Tempus fugit*

Towards an Ontology Update Language

Uta Lösch, Sebastian Rudolph, Denny Vrandečić, and Rudi Studer

Institut AIFB - Universität Karlsruhe (TH)
Karlsruhe, Germany
{uhe,sru,dvr,rst}@aifb.uni-karlsruhe.de

Abstract. Ontologies are used to formally describe domains of interest. As domains change over time, the ontologies have to be updated accordingly. We advocate the introduction of an Ontology Update Language that captures frequent domain changes and hence facilitates regular updates to be made in ontologies. We thoroughly discuss the general design choices for defining such a language and a corresponding update framework. Moreover, we propose a concrete language proposal based on SPARQL Update and provide a reference implementation of the framework.

1 Introduction and Motivation

Ontologies (see [12] for a comprehensive overview) are formal specifications of the knowledge about a domain of interest. They constitute one of the central concepts in the field of Semantic Web technologies and facilitate information integration and exchange as well as semantic search. Usually, their expressive power exceeds that of traditional databases and allows to infer new information that is not explicitly present in the specification but a logical consequence of it (the so-called *implicit knowledge*).

Currently, the most commonly used ontology languages are RDF and OWL, both standardized by the World Wide Web Consortium (W3C). RDF ([7], in particular the RDF Schema extension) constitutes a so-called "leightweight" ontology language, providing basic modeling features for assertional (instance-related) and terminological knowledge handling classes, binary relations (so-called properties), hierarchies of classes (also referred to as taxonomies) and properties as well as domain and range specifications for properties. OWL (the Web Ontology Language, [8]) constitutes the other well-established knowledge representation formalism for the Semantic Web, based on Description Logics [1], a family of decidable yet expressive logics for which highly optimized inferencing

* Supported by the Deutsche Forschungsgemeinschaft (DFG) under the ReaSem project and the Graduate School Information Management and Market Engineering and by the European Commission through the IST project ACTIVE (ICT-FP7-215040).

implementations exist. Syntactically, OWL can be represented in RDF(S) where certain vocabulary elements impose stronger restrictions on the semantics.

Following the uptake of ontology-based technologies by industry, the comprehensive modeling of complex domains will become necessary. We expect future ontologies to rarely be developed by a single person starting from scratch. Rather collaborative design and development of ontologies and continuous refinement will become the usual scenarios for which elaborate methodologies and appropriate tool support will become crucial (see for example [15, 14]).

In this spirit, *ontology change* is one of the fundamental issues to be addressed. Obviously, there are diverse reasons for changing an ontology. We argue that it is helpful to conceptually distinguish two cases by answering the question:

Is the ontology changed due to an according change in the described domain?

Note that there are many cases where the answer to that question would be no: an ontology change might be the consequence of the discovery of modeling errors (*ontology repair*) or the acquisition of new additional domain knowledge (*ontology amendment*). Obviously, this type of ontology changes reflects a change in the way the modeler conceives or formalizes the domain of interest. As an example, consider the case that new findings in genetics might imply that the taxonomy of living beings has to be corrected in order to properly reflect the current knowledge of the real situation. A lot of research has been devoted to this kind of ontology change (employing techniques from belief revision, knowledge acquisition, ontology learning and ontology evolution to name just a few). We propose the term *ontology refinement* to subsume all those activities.

As opposed to those, we will be concerned with the task that we refer to as *(temporal) ontology update*: changes to the ontology might become necessary as the underlying domain changes over time. As time passes, the state of affairs in the domain like a person's employer or her academic title may change. However, such changes are not restricted to assertional knowledge, but may also concern the schema. As an example consider EU membership: it is expected that additional countries become member of the EU, thereby changing the terminological definition of the class of EU citizens.

In many cases it will be possible to come up with *change patterns* which describe typical ways in which a domain may evolve. For example, a domain individual recorded as underage may turn into an adult, while the opposite is impossible. While most of these patterns will probably deal with the update of assertional knowledge, they may also occur on the schema level (as an example consider a country becoming member of the EU).[1]

The observation that such change patterns exist leads to the idea to formally specify typical ways in which an ontology may be updated over time. As an example, in a biological domain, an individual might cease to be member of the class `Caterpillar` and become member of the class `Butterfly` instead. These update specifications may concern schema knowledge as well as fact knowledge.

