with (part of) the complexity of the aforementioned case-based reasoning as the severe limitations on the expressiveness of the underlying model constrain drastically the number

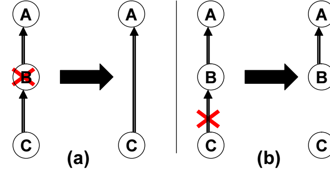

Fig. 2. Implicit knowledge handling in KAON

of supported operations and cases to consider. Finally, in KAON, some possible side-effects are missing (ignored) for certain operations, while the selection process implied by KAON's parameterization may exhibit invalid or non-uniform behavior in some cases. As an example, consider Figure 2, in which the same evolution strategy was set in both (a) and (b); despite that, the implicit IsA from C to A is only considered (and retained) in case (a).

Table 1. Summary of ontology evolution tools

			Protégé	KAON	OntoStudio	OilED	SWKM
Change Representation	Fine-grained Model (Step 1)		✓	✓	×	✓	✓
	Supported Operations (Step 2)	Elementary	✓	✓	✓	×	✓
		Composite	×	×	×	×	✓
Semantics of Change	Validity Model (Step 3)	Faithful	×	×	✓	✓	✓
		Complete	×	×	✓	×	✓
	Invalidity Resolution (Step 4)	No alternatives				✓	
		One alternative	✓				
		Many alternatives		✓			
		All alternatives			✓		✓
	Selection Mechanism (Step 5)	None	✓			✓	
		Per-case		✓	✓		
		Globally					✓

Table 1 summarizes some of the key features of ontology evolution systems, categorized according to the 5-step process introduced in this paper, and shows how each step is realized in each of the four systems discussed here, as well as in the Change Impact Service of SWKM, described in Sections 4, 5 below.

We argue that many of the problems identified in this section could be resolved by introducing an adequate evolution framework that would allow the description of an algorithm in more formal terms, as a modular sequence of choices regarding the ontology model used, the supported operations, the validity model, the identification of plausible side-effects and the selection mechanism. Such a framework would allow justified reasoning on the system's behavior, with-

out having to resort to a case-by-case study of the various possibilities. To the best of the authors' knowledge, there is no implemented system that follows this policy. In Section 4, we describe such a framework and specialize it for RDF ontologies.

4 A Formal Framework for RDF/S Ontology Evolution

Our evolution framework consists of a fine-grained modeling of ontologies (step 1), a description of how both elementary and composite operations can be handled in a uniform way (step 2), a validity model formalized using integrity rules (step 3), which also allow us to document how side-effects are generated (step 4), and, finally, a selection mechanism based on an ordering that captures the Principle of Minimal Change (step 5). This framework will be instantiated to refer to RDF updating, but can be used for many different declarative languages, by tuning the various parameters involved.

4.1 Model Selection, Supported Operations and Validity Model

The representation model we use in this paper is the RDF language, in particular the model described in [11]. For ease of representation, RDF constructs will not be represented in the standard way, but we will use an alternative representation, which, in short, amounts to mapping each statement of RDF to a First-Order Logic (FOL) predicate (see Table 2); this way, a class IsA between A and B, for example, would be mapped to the predicate: $C_IsA(A, B)$, while a triple denoting that the domain of a property, say P, is C, would be denoted by $Domain(P, C)$. Note that the standard alternative mapping (e.g., for IsA: $\forall x A(x) \rightarrow B(x)$) does not allow us to map assertions of the form "C is a class", and, consequently, does not allow us to handle operations like the addition or removal of a class, property, or instance (see [3] for more details on this issue). Notice that the same representation pattern can be used for other declarative languages as well, even though it is more suitable for simpler ones [3].

