Version Management in Semantic Web Services Using OWL-S*

Maria Cecilia Bastarrica, Carlos Hurtado, and Alejandro Vaisman

Department of Computer Science, Universidad de Chile
{cecilia, churtado, avaisman}@dcc.uchile.cl

Abstract. In the last few years there has been an increasing interest in studying ontology evolution and versioning for the World Wide Web, in particular, applied to OWL. However, little attention has been given to the problem of Web services evolution, with a focus on OWL-S, an ontology of services recently proposed. In this paper, we show that recent work on Temporal RDF can be extended to support versioning of an ontology of services. We introduce a formal model and a query language that allow accessing different versions of an OWL-S specification. We present the language semantics and discuss complexity issues. We show how our proposal can be implemented within the OWL-S framework.

1 Introduction

OWL [19] is an ontology language for the Semantic Web, developed by the World Wide Web Consortium (W3C). It allows the representation of information about categories of objects, and how these objects interrelate. This information, in a Semantic Web scenario, can help to develop efficient automated processes in order to access information on the Web. OWL is built on top of the Resource Description Framework (RDF) [1,11], and extends RDF and RDFS, adding restrictions on properties, and operations like disjunction and negation.

Web services are software applications that interact using Web standards. Although Web service technology is rapidly gaining popularity, it still requires more human involvement than may be wanted. Avoiding this would imply the ability of automatically discovering and invoking Web services. Semantic Web technology has helped to solve this problem by means of *ontologies of services* that are used for representing a *service profile* (for describing services offered by a Web site). These ontologies can be used by service-seeking agents. The efforts for defining a standard for ontologies of services led to OWL-S [17], a language that allows to describe what a service provides, what a service requires from the users, how the service works, and how the service is used. OWL-S is aimed at enabling efficient automatic Web service discovery, invocation, interoperation, and execution monitoring.

* Partially supported by Millennium Nucleus Center for Web Research, Grant P04-67-F, Mideplan, Chile, and Project Fondecyt No. 1050642.

1.1 Motivation

Today's business systems must be able to adapt to changes and so does the Semantic Web. Most of the change management tasks are still being performed manually [20], which is time-consuming and error prone. It would be desirable to add change management capabilities to the Semantic Web. An example of an evolving ontology is *MeSH*, a medical ontology used by MEDLINE, a huge source of medical information on the Web. *MeSH* is frequently updated in order to stay in line with the state-of-the-art in medical research. The changes that *MeSH* goes through consist of the addition of new terms, and also reclassification of such terms. It seems clear that there is need for ontology evolution support on the Web, as it has been pointed out in [5,10,13,14], among other works.

Another real-life example that illustrates the need for Web services versioning, is the area of mobile phones. Phone companies need to incorporate new services continuously in order to keep competitive in the market. To achieve the required flexibility and evolvability many of them are currently migrating their platforms towards service oriented architectures [4] where each service is implemented as a Web service. Each Web service may then provide a particular service with as many diverse operations as different cell phone platforms need to access the service. For example, the monotone ringtone service is different depending on the phone model. As the cell phone technology advances rapidly, new phone devices appear continuously and there is a need to provide support for them. Also, it may be necessary to keep the service version history in order to provide the service if holders of older phone models require it. Another typical use of this version history keeping is to determine the time interval in which the cell phone company supported the service for a certain phone model so that it can charge the phone vendor for providing support for its devices.

There are many ways to address ontology changes. Stojanovic [20] classifies ontology changes in four categories: (a) ontology management; (b) ontology modification; (c) ontology evolution, and (d) ontology versioning. There is limited work on ontology evolution, but little has been done on ontology versioning [13]. In this paper we address a topic still more unexplored: versioning of Web services ontologies. Versioning has been recognized as a relevant problem in Web service development [2,3]. However, no formal study has been attempted. Developers devise ad-hoc solutions when they need to deal with the problem of maintaining different versions of their applications. A proper mechanism for version management would allow easy access to different versions of a Web service ontology as of any point in time. Moreover, it would add flexibility, allowing to access simultaneously multiple versions of a service [3].