[1] While here we focus on change patterns that reflect temporal and domain-specific changes, we are aware that such patterns may be identified beyond this use case (our approach may thus be applicable in other change scenarios, too).

On a more general level, update specifications allow to encode process knowledge and associate it with the ontology, such that it can be used for updates.

The specifications can be seen as operational descriptions how to update an ontology as a consequence to information entered into the system. However, in contrast to common Ontology Evolution approaches [13], we propose to base those updates on domain specific knowledge about temporal changes. Using the above mentioned information, a natural reaction to asserting that an individual `willie` is now a class member of `Butterfly` would be to also retract the information that `willie` is an instance of `Caterpillar` [3].

This way, an ontology change requested by a person responsible for ontology maintenance can be supplemented by additional changes. This allows to prevent modeling flaws that might occur due to only partially entered information. Generally, an ontology update specification allows constraining ontology changes to clearly defined, foreseen cases in a domain-specific way.

Then, frequent maintenance or update tasks can be transferred from knowledge engineers (roughly: the "ontology administrator") to knowledge workers (possibly formally less skilled users in charge of monitoring changes in the domain of interest and transferring them into the ontology) while minimizing the risk of introducing errors. Like the actual ontology, the according update specification has to be created by a knowledge engineer who is also in charge of ontology refinement activities as well as addressing unforeseen changes not anticipated by the update specification that might become necessary.

In this paper, we elaborate on our idea of a framework and a language for ontology updates. Section 2 explains our design choices when working out the overall and more detailed aspects of an ontology update language. In Section 3 we briefly sketch the architecture of a system where ontology update specifications are employed. Section 4 provides the formal specification of our proposed Ontology Update Language and Section 6 describes the details of our reference implementation of an according ontology update system. In Section 5 the benefits of our approach will be illustrated by means of a small example before we conclude and provide directions for future research in Section 7.

2 Design Choices

In this section, we review the major choices and questions to be addressed when designing a framework for ontology update management.

2.1 Ontology-inherent Temporal Knowledge vs. External Specification

One approach to capture temporal changes in a domain is to use logic formalisms that allow for their description inside ontological specifications. There is a plethora of formalisms and approaches such as temporal logics or situation calculi that provide a logic-inherent way of describing temporal and dynamic

phenomena in the domain. Clearly, these approaches have advantages whenever the intended use of the ontology includes reasoning over domain changes (maybe even planning). However, besides the more complicated formalisms, a usual drawback of such formalisms is the high reasoning complexity.

Note that our goal is much more moderate: from the above described, it becomes clear that we aim at designing an operational formalism that – given a change request – deterministically comes up with an updated ontology in a timely manner. Moreover, we would like to stick to the usual approach that an ontology encodes a static description of the domain, hence every state of the actual ontology should be considered a kind of "snapshot".

Therefore, we adopt an approach keeping the actual ontology and the according update specification distinct, which also enables to use off-the-shelf reasoners for dealing with the ontology part.

2.2 Unguided Belief Revision vs. Guided Update

Although most of the work done in the area of belief revision has dealt with scenarios of ontology refinement (see [6] for an RDFS-based formal framework and [10] for a survey), some proposals have been made to also address the update scenario [5]. Notwithstanding, belief revision approaches try to resolve inconsistencies that were introduced by an update. However, many changes in the ontology will not lead to an inconsistent state, thus no additional changes are performed. Therefore, the applicability of belief revision is restricted to formalisms expressive enough to cause inconsistencies. While this is certainly the case for OWL, causing "meaningful" inconsistencies in RDF(S) is virtually impossible.[2] To a certain extent, this fallback can be mitigated by adding additional constraints beyond RDF(S) on top of an RDF(S) knowledge base[3] [6].

As another downside of belief revision, note that the strategies to restore consistency do not take domain specifics into account. To illustrate that, consider the following example: let a knowledge base contain the disjointness of the classes `Adult` and `Child` and the fact that `Peter` belongs to the class `Child`. If we now add that `Peter` is also an instance of the class `Adult`, the knowledge base becomes inconsistent and (if configured appropriately) a belief revision approach would retract `Child(Peter)`, as newly added facts override those already present. While this is the desired behavior, re-adding `Child(Peter)` to the new knowledge base would lead to the deletion of `Adult(Peter)` irrespective of the actual irreversibility of this development in the described domain. Clearly, a more appropriate "reality-aware" reaction of an update mechanism would be to reject the second change request.