Table 2. Representation of RDF facts using FOL predicates

RDF triple	Intuitive meaning	Predicate
C rdf:type rdfs:Class	C is a class	$CS(C)$
P rdf:type rdf:Property	P is a property	$PS(P)$
x rdf:type rdfs:Resource	x is a class instance	$CI(x)$
P rdfs:domain C	domain of property	$Domain(P, C)$
P rdfs:range C	range of property	$Range(P, C)$
C_1 rdfs:subClassOf C_2	IsA between classes	$C_IsA(C_1, C_2)$
P_1 rdfs:subPropertyOf P_2	IsA between properties	$P_IsA(C_1, C_2)$
x rdf:type C	class instantiation	$C_Inst(x, C)$
x P y	property instantiation	$PI(x, y, P)$

We equip our FOL with closed semantics, i.e., admit the *closed world assumption* (CWA). This means that, for a set S and a formula p, if $S \nvdash p$, then $S \vdash \neg p$. We overload $\vdash$ relation so as to be applicable between two sets as well: for two sets S, S' it holds that $S \vdash S'$ iff $S \vdash p$ for all $p \in S'$. Let us denote by L the set of ground facts allowed in our model (e.g., $C_IsA(A, B), \neg CS(C)$), and L^+ the set of positive ground facts of L (e.g., $C_IsA(A, B)$).

An ontology is represented as a set of positive ground facts only, so an ontology is any set $O \subseteq L^+$. Given CWA, the definition of an ontology and FOL semantics, it follows that: (a) an ontology is always consistent (in the standard FOL sense), (b) a positive ground fact is implied by an ontology iff it is contained in it, and, (c) a negative ground fact is implied by an ontology iff its positive counterpart is not contained in it.

An *update* is any set of positive and/or negative ground facts, so an update is any set $U \subseteq L$. According to the Principle of Success, an update should be implemented upon the ontology. Implementing a positive ground fact contained in an update is easy: all we have to do is add it to the ontology. However, this is not true for negative ground facts, because negative ground facts cannot be contained in an ontology, by definition. By CWA and the property (c) above, we conclude that "including" a negative ground fact in an ontology is equivalent to removing its positive counterpart. Given this analysis, we conclude that positive ground facts in an update correspond to additions, while negative ones correspond to removals. This way of viewing updates allows us to express essentially any operation, because any operation can be expressed as a set of additions and/or removals of ground facts in our model. Thus, we put no constraints on the allowed (supported) update operations.

Our framework needs also to define its validity model in a formal way. Validity can in general be formalized using a set of integrity constraints (rules) upon the ontology; therefore, a validity model is a set R of generic FOL formulas, which correspond to the axiomatization of the constraints of the model. For technical reasons that will be made apparent later, we constrain R to contain only "$\forall\exists$" formulas. Notice that the validity constraints should: (a) capture the notion of validity in the standard sense (e.g., that class subsumptions should be applied between classes in the ontology) and (b) encode the semantics of the various constructs of the underlying language (RDF in our case), which are not carried over during the transition to FOL (e.g., IsA transitivity) [3]. The latter type of constraints is very important, in the sense that it forces an ontology to contain all its implicit knowledge as well in order to be valid.

Similar to our approach, the authors of [17] consider the case of updating a set of facts representing a knowledge base, under a set of well-formed constraints on this base. However this work supports rather naïve changes as it does not consider any side-effects for a change (storing the updates that violate any rules as *exceptions* to the latter) nor composite updates. So, instead of implementing a more sophisticated change mechanism the authors of [17] emphasize on minimizing the size of the knowledge base, in the face of an update. Another work which considers updating structured data under constraints is presented in

[16], where XML documents are automatically evaluated against a set of rules they should adhere to. However in case of invalidities, the process of updating the documents accordingly is left to be done manually. Therefore both of these works are essentially different from our approach as we develop an automated, parameterizable to its change policy, change mechanism, under a certain validity context (set of rules).

Table 3. Indicative list of validity rules

Rule ID/Name	Integrity Constraint	Intuitive Meaning
R3 *Domain* Applicability	$\forall x, y : Domain(x, y) \rightarrow PS(x) \wedge CS(y)$	Domain applies to properties; the domain of a property is a class
R5 *C_IsA* Applicability	$\forall x, y : C_IsA(x, y) \rightarrow CS(x) \wedge CS(y)$	Class IsA applies between classes
R12 *C_IsA* Transitivity	$\forall x, y, z : C_IsA(x, y) \wedge C_IsA(y, z) \rightarrow C_IsA(x, z)$	Class IsA is Transitive

Table 3 contains an indicative list of the rules we use for RDF [11] (see also [7] for a similar effort). Notice that the rules presented are only a parameter of the model; our framework does not assume any particular set of rules (in the same sense that it does not assume any particular ontology representation language). However, the task of defining the respective rules becomes increasingly complex as the expressive power of the underlying logic increases, so this technique is more useful for less expressive languages (like RDF) [3].