1.2 Problem Statement and Contributions

We address the problem of Web service versioning and we show that OWL-S, as a language for describing Web service ontologies, can be extended in order to support versioning. In this way, we are able to query and use past states of a service ontology. For this purpose, we use the approach in [6,7] for RDF.

A proper version management mechanism requires a temporal data model to support it. Thus, this paper proposes a temporal data model supporting versioning of Web services. We define an abstract model for OWL-S, and extend its components with temporal labels that indicate their intervals of validity, leading to a temporal OWL-S graph. This temporal model, denoted OWL-S(T), allows to manage versioning at two levels: version and state (i.e., within a version different states can be supported). We then define a notion of OWL-S(T) consistency, based on the framework proposed in [9], and we show how consistency can be checked in a temporal setting, ensuring that a document is consistent at any point in time. We propose a query language that supports typical temporal queries, and allows to retrieve versions of an OWL-S(T) document as of any instant in time. We give the semantics an complexity of the language. Finally, we sketch how our proposal can be implemented in the OWL-S framework.

Section 2 discusses related work. Section 3 gives an overview of temporal concepts and the OWL-S notions used in the paper; we also introduce the abstract model for OWL-S. Section 4 presents the model for introducing time in OWL-S and Section 5 a proposal for a query language. Section 6 discusses how the introduced concepts could be implemented within the OWL-S framework. We conclude in Section 7.

2 Related Work

The Ontology Web Language (OWL) [19] was developed by the W3C Web Ontology Working Group. Many of its features come from its predecessor DAML+OIL, and from the fields of Description Logic and Knowledge Representation. Horrocks *et al* [12] detail the evolution of OWL. OWL is built on top of RDF [16], a metadata language for the Semantic Web. Several languages for querying ontologies have been proposed and implemented, some of them in the lines of traditional database query languages, others based on logic and rule languages [8,15].

Although temporal management has been recently studied for semistructured, XML, and RDF data, little has been done to this respect in the Semantic Web setting. Among the four categories in which Stojanovic [20] classified ontology changes, we are particularly interested in *evolution* and *versioning.* Stojanovic addressed the first problem, and defined the requirements for an efficient ontology evolution system. This approach was implemented in the so-called KAON framework. The problem of preserving consistency upon an evolving ontology was studied by Haase *et al* [9]. This work provides a comprehensive overview of the state-of-the-art in ontology evolution [10]. Finally, Flouris *et al* [5] claim that the current approaches for ontology evolution lack formality, and propose a model that generalizes and applies the AGM postulates.

In the field of ontology versioning, Klein *et al* [13] present a system, called Onto View, that helps specifying relations between different versions of an ontology (but does not keep track of the history). Visser *et al* [22] propose a temporal reasoning framework for the Semantic Web, applied in BUSTER, an ontology-based prototype supporting the so-called *concept@location in time* type of query.

Huang *et al* propose a reasoning framework for ontology versioning, based on temporal logic; they claim that ontology evolution is well-understood, although ontology integrity is still an open research field. Their multi-version reasoning framework is aimed at discovering inconsistencies caused by ontology evolution. It is defined as an extension of linear temporal logic (LTL), denoted LTLm.

There is a clear need for version management in the design of Web services [2,3]. Brown *et al* [2] classified changes in Web services in two broad classes: *backward-compatible* and *non-backward-compatible.* In the former, we have the addition of new WSDL operations and new XML schema types. In the latter, they include: removing/renaming operations, changing the parameters of an operation, and changing the structure of a complex data type. In spite of this there is still no study of temporality issues in OWL-S ontologies and query languages.

3 Preliminaries

3.1 Temporal Issues

The existing approaches to ontology versioning are based on developing a new physical version of the ontology each time a change occurs. In [6], they propose a different approach for the evolution of RDF specifications, which can be seen as a logical theory. They timestamp the RDF triples with their interval of validity. In order to introduce the time dimension into OWL-S, we are faced with the same question: should we maintain a snapshot of each state of the graph or, should we label the elements of the OWL-S specification that are subject to changes? Although both models are equivalent, the first one appears to be not suitable for queries of the form: "all time instants where some condition Φ holds in the specification". It is well-known [21] that there are at least two temporal dimensions to consider: *valid* and *transaction* times. *Valid* time is the time when data is valid in the modeled world; *transaction* time is the time when data is actually stored in the database. The snapshot approach captures transaction time, while labeling is mostly used when representing valid time. The approach we present in this paper can support both time dimensions.

