Light-weight Ontology Versioning with Multi-temporal RDF Schema

Fabio Grandi
Dipartimento di Elettronica, Informatica e Sistemistica
Alma Mater Studiorum – Università di Bologna
Bologna, Italy
Email: fabio.grandi@unibo.it

***Abstract*—In this paper, we present a multi-temporal RDF data model, which can be used to support RDF(S) light-weight ontology versioning. The data model is equipped with ontology change operations, which are defined in terms of low-level updates acting on RDF triples. As a result, the operational semantics of a complete set of primitive ontology changes has been formalized, taking care of preservation of class/property hierarchies and typing constraints. When used within the transaction template, which has also been introduced, the proposed ontology changes allow knowledge engineers or maintainers of semantics-based Web resources to easily define and manage temporal versions of an RDF(S) ontology.**

***Keywords*-ontology; versioning; temporal data; RDF(S).**

I. Introduction

In some application domains, when an ontology is changed, the past version is required to be kept in addition to the new version (e.g., to maintain compatibility with applications and resources referencing the past one), giving rise to multi-version ontologies. Agents in such domains may often have to deal with a past perspective, like a Court having to judge today on a fact committed several years ago in the legal domain, where ontologies must evolve as a natural consequence of the dynamics involved in normative systems [9]. Moreover, several time dimensions are usually important for computer applications in such domains.

In this vein, we previously considered in [9] ontologies encoded in OWL/XML format and defined a temporal data model for the storage and management of multi-version ontologies in such a format. In [5], [6], we indeed considered ontologies serialized as RDF graphs [19], and introduced a multidimensional temporal data model and query language for the storage and management of multi-version ontologies in RDF format. In particular, since the triple store technology [20] for RDF is supposed to provide scalability for querying and retrieval, the temporal RDF data model we introduced in [5] is aimed at preserving the scalability property of such an approach as much as possible also in the temporal setting. This has been accomplished through the adoption of *temporal elements* [4], [12] as timestamps and a careful definition of the operational semantics of modification statements, which prevents the proliferation of *value-equivalent* triples even in the presence of multiple temporal dimensions.

In this work, we further focus on *light-weight ontologies* expressed with RDF(S), that is based on the vocabulary defined in RDF Schema [18], which are widespread and present a fast sharing rate in the loosely controlled and distributed environment of the Web and Web 2.0 [15]. Relying on the data in [3], Theoaris et al. estimate that 85.45% of the Semantic Web schemas are expressed in RDF(S) [14]. Hence, we will introduce in Sec. II a multi-temporal data model and an operation set, which can be used to support temporal versioning of RDF(S) ontologies. In particular, valid and transaction time dimensions and the types of ontology versioning, which stem from their adoption, are presented in Sec. II-A, the adopted underlying temporal RDF data model is briefly sketched in Sec. II-B, a comprehensive model for temporal RDF(S) ontology versioning is introduced in Sec. II-C and the definition of a complete set of ontology change primitives is provided in Sec. II-D. Conclusions will be finally found in Section III.

II. A Multi-temporal RDF(S) Data Model for Light-weight Ontology Versioning

As RDF Schemas are quite similar to the type system of an object-oriented langauge and, thus, an ontology definition via RDFS closely resembles an object-oriented database schema, one could think to apply temporal schema versioning techniques like those in [8] to ontology versioning. However, there are two main differences between such two worlds, which make this application non straightforward. The first difference is that properties are first-class objects in RDFS and, thus, they cannot be dealt with as components of a class type, like in an object-oriented schema, but must be managed independently. The link between classes and properties is supplied by a sort of third-party tool, represented by domain and range definitions. The second difference is that, whereas in object-oriented databases we can separate the intensional (schema change) and the extensional (change propagation) aspects, in RDF(S) ontologies the two aspects are strictly related, since instances are part of the ontologies themselves and, thus, some ontology changes cannot be performed without affecting instances [16]. However, we will see in Sec. II-C that, with the proposed approach, both these aspects will turn into an advantage.