As opposed to belief revision, our approach aims at preventing inconsistent ontology states that might arise from incomplete change requests. To this end,

[2] More precisely: in RDF(S) inconsistencies can only be provoked by so-called XML-clashes, which is more a datatype-related unintentional peculiarity than a design feature.

[3] Note that we use the terms ontology and knowledge base interchangeably throughout the paper.

change requests are completed by further ontology changes based on specified knowledge about how a domain may develop. This way, consistency can be preserved; as a worst case, the change request can be denied.

Note however, that those two approaches are not mutually exclusive but could be combined: in case of a change request not matching any of the anticipated update patterns, applying a belief revision "fallback solution" might sometimes be preferable to simply rejecting the request.

2.3 Syntactic vs. Semantic Preconditions

In most cases, the best way to react to a change request will depend on the current state of the ontology. Hence it is crucial to provide the opportunity to formulate respective preconditions for triggering changes. There are essentially two distinct kinds of checks that can be done against an ontology: *semantic* and *syntactic*[4] ones.

If some changes should be made depending on the validity of some statement in the ontology, we have to employ reasoning in order to decide whether the statement is logically entailed by the given information. As a special case of this, one could diagnose whether an intended change would turn the ontology inconsistent and reject the requested change on these grounds. Semantic checks provide the more thorough way of testing the knowledge contained in an ontology, however the reasoning to be employed may be expensive with respect to memory and runtime.

The alternative would be to just syntactically determine whether certain axioms are literally contained in a knowledge base. This would be less expensive than the semantic approach. Yet usually, there are many possible ways to syntactically express one piece of semantic information making a naïve syntactic "pattern matching" approach problematic at best. One way to mitigate that problem while still avoiding to engage in heavy-weight reasoning would be to syntactically normalize the ontology and the change request. That is, the ontology is transformed into a semantically equivalent, but syntactically more constrained form, facilitating to identify and manipulate pieces of semantic information by purely syntactic analyses.

Since both kinds of preconditions are useful under different circumstances, we argue that an ontology update formalism should offer both options leaving to the knowledge engineer to decide which one should be used in a specific case.

2.4 Change Feedback

It has to be expected that the changes mediated by an ontology update specification might not be directly obvious for the knowledge worker. However, it is clearly crucial to ensure that the system's behavior is as transparent as possible to the knowledge worker. For this reason, feedback about the automated

[4] In our actual proposal, we refrain from taking a purely syntactic approach, but rather the structural level of the RDF graph that is used by SPARQL.

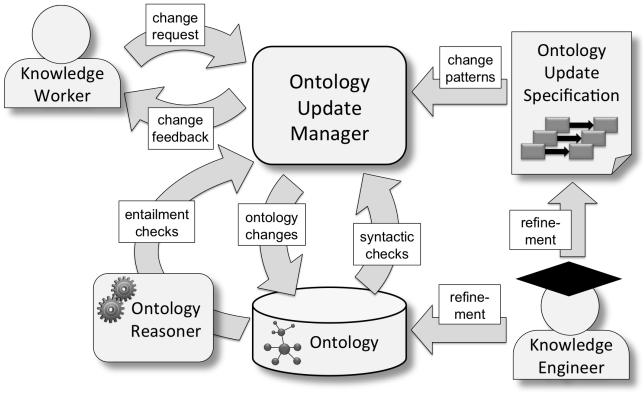

Fig. 1. Ontology Update Architecture

reactions to a change request should be an essential part in any practically employable ontology update framework.

In order to provide informative feedback, the ontology engineer has to provide template-like explanation snippets commenting on the nature of the change patterns contained in the update specification and the changes triggered by them. At runtime, those templates instantiated with the actually changed ontology elements can be presented to the knowledge worker in order to explain what actually happened to the ontology.

3 System Architecture

In this section, we propose an abstract architecture for an ontology update framework taking into account the design choices made in the previous section. A concrete instantiation of this architecture is described in the subsequent sections. The suggested architecture is sketched in Fig. 1. Therein, the usual unguided interaction mode of committing changes directly to the ontology is complemented by an additional update management component as editing interface for the knowledge worker. Still, the knowledge engineer will be able to directly change both the ontology and the ontology update specification.