For evolving OWL-S specifications, labeling may do better in scenarios where changes are frequent and only affecting a few elements. In this situation, creating a new *physical* version of the graph each time an update occurs may lead to large overheads when processing temporal queries that span multiple versions. Thus, labeling will be our approach. We work with the point-based temporal domain for defining our data model and query language, but we encode time-points in intervals whenever possible, for the sake of clarity. We consider time as a discrete, linearly ordered domain, as usual in virtually all temporal database applications. As usual in temporal databases, the (moving) current instant is denoted *Now.*

3.2 OWL-S Overview

Software agents that access Web services need a description of the available services to perform an efficient service lookup. This description is provided by OWL-S [17]. At a high level of abstraction, OWL-S can be seen as an ontology structure

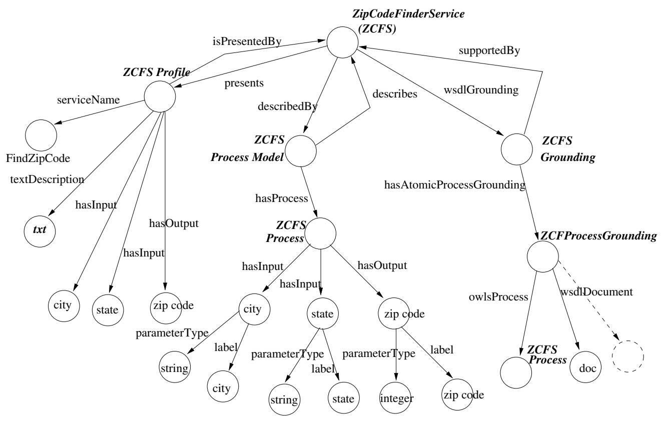

Fig. 1. Running example

whose instances are the OWL-S specifications. This structure or schema is composed of three classes: *ServiceProfile*, *ServiceModel* and *ServiceGrounding*. These classes are related to the *Service* class by the properties *presents*, *describedby*, and *supports*, respectively. These properties have also their inverse. For simplicity we focus on the properties `serviceName, textDescription, has parameter, has input, has output` for the class *Profile*; from the *Process* class (a subclass of *ServiceModel*), we use the `has input, has output, parameterType` and `label` properties; from *ServiceGrounding* we use the `wsdlDocument` property, which is the URI of the WSDL operation corresponding to an atomic process. We have chosen these properties because they give a good intuition of the problem, and their instances are likely to change over time.

Throughout the paper we use the service ontology depicted in Figure 1, adapted from [18]. This is an abstract representation of an OWL-S specification for a Web service that receives a pair city-state, and returns the corresponding zip code (for brevity, we only partially show the *Grounding* part). The service *Profile* tells what the service does, the *Process Model* tells the service clients how to use the service, and the *Grounding* specifies how an agent can access the service. WSDL operations bind the ontology to the implementation of the Web service. In the example of Figure 1, we have included the types and names of the service input and output parameters. The following fragment corresponds to the abstract graph in Figure 1.

```
  ...
<grounding:WsdlAtomicProcessGrounding rdf:ID=‘‘ZCFProcessGrounding’’>
 <grounding:owlProcess rdf:resource=‘‘#ZipCodeFinderProcess’’/>
 <grounding:wsdlDocument>
 ‘‘http://www.dcc.uchile.cl/2005/ws/docum/ZipCode-v1.0.asmx?WDSL’’
 </grounding:wsdlDocument>
 .... >
```

3.3 Abstract Model of OWL-S

In what follows, we adopt an abstract model for an instance of an OWL-S specification and we describe it as a graph (from now on, an *OWL-S graph*). We are not interested in the OWL-S structure, given that we will consider this structure *static*. Thus, we do not need our model to represent relations like *subpropertyOf*, *subclassOf*, and so on. Figure 2 (a) shows an abstract model for an OWL-S instance. The nodes represent resources (domain and range of each property). The edges represent OWL-S properties, which include properties provided by the OWL-S ontology such as `presents`, `serviceName`, `hasInput`, and `hasOutput`. Although we have denoted the properties p_i with different names, this may not always be the case. Edges denoted $inv(p_i)$ represent the inverse property of p_i, like in the case of *describes* and *describedBy* in Figure 1. Note that this graph is analogous to an RDF graph.