4.2 Formalizing Our Model

We now have all the necessary ingredients for our formal definitions. Initially, an update algorithm can be formalized as a function mapping an ontology (i.e., a set of positive ground facts) and an update (i.e., a set of positive and negative ground facts) to another ontology. Thus:

Definition 1. *An update algorithm is a function* $\bullet : L^+ \times L \mapsto L^+$.

An ontology is *valid* iff it satisfies the rules of the validity model R, i.e., iff it implies all rules in R. Thus:

Definition 2. *An ontology* O *is valid, per the validity model* R*, iff* $O \vdash R$.

As already mentioned, the Principle of Success implies that all positive ground facts in an update should be included in the result, whereas the positive counterparts of the negative ground facts in an update should not. Thus, any (positive or negative) ground fact p in an update U should be implied by the result of the change operation. Of course, this is true for feasible updates; for infeasible updates, by definition, there is no valid ontology that satisfies the above requirement. Therefore:

Definition 3. *An update U is called feasible, per the validity model R, iff there is a valid ontology O ($O \vdash R$) such that $O \vdash U$. An update U is called infeasible iff it is not feasible.*

Definition 4. *Consider a language L, a set of validity rules R and an update algorithm $\bullet : L^+ \times L \mapsto L^+$. Then:*

- *The algorithm $\bullet$ satisfies the Principle of Success iff for all valid ontologies $O \subseteq L^+$ and all feasible updates $U \subseteq L$, it holds that $O \bullet U \vdash U$.*
- *The algorithm $\bullet$ satisfies the Principle of Validity iff for all valid ontologies $O \subseteq L^+$ and all feasible updates $U \subseteq L$, it holds that $O \bullet U$ is a valid ontology.*

Notice that the above definition does not handle the cases where the input ontology is not valid to begin with, or when the update is infeasible; these are limit cases that will be handled separately later.

4.3 Invalidity Resolution and Action Selection

As already mentioned, the raw application of an update would guarantee success but could often violate validity (i.e., it could violate an integrity constraint). For example, under the validity context of Table 3, the raw application of the class deletion of Figure 1 would violate rule R5. In such cases, we need to determine the various options that we have in order to resolve the invalidity.

The formalization of the validity model using rules has the important property that, apart from detecting invalidities, it also provides a straightforward methodology to determine the various available options for resolving them. In effect, the rules themselves and the FOL semantics indicate the appropriate side-effects to be taken when an invalidity is detected. In the example with the class deletion (Figure 1), rule R5 implies that, in order to restore validity after the removal of class B (denoted by $\neg CS(B)$), we must delete the IsAs involving B.

In the general case, detecting and restoring an invalidity would require a FOL reasoner; however, our assumption that an ontology is a set of positive ground facts and that a rule is a "$\forall\exists$" formula, allows us to develop a much more efficient way. In particular, a "$\forall\exists$" rule can be equivalently rewritten as the conjunction of a set of *subrules*, where each subrule is a formula of the form $\forall \bigvee \exists$ (see Table 4). Thus, by definition, an ontology O is valid iff it implies all subrules of all rules of the validity model. A subrule is implied by O iff, for all possible variables, at least one of the constituents of the disjunction is true (i.e., implied). Thus, a subrule can be violated iff a previously true constituent of the subrule is, due to the update, rendered false (i.e., not implied) and there is no other true constituent of the subrule. Similarly, the possible ways to render a violated subrule true should be chosen among all the constituents of the subrule, i.e., we should select one of the constituents of the subrule to be rendered true (through a side-effect); notice that the selected constituent should not be the one that was rendered false by the update itself (or else we would violate success).