The typical work flow of an ontology update step is carried out as follows: Initially, the knowledge worker issues a change request by providing a piece of knowledge to be added to or deleted from the ontology.

A change request will only be acted upon if the accompanying Ontology Update Specification accounts for it. The Ontology Update Manager will scan the Update Specification for an update rule whose applicability conditions are satisfied by the uttered change request and the ontology. Those applicability conditions might contain syntactic as well as semantic checks. If there are several applicable update rules, only one of them is applied.

If an applicable update rule has been determined, the change request can be acted on accordingly by denying or accepting it but possibly also by carrying out more ontology changes than explicitly requested.

Finally, a feedback message describing the activated change pattern and containing the actually performed changes is generated and sent back to the knowledge worker.

In case no applicable update specification was found, the change request is denied. It is however logged such that the ontology engineer can take care of it later and also refine the Ontology Update Specifications if needed.

The proposed framework is inspired by database triggers as e.g. described in the SQL standard [4]. Database triggers are stored procedures that are activated by changes that are submitted to the database. They may be defined for insertions, updates and deletions of instance data in the database. Our approach provides the same kind of functionality for instance data in ontologies while additionally allowing for defining update specifications for schema changes, which is usually not possible with database triggers.

4 A Language Proposal

In this section, we instantiate our previous general considerations by providing an ontology update framework based on RDF(S) and SPARQL, as well as SPARQL Update. This framework consists of the syntax of the Ontology Update Language (OUL, specified in Fig. 2) together with the precise description how ontology change requests are to be handled by the ontology management component (cf. Alg. 1).

Every update rule (also called changehandler) has an identifier. It carries a change request pattern, expressing for which change request it can be applied and some preconditions that define whether a change request can be handled by this rule depending on the current ontology state. If several matching changehandlers exist, the first one occurring in the update specification will be applied.[5]

A change request pattern is defined by the `WHERE`-clause of a SPARQL select query that is to be evaluated on the change request. That means that the statements submitted for a change are interpreted as an RDF graph and it is checked whether the change request pattern executed as SPARQL query yields any (or, if the `unique` option is set: exactly one) result on this change graph (lines 4 – 6 in the Algorithm) The result of this queries are bindings of all variables that are present in the `WHERE`-clause. Those bound variables can be reused later in the preconditions and actions part.

If a match is found, the precondition of the changehandler is evaluated. This determines whether the changehandler is applicable or whether another matching one has to be found. There are three basic types of preconditions: a syntactic

[5] We are aware that we thereby deviate from pure declarativity. However, for a first proposal, this kind of implicit priority declaration seems both intuitive and computationally feasible.

```
CREATE CHANGEHANDLER <name>
FOR <changerequest>
AS
  [ IF <precondition>
  THEN ] <actions>
<changerequest>  ::== add    [unique] (<SPARQL>)
                  |   delete [unique] (<SPARQL>)
<precondition>   ::== contains(<SPARQL>)
                  | entails(<SPARQL>)
                  | entailsChanged(<SPARQL>)
                  | (<precondition>)
                  | <precondition> and <precondition>
                  | <precondition> or <precondition>
<actions>        ::== [<action>]|<action><actions>
<action>         ::== <SPARQL update>
                  | for( <precondition> ) <actions> end;
                  | feedback(<text>)
                  | applyRequest
<SPARQL>         ::== where clause of a SPARQL query
<SPARQL update>  ::== a modify action (in SPARQL Update)
<text>           ::== string (may contain SPARQL variables)
```

Fig. 2. Ontology Update Language syntax specification in BNF.

check (that simply verifies whether certain triples are contained in the ontology) is performed via `contains`. Semantic entailment checks may be performed on the ontology in its current, unaltered state (`entails`) or on the "hypothetical" ontology that would result from carrying out the changes as requested (`entailsChanged`). As explained above, it is desirable to provide syntactic and semantic checks on the ontology, as syntactic checks are less expensive but also less accurate than semantic checks. Syntactically, basic preconditions are also the `WHERE`-part of a SPARQL query.