Definition 1 (OWL-S graph). *An OWL-S graph is a set of RDF triples* (a, p, b)*, where* p *is an OWL-S property.*

We next incorporate OWL-S constraints and consistency in our model. We denote Σ the set of OWL constraints given in the OWL-S specification. As an example, we have in Σ a constraint that states that all property p is equivalent to $inv(p)$, and a constraint that `isDescribedBy` has max cardinality 1.

Definition 2 (OWL-S Consistency). *An OWL-S graph is consistent if an only if it satisfies the set of OWL-S constraints* Σ*.*

4 Introducing Time into OWL-S

As mentioned in Section 3, we consider the schema (i.e., OWL-S ontology) as fixed. Thus, the instances of the ontology are the only elements subject to change. We extend the graph in Definition 1 with temporal labels, yielding a *temporal OWL-S graph*, and we state consistency conditions these graph must satisfy.

We assume the existence of three finite sets: intervals $\mathcal{I}$, timestamps T, and versions $\mathcal{V}$, and two functions: $\texttt{init} : \mathcal{I} \rightarrow T$ and $\texttt{end} : \mathcal{I} \rightarrow T$, which return the starting and ending timestamps of an interval, respectively.

Definition 3 (Temporal OWL-S graph). *A temporal OWL-S graph* H *is a tuple* (G, V, ρ)*, where* G *is an OWL-S graph whose triples are annotated with intervals in* $\mathcal{I}$*,* $V \subseteq \mathcal{V}$ *is a set of versions, and* $\rho : V \rightarrow \mathcal{I}$ *is a function that assigns to each version an interval (lifespan of the version). The intervals in* $ran(\rho)$ *do not appear in* G*.*

Note that the model only supports single intervals. This introduces some limitations on the model's expressive power. However, the model can be extended to support sets of intervals. Figure 2 (b) shows a temporal OWL-S graph.

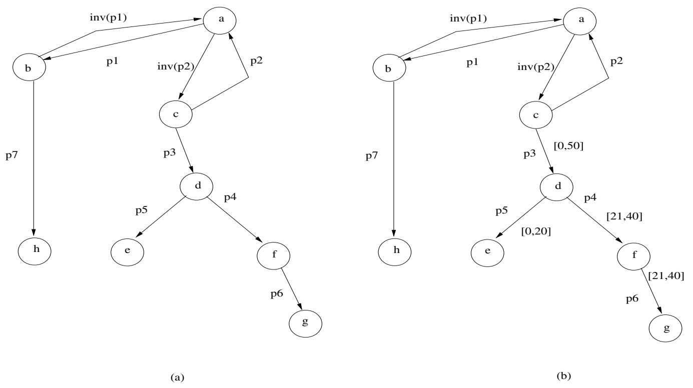

Fig. 2. (a) An abstract OWL-S graph. (b)The corresponding *t-OWL-S graph.*

Definition 4 (Lifespan of a node). *Given a temporal graph H, the lifespan of a node n of it, denoted $lifespan(n)$ is the interval i such that $\texttt{init}(i) = Min(\{\texttt{init}(i') : (a,p,n)[i'] \in G\} \cup \{\texttt{init}(i') : (n,p,b)[i'] \in G\})$ and $\texttt{end}(i) = Max(\{\texttt{end}(i') : (a,p,n)[i'] \in G\} \cup \{\texttt{end}(i')(n,p,b)[i'] \in G\})$.*

Definition 5 (Snapshot). *Given a temporal OWL-S graph G, a* snapshot *of G at time t, denoted $G(t)$, is an OWL-S graph, with triples (a,p,b) such that $(a,p,b)[t]$ is in G.*

Definition 5 provides the link between OWL-S with versioning, and temporal OWL-S. A specification has as many *versions* as different snapshots can be obtained at different instants. Thus, all versions of an OWL-S specification are embedded in a single document.