Let us explain this process using an example. Consider rule R5, which is broken down into two subrules, as shown in Table 4. Let's consider subrule R5.1;

Table 4. Breaking rules into subrules

Rule ID/Name	Subrules of the rule
R3 *Domain* Applicability	$R3.1 : \forall x, y : \neg Domain(x, y) \vee PS(x)$ $R3.2 : \forall x, y : \neg Domain(x, y) \vee CS(y)$
R5 C_IsA Applicability	$R5.1 : \forall x, y : \neg C_IsA(x, y) \vee CS(x)$ $R5.2 : \forall x, y : \neg C_IsA(x, y) \vee CS(y)$
R12 C_IsA Transitivity	$R12.1 : \forall x, y, z :$ $\neg C_IsA(x, y) \vee \neg C_IsA(y, z) \vee C_IsA(x, z)$

this subrule is satisfied iff for all variables x, y, it either holds that $\neg C_IsA(x, y)$, or it holds that $CS(x)$. If we remove a class (say B, denoted by $\neg CS(B)$) which previously existed in the ontology (cf. Figure 1), we should verify that subrule R5.1 is still true. This practically amounts to verifying that no class IsA starting from B exists in the ontology, i.e., that $\neg C_IsA(B, y)$ is true for all y. If any such y exists (say $y = C$), then we must remove the respective IsA (i.e., $\neg C_IsA(B, C)$ should be recorded as a side-effect).

Rule R12 is similar: R12.1 (which is the only subrule of R12) can be violated by, e.g., the addition of an IsA (say $C_IsA(C, B)$). This could happen if, for example, an ontology contains $C_IsA(B, A)$, but not $C_IsA(C, A)$ (cf. Figure 3). To see this, set $x = C, y = B, z = A$ in R12.1, Table 4. The difference with the previous case is that now the violation can be restored in two different ways: either by removing $C_IsA(B, A)$, or by adding $C_IsA(C, A)$ (i.e., either $\neg C_IsA(B, A)$ or $C_IsA(C, A)$ could be selected as side-effects).

Notice that the selected side-effects are updates themselves, so they are enforced upon the ontology by being executed along with the original update; moreover, they could, just like any update, cause additional side-effects of their own. Another important remark is that, in some cases (e.g., R5.1), the invalidity resolution mechanism gives a straightforward result, in the sense that we only have one option to break the invalidity; in other cases (e.g., R12.1), we may have more than one alternative options. In the cases where we have different alternative sets of side-effects to select among, a mechanism to determine the "best" option, according to some metric, should be in place. In Section 3, we showed that our "preference" among the side-effects can be encoded using an ordering; given such an ordering (say $<$), all we need to do is find the minimal set of side-effects (with respect to $<$) among all possible ones and implement it.

As usual, our framework does not depend on any particular ordering. For technical reasons however, not all orderings can be employed for this purpose. In particular, to guarantee the rationality of the results, the ordering should depend on the underlying ontology as well (e.g., it is generally accepted that the removal of a general class is more "severe" than the deletion of a more specific class, but this criterion implies knowing the position of the class in the class hierarchy of the ontology). In addition, the ordering should be transitive and total; furthermore, it should be monotonic with respect to $\subseteq$ (i.e., $U \subseteq U'$ implies $U \leq U'$). Moreover, it should not be affected by void changes: for example, the

addition of class C is a void operation in an ontology that already contains C and the removal of class D is a void operation in an ontology that does not contain D, so the inclusion of $CS(C)$ (or $\neg CS(D)$, respectively) in the side-effects of an update upon the above ontologies should not affect the "mildness" of the update. Finally, the ordering should be antisymmetric, modulo void operations (i.e., two updates have the same "mildness" iff their non-void operations are identical). We will call *update generating* an ordering satisfying these properties.

In our implementation, the proposed ordering is based on the ordering shown in Table 5 among the 18 positive and negative predicates. This ordering is expanded to refer to updates (i.e., sets of ground facts) using the general idea that an update U_1 is "preferable" or "better" than U_2 (denoted by $U_1 < U_2$) iff the "worst" predicate used in U_1, is "better" than the "worst" predicate used in U_2 where the predicates' relative preference is determined by the order shown in Table 5. Ties are resolved using cardinality considerations and/or the relative "importance" of the predicate's arguments in the original ontology, according to certain rules that determine "importance". Further details are omitted due to space limitations. It can be proven that our ordering is update-generating.