Basic preconditions can be combined by `and` and `or`. As it is reasonable to allow for (yet unbound) variables to occur in several basic preconditions, `and` and `or` are realized as join and union on the result sets of the basic preconditions. In the end, a precondition is considered to be successful (line 8) if its result set contains at least one entry. Before evaluation of the precondition, all variables that occurred before in the change request pattern are substituted by their binding (line 7, in the case of several existing bindings, the first one is chosen).

If the precondition is evaluated successfully, the actions specified in the changehandler's body are applied to the ontology.[6]

As for the evaluation of preconditions, all variables occurring in the action part of the changehandler that were bound before (i.e. that were present in either

[6] As it is possible to specify a changehandler which matches any request, it is up to the knowledge worker what should be done with change requests for which no matching changehandler is found.

Algorithm 1: Processing of Change Request

Input: ontology $\mathcal{O}$ consisting of axioms (RDF triples – Note that in RDFS every axiom is represented by exactly one triple),
ontology update specification $\mathcal{US}$ treated as list of changehandlers,
change request $op(Ax)$ where $op \in \{\texttt{add}, \texttt{del}\}$ and Ax is a set of axioms resp. triples.
Data: *candidate* changehandler that is checked for applicability
toExecute container to store the activated changehandler
updateList list of SPARQL Updates to be carried out, initially empty
Result: Updated ontology $\mathcal{O}$

```
1  //find an appropriate changehandler
2  while toExecute.isEmpty and not US.endOfDocument do
3  |  candidate ← US.nextChangeHandler
4  |  matches ← SPARQLmatch(candidate.changerequest, op(Ax))
5  |  if not matches.isEmpty then
6  |  |  if matches.count == 1 or not candidate.changerequest.unique then
7  |  |  |  instPrecondition ←
   |  |  |  Substitute(candidate.precondition, matches.first)
8  |  |  |  if not evaluate(instPrecondition, O).isEmpty then
9  |  |  |  |  toExecute.add(candidate)
10 |  |  |  end
11 |  |  end
12 |  end
13 end
14 //execute actions, if applicable
15 if not toExecute.isEmpty then
16 |  todo ← Substitute(toExecute.first.actions, matches.first)
17 |  cumulateActions(O, todo, updateList)
18 |  foreach update ∈ updateList do
19 |  |  apply update to O
20 |  end
21 end
22 return O
```

the change request or in the precondition) are replaced by their binding before the action part is applied (line 16). If several possible bindings were found for the change request or the precondition, the first binding is chosen.

Elementary actions that can be carried out are knowledge base changes (expressed as SPARQL Update operations), the `applyRequest` action (carrying out the originally uttered change request), and feedback messages. Moreover, elementary actions can be nested into loops which iterate over the result set of a precondition. While executing the action part of a changehandler, the activated ontology changes are not directly applied but first assembled in a list (line 17) and applied thereafter. This way all the loop preconditions are evaluated against the original ontology (or, in the case of `entailsChanged`, against the ontology that has been altered in the initially proposed way), thereby preserving a declar-

ative flavor. The application of the changes is done in an atomic manner after assembling is complete.

As it would not be easy to ensure termination or avoid high computational cost if the *actions* part of a change request was allowed to trigger other change requests, no other changehandlers are triggered during the execution of a changehandler. While this ensures termination, it makes the ontology engineer responsible for "manually" handling all additional changes that might become necessary due to the changes during the execution of the changehandler.

As a preliminary solution, the association of changehandlers with an ontology works similar to the association of DTDs with a XML document [2]. They can either be defined inline in the document specifying the data or they can be defined in external files. In either case, the changehandler is defined in a comment (such that the rdf file can also be parsed by ontology management systems that do not support OUL). All comments that have an extra '-' at the beginning are parsed as changehandler definitions. This begin of the comment may be followed by a file name enclosed in quotation marks, which defines an external changehandler, or by the definition of a changehandler enclosed in square brackets, defining the changehandler inline.