4.1 Updates

We consider the following subset of the changes proposed in [2]: (a) add a new WSDL operation; (b) remove an WSDL operation; (c) change the parameters of an operation. However, note that the abstract model defined above hides most of the non-temporal semantics of OWL-S. In this way, the update operations can be implemented as operations over the graph, as follows: (1) adding a new operation at time t is the insertion of a new edge (and the corresponding node), with temporal label $[t, Now]$; (2) removing (at time t) an existing WSDL operation (indicated by and edge with label $[t_s, Now]$) is implemented replacing the label by $[t_s, t]$ (thus, the history is retained); (3) changing some parameter implies the following operations over the graph: (a) remove, at time $t-1$, the edge to be modified; (b) add an edge with the new parameter, at time t. For example, suppose we want to update property p_7, at time $t = 100$, replacing the range URI h, by r, in the graph in Figure 2. The edge $(b, p_7, h)[0, Now]$ is replaced by $(b, p_7, h)[0, 99]$, and a new edge $(b, p_7, g)[100, Now]$ is inserted.

This definition of updates shows that we can partition the time line (i.e. the temporal document's lifespan) in a set of intervals such that, within these intervals, all the snapshots remain the same.

Definition 6 (Interval Partition). *The interval partition P of a set of intervals $i_1, \ldots, i_n$, is the smallest set of intervals $\mathcal{P} = P_1, \ldots, P_n$, such that all the P_i's in $\mathcal{P}$ are pairwise disjoint and $\mathcal{P}$ contains a partition of every interval i_j.*

4.2 Consistency

Hasse *et al* [9] studied the problem of keeping an evolving ontology consistent through its different states. They defined three different notions of consistency, respectively denoted κ_S, κ_L, and κ_U: (a) structural; (b) logical; (c) user-defined. We are interested in extending OWL-S consistency for the temporal model. Therefore we show how the set of OWL-S constraints Σ, which captures structural and logical consistency, can be applied in the temporal setting.

Definition 7 (Consistency). *A temporal OWL-S graph G is consistent if and only if every snapshot $G(t)$ is consistent.*

Given a temporal OWL-S graph G and an interval i, we denote $G(i)$, the (non-temporal) OWL-S graph with triples (a, b, c) such that $(a, b, c) : i \in G$.

Theorem 1. *A temporal OWL-S graph G is consistent if and only if for every interval i in the partition of the intervals of G, $G(i)$ is consistent.*

Proof. (sketch) Follows from defining the partition of the sets of intervals in the graph. The document is consistent within the set of timestamps defined by each interval in the partition.

The theorem yields an algorithm to verify consistency, which consists of two steps: (i) compute the partition of the intervals of G; (ii) compute $G(i)$ for each interval i in the partition and check satisfiability of the constraints in $G(i)$. Step (i) can be done by building intervals from consecutive timestamps mentioned in the intervals of G. Satisfiability can be checked using any OWL reasoner.

5 Querying Temporal OWL-S

The temporal OWL-S graph can be fully queried using the notion of temporal RDF [6,7]. Here we sketch a query language that allows to query different versions of OWL-S(T) specifications, along with changes inside versions themselves.

In order to capture changes inside versions, we introduce the notion of a state. Therefore, our query language augment standard RDF querying with the notions of version and state. Intuitively, a state is a maximal interval inside a version for which the OWL-S specification does not change.