Table 5. Ordering of predicates

$PI < C_Inst < P_IsA < C_IsA < \neg PI < \neg C_Inst < \neg P_IsA < \neg C_IsA < \neg Domain < \neg Range < \neg CI < \neg PS < \neg CS < Domain < Range < CI < PS < CS$

4.4 Rational Ontology Evolution Algorithms

Now consider an update U applied upon an ontology O per the update algorithm $\bullet$, returning $O \bullet U$. The question is, what were the effects and side-effects that were applied upon O to get $O \bullet U$? The restriction that ontologies contain only positive ground facts is extremely helpful in this respect too. In particular, we can define the *Delta* between two ontologies as follows:

Definition 5. *Consider two ontologies* $O_1, O_2 \subseteq L^+$. *The Delta between* O_1 *and* O_2 *is defined as* $Delta(O_1, O_2) = \{p \mid p \in O_2 \setminus O_1\} \cup \{\neg p \mid p \in O_1 \setminus O_2\}$.

Notice that the result of *Delta* is an update, i.e., $Delta(O_1, O_2) \subseteq L$; given the above definition, the actual set of effects and side-effects that were applied upon O to get $O \bullet U$ is just $Delta(O, O \bullet U)$. Notice that $Delta(O, O \bullet U)$ will just return the non-void operations that led from O to $O \bullet U$; this is not a problem, as void operations do not affect the ordering. Given this *Delta* function, the Principle of Minimal Change can be formalized by requiring that an update algorithm should return an ontology $O \bullet U$ such that $Delta(O, O \bullet U)$ is minimal compared to $Delta(O, O')$ for all other possible results O'.

Of course, we need to specify what are the other "possible results", which, as already mentioned, are the ones that satisfy the Principles of Success and Validity. Thus, our formal definition of the Principle of Minimal Change should be coupled with the other principles. We will therefore define a *rational* update algorithm to be one that satisfies the Principles of Success and Validity, and, among all the possible results that satisfy these two principles, it selects the one that has the minimal impact upon the original ontology (Principle of Minimal Change). Notice that there are certain limit cases which need to be handled separately, i.e, the case when the original ontology (to be updated) is invalid, and the case when the update itself is infeasible:

Definition 6. *Consider a language L, a validity model R, an update-generating ordering $<$ and an update algorithm $\bullet : L^+ \times L \mapsto L^+$. Then the algorithm $\bullet$ is called rational iff it satisfies the following requirements for all $O \subseteq L^+, U \subseteq L$:*

Limit Cases: *if O is not valid or U is infeasible, then $O \bullet U = O$*
General Case: *if O is valid and U is feasible, then $\bullet$ satisfies the following:*
Principle of Success: $O \bullet U \vdash U$.
Principle of Validity: *$O \bullet U$ is valid.*
Principle of Minimal Change: *For any O' such that $O' \vdash U$ and O' is a valid ontology, it holds that $Delta(O, O \bullet U) \leq Delta(O, O')$.*

Note that rationality depends on the model (which determines L and L^+), the validity rules (for the Principle of Validity) and the ordering (for the Principle of Minimal Change). Therefore, there is no "universally rational update algorithm", but rationality depends critically on these parameters.

5 Algorithms

5.1 General-purpose Algorithm

We will now show how one can use the above formal framework in order to develop a rational evolution algorithm (which is shown in Table 6). Let us consider the update example of Figure 3. Our original update is $U = \{C_IsA(C, B)\}$, denoting that an IsA between C and B should be added. We first need to check whether this update will violate any rule (line 4.1); as mentioned in Section 4, this can be done by checking against all subrules in which $\neg C_IsA$ appears. In general, several rules may be violated, in which case we process them in any order (line 4.2). In our example, it can be verified that the addition of $C_IsA(C, B)$ will only violate subrule R12.1 (IsA transitivity), for $x = C, y = B, z = A$. This is true because the addition of $C_IsA(C, B)$ should cause the addition of the implicit knowledge $C_IsA(C, A)$ as well. This option is the standard way of satisfying transitivity, but our rule also gives us the alternative to remove the old IsA between B and A (to prevent the transitivity rule from firing).