5 Examples

In this section, we provide a small example aimed at both advocating the potential usefulness of our proposed update framework and demonstrating the concrete work flow. We start with the knowledge base from Fig. 3. This RDFS specification, where we use Turtle syntax for better readability, contains the knowledge of the current status of our domain. Furthermore suppose this ontology is accompanied by the ontology update specification shown in Fig. 4. The first changehandler therein deals with the case that somebody leaves his/her current affiliation. In that case, the deletion of the affiliation information has to trigger further changes: the person will not continue to lead projects at the

```
philipp     rdf:type           PhDStudent .
philipp     hasAffiliation     aifb .
philipp     leads              xmedia .
philipp     worksOn            multipla .
philipp     supervises         thanh .
thanh       rdf:type           PhDStudent .
thanh       hasAffiliation     aifb .
thanh       worksOn            xmedia .
xmedia      rdf:type           Project .
xmedia      assocInstitution   aifb .
multipla    rdf:type           Project .
multipla    assocInstitution   aifb .
```

Fig. 3. Example knowledge base.

```
CREATE CHANGEHANDLER leavesInstitution
   FOR del { ?x  hasAffiliation  ?y }
AS applyRequest;
   feedback("?x is no longer affiliated to ?y");
   for(contains(?x ?wol ?z . ?z rdf:type Project .
                ?z  assocInstitution ?y .
                FILTER(?wol=worksOn || ?wol=leads)))
      delete data {?x ?wol ?z};
      feedback("Thus, ?x does not lead/work on project ?z anymore."); end;
   for(contains(?x  supervises ?z .  ?z  hasAffiliation ?y))
      delete data {?x supervises ?z};
      feedback("Thus, ?x does not supervise ?z anymore"); end;

CREATE CHANGEHANDLER authorsPhD
   FOR add { ?x  swrc:authorOf  ?y }
AS IF entailschanged( ?y  rdf:type swrc:PhDThesis . )
   THEN applyRequest;
        delete data { ?x  rdf:type  swrc:PhDStudent};
        feedback("Change accepted. ?x authored a PhDThesis,
                  so he is no PhD student anymore.");
```

Fig. 4. Example ontology update specification.

institution he/she leaves nor to supervise persons. Now suppose the following change request, indicating that Philipp is leaving the AIFB institute, is entered into the system:

```
delete data {philipp hasAffiliation aifb .}
```

The system will now check whether this change request matches the first changehandler's change pattern `del { ?x hasAffiliation ?y }`. This is the case, as the corresponding SPARQL query yields a result which binds `philipp` to `?x` and `aifb` to `?y`. The considered changehandler is now executed as it does not contain further preconditions for activation. So, the specified actions will be carried out: `applyRequest` means that the initial change request is granted and added to the list of updates to be executed. After that the following message is displayed:

```
philipp is no longer affiliated to aifb.
```

Next, we consider the graph pattern in the first loop. Note that it contains variables that have already been bound by the change pattern matching. Before evaluating the loop action, those variables are substituted by their bindings, in our case resulting in the following rewritten loop action:

```
for(contains(philipp ?wol ?z . ?z rdf:type Project .
             ?z  assocInstitution aifb .
             FILTER(?wol=worksOn || ?wol=leads)))
   delete data {philipp ?wol ?z};
   feedback("philipp does not lead/work on project ?z anymore"); end;
```

Now, the conditional part of the rewritten loop action is matched against the database, yielding the following two variable bindings: `?wol`↦`leads`, `?z` ↦`xmedia` and `?wol`↦`worksOn`, `?z` ↦`multipla`. Next, for each of these two bindings, the subsequent actions are executed: therefore, the triples `philipp leads xmedia.` and `philipp worksOn multipla.` are scheduled for deletion and the following two messages are prompted to the user:

```
Thus, philipp does not lead/work on project xmedia anymore.
Thus, philipp does not lead/work on project multipla anymore.
```

In analogy to that, by executing the second loop of the activated changehandler, `philipp supervises thanh.` is scheduled for deletion and the message

```
Thus, philipp does not supervise thanh anymore.
```

prompted to the user. Finally, all the scheduled changes are carried out.

Finally, we will present some standard situations or decisions which might occur in an ontology update setting and how they can be realized by means of the Ontology Update Language as presented in this paper.