Definition 8 (State). *Given an OWL-S(T) graph $H = (G, V, \rho)$, the set S of states of H is the smallest interval partition of the intervals in $\rho(V)$ such that for each interval $i \in S$ and for every pair of timestamps t_1, t_2, where $\texttt{init}(i) \leq t_1, t_2 \leq \texttt{end}(i)$, we have $H(t_1) = H(t_2)$.*

5.1 Queries by Example

Both, states and versions have "init" and "end" timestamps. We use the functions `init` and `end` to refer respectively to them. As an example, $\texttt{init}(s_2) = \texttt{end}(s_1)$, says that s_2 is a successor of s_1, and by $\texttt{end}(v) = \texttt{end}(s)$ we restrict s to be the last state of version v. We also include standard arithmetic built-in predicates $(<, \leq, =, \geq, >)$ to compare timestamps. For instance, $\texttt{init}(s_2) < 2$ tells that s_2 started before timestamp 2.

Now, let us begin with a simple query: "Find the inputs of the Web service *ZipCodeFinderService* in the last state of version v1".

$$\begin{aligned}(\mathit{ZipCodeFinderService},\ \mathit{hasInput}, ?U)[?S] \leftarrow \\ &(?X, \mathit{hasInput}, U?)[?S][v1],\ \texttt{end}(?S) = \texttt{end}(v1) \\ &(\mathit{ZipCodeFinderService}, \mathit{presents}, ?X)[?S][v1].\end{aligned}$$

Annotations inside brackets in the queries represent variables that range over states and versions. The following query returns "the profiles and states of version $v1$ of Web service *ZipCodeFinderService*".

$$\begin{aligned}(\mathit{ZipCodeFinderService},\ \mathit{presents}, ?Z)[?S] \leftarrow \\ &(\mathit{ZipCodeFinderService}, \mathit{presents}, ?Z)[?S][v1].\end{aligned}$$

Now consider the query "Find the versions of Web services that output a Zip code". We express it as the following query:

$$(?V, \texttt{versionOf}, ?Y) \leftarrow (?Y, \mathit{hasOutput}, \mathit{zipcode})[?S][?V],\ (?S, \mathit{presents}, ?Y)[?S][?V].$$

5.2 Semantics

We consider the following disjoint sets of variables: a set V_r of RDF variables, a set V_v containing version variables, and a set V_s of state variables. Individual variables are denoted $?X$, $?Y$, $?Z$, etc.

A query is a tableau, which is a pair $(H, B \cup A)$, where H and B are graph patterns, and the set A has the usual arithmetic predicates over timestamps and applications of the functions `init` and `end`. A graph pattern is a set of expressions of the form $(a, b, c)[s][v]$, where (a, b, c) is an RDF triple where some elements may be variables in V_r, $s \in V_s$ is a state variable, and v may be a version variable in V_v or a constant for a version name.

We adopt a notion of *safe rule* similarly to Datalog to prevent operations on infinite predicates. A rule is *safe* if each of its variables appear as an argument in a non-built-in predicate of the body.

In order to give the semantics of a query we transform a temporal OWL-S graph $H = (G, V, \rho)$ into an RDF graph whose triples are annotated with versions and states. This annotated graph, denoted $\texttt{VS}(H)$, is a set containing triples $(a, b, c)[s][v]$, which establishes that (a, b, c) holds in a state s of H, and s arises within a version $v \in V$ (i.e., $\texttt{init}(s) \leq \texttt{init}(v)$ and $\texttt{end}(v) \leq \texttt{end}(s)$).

Given an interval i and a set of intervals S, we denote $\texttt{CoverSet}(i, S)$ the set containing the intervals $i' \in S$ such that $\texttt{init}(i') \leq \texttt{init}(i)$ and $\texttt{end}(i) \leq \texttt{end}(i')$. Then, the set $\texttt{VS}(H)$ is obtained as follows. Let S be the set of states of H, and let U be the set of intervals in G. For each interval $i \in S$, and version v that contains i, we annotate with $[i][v]$ all the triples in $\bigcup_{i' \in \texttt{CoverSet}(i,U)} H(i')$, and add them to $\texttt{VS}(H)$.

The semantics is similar to temporal RDF [6]. Given a query $(H, B \cup A)$ and a OWL-S(T) graph H, for each matching of the graph pattern B in $\texttt{VS}(H)$, pick the values of the variables for versions and states, and check if they satisfy the built-in predicates in A. If this is the case, construct a pre-answer, which is the graph resulting by substituting the values of the variables in the head. The answer of the query is the union of all pre-answers.