In order to explore all alternatives regarding the possible side-effects, the comparison (using $<$) between the first and the second option is postponed until the

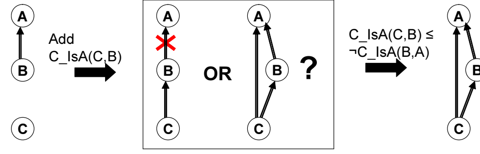

Fig. 3. Adding a class subsumption

full set of side-effects has been computed. Therefore, at this point, the algorithm suggests two different alternative updates, one per possible side-effect, namely $U_1 = \{C_IsA(C,B), C_IsA(C,A)\}$ and $U_2 = \{C_IsA(C,B), \neg C_IsA(B,A)\}$ (line 4.2.1). Then, the algorithm recursively calls itself twice (once for U_1 and once for U_2). Both calls will indicate no further side-effects, as there are no further rules violated; in the general case, the side-effects could have side-effects of their own, so the recursion should continue until no further side-effects exist. Once all recursions stop, the returned sets of side-effects are compared using $<$ and the minimal is selected for implementation (line 4.2.2). In this case, the first option (i.e., U_1) is the "best", according to $<$ (see Table 5), i.e., the IsA between C and A should be added; this indeed sounds like the most natural result, but it could be different if the ordering was different.

If, during the recursion, the so-far processed predicates turn out to contradict each other (line 1), then the particular branch of execution will obviously not lead to an acceptable solution, so the special value *infeasible* is returned; if all branches return *infeasible*, then the entire update is infeasible (and the recursive process will also return *infeasible*). The same special value is returned by certain branches (line 2) when their cost is predicted to be too large to be an acceptable solution, so there is no point in exploring them further.

Table 6. General-purpose algorithm

Input: Model, Rules, Ordering <, Update U, Ontology O
WHILE there exist unprocessed predicates in U execute the following steps:
(1) If the predicates that have been processed so far contradict each other, return INFEASIBLE
(2) If the total cost of the union of the predicates processed so far and the remaining predicates (in U) is larger than the best solution found so far, return INFEASIBLE
(3) Select (arbitrarily) an unprocessed predicate in U, say P
(4.1) IF there is no rule violated by P, THEN mark P as processed, add P to the side-effects of U and recursively call the algorithm using the same U
(4.2) ELSE select (arbitrarily) one violated rule, say R
(4.2.1) FOR each possible way to resolve the violation of R, add the respective predicates as side-effects in U and recursively call the algorithm using the new U
(4.2.2) When recursion returns compare (using <) the returned side-effects and return the "best" to the caller
Output: Update U enriched with its side-effects

Notice that the general algorithm (Table 6) is applicable for any language L (i.e., ontology model), validity model R and ordering $<$ and that several details of the algorithm have been brushed out. The general idea is that the case-based reasoning performed manually in other systems is now in-built in the algorithm, so it is performed automatically and in a parameterizable way. The algorithm's complexity depends on its parameters, namely the language, validity model and ordering; for the particular parameters used for RDF (described above), termination can be guaranteed:

Theorem 1. *For the language, validity rules and ordering described in Section 4, the algorithm of Table 6 terminates for any input O, U.*

Termination is guaranteed by the form of the rules and the ordering (cost model) used. In particular, it can be shown that, whenever there exists a non-terminating recursive path (branch), there exists also a terminating one that is significantly less costly. By carefully choosing the processing order of the various side-effects (line 4.2.1), we can guarantee that the non-terminating branches will be pruned in line 2, before jeopardizing termination.

The algorithm described in Table 6 returns the effects and side-effects of the original update, or the special value *infeasible*. The end result of this recursive algorithm can then be trivially applied upon the original ontology, by simply adding every positive ground fact of the output to the ontology, and removing the positive counterpart of any negative ground fact of the output from the ontology. The result will be a valid ontology which should be returned as the result of the update. The following can be shown:

Theorem 2. *For any given language, validity rules and update-generating ordering, if the algorithm described above terminates, then it implements a rational change operation.*