Restricting the Size of the Change. Clearly, it is not always desirable to permit change requests of arbitrary size. In principle, an entire ontology could be added in one step, which would be a situation hard to handle with update rules. One solution to this is to restrict the size of the change a priori. As an extreme case of this, only one RDF triple per change might be allowed. While this constraint can be imposed by external means, our formalism is flexible enough to handle it. In order to allow only add changes consisting of one triple, the following changehandler would have to be inserted at the beginning of an ontology update specification:

```
CREATE CHANGEHANDLER tooMuchForOneBite
   FOR add ( { ?a ?b ?c . ?d ?e ?f .
       !(sameTERM(?a,?d) && sameTERM(?b,?e) && sameTERM(?c,?f))} )
AS feedback("Request denied. Only one triple per change!");
```

In words, the change request pattern checks whether there are two distinct triples contained in the change request. If so, the changehandler is activated without doing any changes (thereby effectively rejecting the request). Note that the implemented selection strategy also prevents any subsequent changehandler in the document from being activated.

Handling of Change Requests. Of course, change requests might occur which do not activate any of the specified changehandlers. In the presented implementation, the request will be tacitly denied in this case. It is however possible to create changehandlers that match any `add` (resp. `delete`) request. Placed at the bottom of an ontology update specification, those can be used to provide feedback whenever no other changehandler was activated:

```
CREATE CHANGEHANDLER noMatchRestrictive FOR add ( { ?a ?b ?c . } )
AS feedback("Request denied. No matching change rule found!");
```

This is just an explication of the default restrictive strategy: every unforeseen request will be denied. In the same way, it is of course possible to realize a permissive strategy by stating

```
CREATE CHANGEHANDLER noMatchPermissive FOR add ( { ?a ?b ?c . } )
AS applyrequest;
   feedback("Request accepted. No matching change rule found.");
```

instead. This way, all requests not matching any of the preceding changehandlers will be complied with.

6 Implementation

We provide an implementation of our architecture and our language proposal at `http://www.aifb.uni-karlsruhe.de/WBS/uhe/OUL/`. The implementation uses Jena as underlying framework for ontology management. This framework was chosen, as Jena provides an implementation of SPARQL update [11].

SPARQL update is an extension of the ontology query language SPARQL [9]. While SPARQL's purpose is to find data in a RDF graph, SPARQL update provides functionality for updating and managing RDF graphs using a SPARQL-like syntax. Update operations allow changing existing graphs by adding data (`insert`), deleting data (`delete`) or both (`modify`). Besides directly inserting/deleting a set of triples, these operations allow to change an ontology based on a pattern. A pattern is a description of a RDF graph, which may contain variables that can be used to describe elements of the graph and can then be determined as result. These results are used in SPARQL update queries to determine which triples should be added resp. deleted from the ontology.

Our implementation provides a wrapper for Jena's SPARQL update endpoint, which implements the ontology update management as we proposed it. SPARQL update requests can be submitted as in the original implementation, but instead of directly executing them, the graph of changes that will have to be applied is constructed and a suitable change handler is searched for as explained in Section 4 and the respective actions are performed. By default, if no change handler is found, the ontology remains unchanged. This approach makes the update management as transparent as possible for the user. The only difference with respect to using the original implementation is using another endpoint and getting all the described advantages. This allows to open SPARQL endpoints for writing access more liberally.

7 Conclusions and Future Work

In this paper, we have addressed the task of updating an ontology due to changes in the described domain. We have argued for a formalism that allows for specifying the domain-dependent ways in which a specific ontology may evolve over time. We thoroughly discussed the crucial design decisions to be made for an

ontology update framework that would automatically align change requests with change patterns and thereby allow to delegate simple ontology maintenance tasks to users not necessarily possessing the expertise of a knowledge engineer. We presented and implemented a proposal for such a framework.

Being aware that our proposal constitutes just a first step towards a much needed functionality for semantic technologies, we identify several directions for future research:

Extending the implementation to OWL. Currently, our implementation works with RDF(S) and SPARQL. Extending it to OWL would require to extend SPARQL accordingly, and to allow Algorithm 1 to use multiple-triple axioms as they often occur in OWL DL knowledge bases.

Interactivity. It might be the case that a change request cannot be unequivocally assigned to one single change pattern. As an example, the change request to retract the fact that an individual is a PhD student might be due to the fact that he got his PhD degree or he dropped out; either of the variants requiring different further ontology changes. While this alternative could be easily accommodated by extending the update language, the ultimate choice which change pattern to apply in a concrete case should be left to the person uttering the change request. Therefore, also additional interactive control loops would have to be introduced into the framework.