5.3 Complexity

We now show that the query language proposed does not increase the complexity of temporal RDF.

Lemma 1. *Given an OWL-S(T) graph $H = (G, V, \rho)$, for each state s of H, there are intervals i, i' in H such that $\texttt{init}(i) = \texttt{init}(s)$ and $\texttt{end}(i') = \texttt{end}(s)$.*

This lemma gives a simple procedure to compute states. We need to order all the timestamps that limit the intervals in H, and search for maximal intervals that have these timestamps as limits, within which the temporal OWL-S graph does not change. This procedure takes $O(N^2M)$, where N is the size of H, and M is the number of intervals in H. It also shows that $\texttt{VS}(H)$ has size in $O(NM)$.

To get the complexity of query processing, we consider the problem of testing emptiness of the query answer set in the following forms: (1) Query complexity version: for a fixed database D, given a query q, is $q(D)$ non-empty? (2) Data complexity version: for a fixed query q, given a database D, is $q(D)$ non-empty?

Theorem 2. *The evaluation problem is NP-complete for the query complexity version, and polynomial for the data complexity version.*

The proof is similar to temporal RDF and is based on the fact that the graph $\texttt{VS}(H)$ over which the search for matching is done is of polysize in H.

6 Temporal OWL-S Implementation

Now we sketch how the concepts introduced in the paper can be embedded in an actual OWL-S specification. For this, we propose two mechanisms: (1) slightly extend the OWL-S vocabulary, specifying a new profile; (2) timestamp the elements of the OWL-S specification.

Versioning Profiles. The first extension we need is a small OWL-S vocabulary in order to define a fourth component of a OWL-S(T) specification (along with profile, process model, and grounding), that we will call a *versioning profile*. The vocabulary of this new profile includes the classes `versioningprofile`, `version`, `interval`, `date`, and the properties `hasVersion`, `spans` (and its inverse `isSpannedBy`), `lifeSpan`, `init` and `end`. The constraints for the versioning profile are: (a) the domain of `lifeSpan` is `version`, and its range is `interval`; (b) the domain and range of `init` and `end` are `interval` and `date`, respectively; (c) the domain and range of `hasVersion` are `versioningprofile` and `version`, respectively; (d) the domain and range of `spans` are *Service* and `versioningprofile`, respectively;(b) the cardinality of the `lifeSpan` property is 1-1.

In our running example, for instance, we would have the following triples stating that $v1$ is a version of the OWL-S(T) specification and its lifespan lies within the interval i, whose limits are 1 and 2:

$(ZCFS, \texttt{isSpannedBy}, ZCFSVersioningProfile)$, $(ZCFSVersioningProfile, \texttt{hasVersion}, v_1)$, $(v1, \texttt{lifeSpan}, i)$, $(i\texttt{init}, 1)$, $(i, \texttt{end}, 3)$.

OWL-S Timestamping. Assume that from 2005 on, a new version of the process which implements the service of our running example was released. The corresponding timestamped OWL-S specification would look as follows:

```
<rdf:RDF xmlns:owl=‘‘http://www.w3c.org/2002/07/owl#’’?>
  xmlns:Time=‘‘http://www.dcc.uchile.cl/db/time’’
  ...
<grounding:WsdlAtomicProcessGrounding rdf:ID=‘‘ZCFProcessGrounding’’>
 <grounding:owlProcess Time:FROM=‘1999-01-01’ Time:TO=‘Now’
  rdf:resource=‘‘#ZipCodeFinderProcess’’/>
 <grounding:wsdlDocument Time:FROM=‘1999-01-01’ Time:TO=‘2004-12-31’>
‘‘http://www.dcc.uchile.cl/2005/ws/docum/ZipCode-v1.0.asmx?WDSL’’
 </grounding:wsdlDocument>
 <grounding:wsdlDocument Time:FROM=‘1999-01-01’ Time:TO=‘2004-12-31’>
‘‘http://www.dcc.uchile.cl/2005/ws/docum/ZipCode-v2.0.asmx?WDSL’’
 </grounding:wsdlDocument>
 ....
```

7 Conclusion

Versioning of Web services ontologies has not yet been studied by the Semantic Web community. We introduced OWL-S(T), a formal model for OWL-S, along with a query language supporting a two-level versioning scheme for OWL-S specifications. Our model and query language allow, for instance, simultaneously accessing different versions of the same specification.