The complete proof of the above theorem is quite complicated and technical, so we provide only a short sketch. Principle of Success is guaranteed by the fact that our algorithm considers all the predicates in U, and all such predicates are added to the side-effects of U (line 4.1). The Principle of Validity is guaranteed as well: the process cannot end unless all violated rules (identified in line 4.2) are restored (line 4.2.1). Finally, the Principle of Minimal Change is guaranteed in line 4.2.2: the recursive character of the algorithm will open up several different branches, each of them spawned by a different way to restore a particular rule violation. Upon returning of each branch, the calculated cost of each branch is compared (line 4.2.2) and only the best is kept; notice that the comparison is made at a position where the entire branch has been explored, so we know its total cost and can guarantee that no ignored branch can have "minimal" cost (so it can't be an acceptable solution). The following corollary is immediate:

Theorem 3. *For the language, validity rules and ordering described in Section 4, the algorithm of Table 6 terminates for any input $O \subseteq L^+, U \subseteq L$ and it can be used to implement a rational change operation.*

5.2 Special-purpose Algorithms

A downside of the generality enjoyed by the algorithm of Table 6 is that it is not efficient. To remedy this problem, we can develop simpler, special-purpose algorithms, for the particular application that we are interested in (RDF in our case). These "instantiations" are much faster than the general algorithm, but can still be proven equivalent to it, i.e., formally sustained. Thus, we can guarantee that they exhibit the expected/desired behavior, by verifying them against the general-purpose algorithm. Notice that these special-purpose algorithms are similar to ad-hoc methodologies employed by other systems; however, using our formal framework and results, one can verify in a straightforward way the correctness (rationality) of those algorithms (see Theorem 4). Moreover, the general algorithm could still be used to implement any possible, unforseen operation.

Table 7 shows, as an example, one such special-purpose algorithm for the removal of a class from an ontology. Notice that some lines of the algorithm (e.g., (1.4.1)-(1.4.4)) would spawn other special-purpose algorithms for executing certain operations (in our case, the removal of IsAs, instantiation links etc), thus, possibly, incurring further side-effects. For this reason, similar algorithms have been developed for other operations, but are omitted due to space limitations.

Table 7. Special-purpose algorithm: remove class C from ontology O

Remove class C:
(1) If class C is in O THEN
(1.1) Remove all class IsA relationships deriving from C
(1.2) Remove all class IsA relationships arriving in C
(1.3) Remove all instantiation links between a resource and C
(1.4) FOR every property P whose range/domain is C
(1.4.1) Remove all property IsA relationships deriving from P
(1.4.2) Remove all property IsA relationships arriving in P
(1.4.3) Remove all instantiation links of P
(1.4.4) Remove P and the information on its range/domain
(1.5) Remove C

Theorem 4. *Consider the language, validity rules and ordering described in Section 4. Then for $U = \{\neg CS(C)\}$ and any $O \subseteq L^+$, the output of the algorithm in Table 6 is the same as the output of the algorithm in Table 7.*

The above theorem can be easily shown by exhaustively considering all the different rule violations that the update under question would cause (by scanning the validity rules for violations); this would verify that the behavior of the special-purpose algorithm is identical to the general-purpose one for the particular order considered. Similarly to the other ontology evolution systems, our special-purpose algorithms cannot handle all possible update requests. However, we can always resort to the general-purpose algorithm if the requested operation is not supported by any special-purpose algorithm. Currently, we have devised

and implemented one special-purpose algorithm for each elementary operation, but we plan to develop more, in order to handle certain useful composite operations. The selection whether to use a special-purpose algorithm or the general-purpose one is made by the system itself, in a transparent manner to the user.

6 Conclusion

In this paper, we identified several difficulties associated with the development of ad-hoc ontology evolution algorithms. We decomposed the process of coping with ontology evolution into 5 discrete steps. This way, devising an ontology evolution algorithm is reduced to the process of instantiating each step in a modular way. To this end, we presented a formal framework with the aid of which an evolution algorithm can be materialized as a set of adequate parameterizations, as follows:

1. The ontology representation model and its mapping to FOL.
2. The definition of the allowed change operations in the model. Notice that this is not necessary, as the framework is general enough to support any update, but we may want to disallow certain operations for some application.
3. The validity rules that allow us to detect invalidities as well as to determine how the invalidities can be resolved.
4. The preference ordering that encodes the selection mechanism.