Combination with belief revision. Although we have argued that the rationale of belief revision does not fit well with our purpose, there are certainly cases where a combination of both is beneficial. Belief revision could be used as a fall-back strategy if a change request would lead to an inconsistent ontology and is not tackled by any of the update specification's change patterns. Instead of simply rejecting the change, belief revision techniques could be more appropriate. In general, belief revision and other coping strategies could be incorporated into the proposed formalism in a plugin-manner as additional actions next to adding and deleting axioms.

Higher order constructs. Our proposal of an ontology update has a rather operational flavor. While this arguably facilitates the employment and allows for an efficient and straightforward implementation, a more declarative way of describing the possible domain changes would be more in the spirit of the current ontology languages. Moreover a specification in OWL would abide by the rationale to reuse formalisms (just as the XML syntax is also used for XML Schema). Hence it seems sensible to introduce a more abstract description layer for complex changes, preferably in OWL. The underlying model for such a declarative framework could be inspired by the usual ways of describing discrete dynamic systems such as finite automata or petri nets. A simple example would be to relate the two classes Child and Adult with each other with a property allowing the transformation of instances of the one class to an instance of the other, e.g `Child disjointTransformationTo Adult`. Note that in OWL2 such a property is legal due to punning.

Learning Change Patterns. Clearly, the success of the proposed framework depends on the quality of the update specification. While in certain domains

the development of such a specification might be straight forward (possibly because there are already informal documents describing the standard processes and work flows) there might be scenarios where this is not the case. Under those circumstances, frequent change patterns could be extracted from ontology change logs by some machine learning techniques. Those findings could then be presented to the knowledge engineer as suggestions for ontology update rules to be incorporated into the specification.

An elaborate ontology update mechanism as presented here allows ontologies to be updated in a more predictable and quality preserving way. Administrators of ontology based system may choose to allow a wider audience to edit their ontologies in a controlled manner, thus extending the collaborative aspect of ontology maintenance.

References

1. F. Baader et al., editor. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2007.
2. T. Bray et al. Extensible markup language (xml) 1.0 (fifth edition). W3C Recommendation, 26 Nov. 2008. Available at `http://www.w3.org/TR/REC-xml/`.
3. E. Carle. *The Very Hungry Caterpillar*. Philomel Books, 1969.
4. C. J. Date and H. Darwen. ISO/IEC 9075-2:2008 (SQL - Part 2: Foundations) , "The SQL Standard", Third Edition (Addison-Wesley Publishing Company, 1993).
5. H. Katsuno and A. O. Mendelzon. On the difference between updating a knowledge base and revising it. In P. Gärdenfors, editor, *Belief Revision*, pages 183–203. Cambridge University Press, December 1992.
6. G. Konstantinidis et al. A formal approach for RDF/S ontology evolution. In M. Ghallab et al., editor, *Proc. ECAI'2008*, pages 70–74. IOS Press, JUL 2008.
7. F. Manola and E. Milner. RDF Primer. W3C Recommendation, 10 Feb. 2004. Available at `http://www.w3.org/TR/REC-rdf-syntax/`.
8. D. McGuinness and F. v. Harmelen. OWL Web Ontology Language Overview. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/owl-features/`.
9. E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. W3C Recommendation, 15 Jan. 2008. Available at `http://www.w3.org/TR/rdf-sparql-query/`.
10. G. Qi and F. Yang. A survey of revision approaches in description logics. In D. Calvanese and G. Lausen, editors, *Proc. RR2008*, volume 5341 of *LNCS*, pages 74–88. Springer, 2008.
11. A. Seaborne et al. SPARQL Update. W3C Member Submission, 15 Jul. 2008. Available at `http://www.w3.org/Submission/2008/SUBM-SPARQL-Update-20080715/`.
12. S. Staab and R. Studer, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, 2004.
13. L. Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, Universität Karlsruhe (TH), 2004.
14. C. Tempich et al. Argumentation-based ontology engineering. *IEEE Intelligent Systems*, 22(6):52–59, 2007.
15. T. Tudorache et al. Supporting collaborative ontology development in protégé. In *International Semantic Web Conference*, pages 17–32, 2008.