A lot of research and practical issues remain open. Among these problems, the development of efficient algorithms for checking consistency, and fixing inconsistent specifications is required. Future work also includes the implementation of our proposal.

References

1. D. Brickley and R.V.(Eds.) Guha. RDF vocabulary description language 1.0: RDF schema. *W3C Recommendation, 10 February 2004.*
2. K. Brown and M. Ellis. Best practices for web services versioning. 2004. http://www-128.ibm.com/developerworks/webservices/library/ws-version.
3. C. Chris Peltz and A. Anagol-Subbarao. Design strategies for web services versioning. *Web Services Journal, SYS-CON Media*, 2004. http://webservices.sys-cop.com/read/44356.htm.
4. Thomas Erl. *Service-Oriented Architecture : Concepts, Technology, and Design.* Prentice Hall, August 2005.
5. Giorgos Flouris, Dimitris Plexousakis, and Grigoris Antoniou. Evolving ontology evolution. In *SOFSEM*, pages 14–29, 2006.
6. C. Gutiérrez, C. Hurtado, and A. Vaisman. Temporal RDF. In *European Conference on the Semantic Web (ECSW'05) (Best paper award)*, pages 93–107, 2005.
7. C. Gutiérrez, C. Hurtado, and A. Vaisman. Introducing time into RDF. *IEEE-TKDE Special Issue on the Semantic Web (in press)*, 2006.
8. P. Haase, J. Broekstra, A. Eberhart, and R. Volz. A Comparison of RDF Query Languages. In *International Semantic Web Conference*, 2004.
9. P. Haase and L. Stojanovic. Consistent Evolution of OWL Ontologies. In *Proceedings of the 2nd. European Semantic Web Conference*, pages 182–197, 2005.
10. P. Haase and Y. Sure. State-of-the-Art on Ontology Evolution. *SEKT/2004/D3.1.1.b/v0.5*, 2004.
11. Patrick Hayes(Ed.). RDF semantics. *W3C Recommendation, 10 February 2004.*
12. I. Horrocks, P. Patel-Schneider, and F. Van Harmelen. From SHIQ and RDF to OWL: the making of a Web Ontology Language. *Journal of Web Semantics*, 1(1):7–26, 2003.
13. M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov. Ontology Versioning and Change Detection on the Web. In *EKAW*, pages 197–212, 2002.
14. A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz. Establishing the Semantic Web 11: An infrastructure for searching, reusing, and evolving distributed ontologies. In *Proceedings of the 12th. International Conference on World Wide Web*, pages 439–448, 2003.
15. A. Magkanaraki, G. Karvounarakis, T.T. Anh, V. Christophides, and D. Plexousakis. Ontology Storage and Querying. Technical Report 308, Foundation for Research and Technology Hellas, Institute of CS, Information System Lab, 2002.
16. F. Mannola and E. Miller. RDF Primer. *W3C Recommendation*, Feb. 2004.
17. David Martin(Ed.). OWL-S: Semantic Markup for Web Services. *OWL-S 1.1 Release.* http://www.daml.org/services/owl-s/1.1.
18. The Mindswap Project. http://www.mindswap.org.
19. M. Smith, C. Welty, and D.L. (Eds.) McGuiness. OWL Web Ontology Language Guide. *W3C Recommendation*, February 2004.
20. L. Stojanovic. *Methods and Tools for Ontology Evolution.* PhD thesis, University of Karlsrhue, 2004.
21. A. Tansel, J. Clifford, and S. Gadia (eds.). *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings, 1993.
22. U. Visser. Intelligent Information Integration for the Semantic web. *Lecture Notes in Computer Science (3159)*, 2004.