Parameters 1,2 and 4 of our framework correspond to steps 1,2 and 5 respectively. The third parameter corresponds to the validity context, based on which our framework instantiates steps 3 and 4. Once these parameters are set, we can apply the general algorithm presented in Table 6 to perform any change. For efficiency reasons, it may be useful to generate simpler special-purpose algorithms based on the general one. This can be done only for specific instantiations of the above parameters, as in the case study of RDF updating presented here.

Our method exhibits a faithful behavior with respect to the various choices involved, regardless of the particular ontology or update operation at hand. It has a formal foundation, issuing a solid, consistent and customizable method to handle any type of change operation, including updates that have not been considered at design time. Our framework is modular and extensible in the sense that it could work with any language, rules and/or ordering given.

As already mentioned, the presented algorithms have been implemented for the Change Impact Service of SWKM, and the initial results are promising. In the future, we plan to identify and optimize the most commonly used update operations. In addition, we plan to verify the effectiveness of our proposed ordering using experiments with real users.

7 Acknowledgements

This work was partially supported by the EU projects CASPAR (FP6-2005-IST-033572) and KP-Lab (FP6-2004-IST-4).

References

1. S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: a Reason-able Ontology Editor for the Semantic Web. *Proceedings of the Joint German/Austrian Conference: Advances in Artificial Intelligence (KI-01)*, 2001.
2. M. Dalal. Investigations Into a Theory of Knowledge Base Revision: Preliminary Report. *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, pp. 475–479, 1988.
3. G. Flouris. On the Evolution of Ontological Signatures. *Proceedings of the Workshop on Ontology Evolution (OnE-07)*, 2007.
4. G. Flouris, D. Manakanatas, H. Kondylakis, D. Plexousakis, and G. Antoniou. Ontology Change: Classification and Survey. *Knowledge Engineering Review (KER)*, to appear.
5. T. Gabel, Y. Sure, and J. Voelker. KAON-Ontology Management Infrastructure. *SEKT informal deliverable*, 3(1), 2004.
6. P. Gärdenfors. Belief Revision: An Introduction. In: P. Gärdenfors, (ed.) *Belief Revision*, pp. 1–20, Cambridge University Press, 1992.
7. S. Munoz, J. Perez, and C. Gutierrez. Minimal Deductive Systems for RDF. In *Proceedings of the 4th European Semantic Web Conference (ESWC-07)*, 2007.
8. N. Noy, R. Fergerson, and M. Musen. The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility. *Lecture Notes in Artificial Intelligence (LNAI)*, 1937:17–32, 2000.
9. N. Noy, and M. Klein Ontology Evolution: Not the Same as Schema Evolution. *Knowledge and Information Systems*, 6(4), pp. 428–440, also available as SMI technical report SMI-2002-0926, 2004.
10. G. Qi, W. Liu, and D.A. Bell. Knowledge Base Revision in Description Logics. *Proceedings of the 10th European Conference on Logics in Artificial Intelligence (JELIA-06)*, 2006.
11. G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and Minimization of RDF/S Query Patterns. In *Proceedings of the 4th International Semantic Web Conference (ISWC-05)*, 2005.
12. L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven Ontology Evolution Management. *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management (EKAW-02)*, 2002.
13. L. Stojanovic, and B. Motik. Ontology Evolution Within Ontology Editors. *Proceedings of the OntoWeb-SIG3 Workshop*, pp. 53–62, 2002.
14. H. Stuckenschmidt, and M. Klein. Integrity and Change in Modular Ontologies. *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, 2003.
15. Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke. OntoEdit: Collaborative Ontology Development for the Semantic Web. *Proceedings of the 1st International Semantic Web Conference (ISWC-02)*, 2002.
16. M. Benedikt, G. Bruns, J. Gibson, R. Kuss, and A. Ng, 'Automated update management for XML integrity constraints', *Proc. Workshop on Programming Languages for XML (PLAN-X)*, (2002).
17. D. Laurent, V. Phan Luong, and N. Spyratos, 'Updating intensional predicates in deductive databases', *Data & Knowledge Engineering*, **26**(1), 37–70, (1998).