A. Multitemporal Ontology Versioning

In the temporal database literature, two time dimensions are usually considered: **valid time** (concerning the real world) and **transaction time** (concerning the computer life) [12]. With these time dimensions, likewise schema versioning in databases [2], [8], three kinds of temporal versioning can also be considered for ontology versioning:

- **Transaction-time ontology versioning** allows *on-time* ontology changes, that is ontology changes that are effective when applied. In this case, the management of time is completely transparent to the user: only the current ontology can be modified and ontology changes are effected in the usual way, without any reference to time. However, support of past ontology versions is granted by the system non-deletion policy, so that the user can always *rollback* the full ontology to a past state of its life.
- **Valid-time ontology versioning** is necessary when *retroactive* or *proactive* modifications [2] of an ontology have to be supported and it is useful to assign a temporal *validity* to ontology versions. With valid-time ontology versioning, multiple ontology versions, valid at different times, are all available for reasoning and for accessing and manipulating instances. The newly created ontology version can be assigned any validity by the designer, also in the past or future to effect retro- or pro-active ontology modifications, respectively. The (portions of) existing ontology versions overlapped by the validity of the new ontology version are overwritten.
- **Bitemporal ontology versioning** uses both time dimensions, that is retro- and pro-active ontology updates are supported in addition to transaction-time ontology versioning. With respect to valid-time ontology versioning, the complete history of ontology changes is maintained as no ontology version is ever discarded (overlapped portions are "archived" rather than deleted). In a system where full auditing/traceability of the maintenance process is required, only bitemporal ontology versioning allows verifying whether an ontology version was created by a retro- or pro-active ontology change.

Other time dimensions can also be considered as further (orthogonal) versioning dimensions [5] in special application domains, like *efficacy* or *applicability* time in the legal or medical domains [7], [9].

B. A multi-temporal RDF database model

We briefly recall here the base definitions of the multi-temporal RDF database model [5] underlying our proposal, starting from an N-dimensional time domain:

$$\mathcal{T} = \mathcal{T}_1 \times \mathcal{T}_2 \times \cdots \times \mathcal{T}_N$$

where $\mathcal{T}_i = [0, \text{UC}]_i$ is the i-th time domain. Right-unlimited time intervals are expressed as $[t, \text{UC}]$, where UC means "Until Changed", though such a symbol is often used in temporal database literature [12] for transaction time only (whereas, e.g., "forever" or ∞ is used for valid time). Such naming choice refers to the modeling of time-varying data, which are potentially subject to change with respect to all the considered time dimensions. Then, a multi-temporal RDF triple is defined as $(s, p, o \,|\, T)$, where s is a subject, p is a property, o is an object and $T \subseteq \mathcal{T}$ is a *timestamp* assigning a *temporal pertinence* to the RDF triple (s, p, o). We will also call the (non-temporal) triple (s, p, o) the value or the contents of the temporal triple $(s, p, o \,|\, T)$. The temporal pertinence of a triple is a subset of the multidimensional time domain, which is represented by a *temporal element* [12], that is a disjoint union of multidimensional temporal intervals, each one obtained as the Cartesian product of one time interval for each of the supported temporal dimensions. A multi-temporal RDF database is defined as a set of timestamped RDF triples:

$$\text{RDF-TDB} = \{\, (s, p, o \,|\, T) \mid T \subseteq \mathcal{T} \,\}$$

with the integrity constraint:

$$\forall (s, p, o \,|\, T), (s', p', o' \,|\, T') \in \text{RDF-TDB}:$$
$$s = s' \wedge p = p' \wedge o = o' \Longrightarrow T = T'$$

which requires that no value-equivalent distinct triples exist. The adoption of timestamps made-up of temporal elements instead of (multi-temporal) simple intervals avoids the duplication of triples in the presence of a temporal pertinence with a complex shape [5].

In practice, we store different triple versions only once with a complex timestamp rather than storing multiple copies of them with a simple timestamp as in other RDF temporal extensions [10], [17], [23]. The memory saving we obtain grows with the dimensionality of the time domain but could be relevant also with a single time dimension [5]. Moreover, since temporal elements are closed under set union, intersection and complementation operations, they lead to query languages that are more natural [4].

The data model is equipped with three modification operations —`INSERT`, `DELETE` and `UPDATE`— with a SQL-like syntax also inspired by the SPARQL/Update proposal [22], and whose semantics has been defined in [5] in such a way, that the integrity constraints concerning temporal elements and the controlled growth of value-equivalent triples made possible by temporal elements are automatically maintained. This fact ensures that, in a temporal setting and compatibly with the growth of the number of versions, unlike other approaches, the scalability property of the triple storage technology is preserved.

C. Temporal Versioning of Light-weight RDF(S) Ontologies

The signature of a Light-weight RDF(S) Ontology [15] can be defined as follows:

$$\mathcal{O} = (\mathcal{C}, \mathcal{H}_{\mathcal{C}}, \mathcal{I}_{\mathcal{C}}, \mathcal{P}, \mathcal{H}_{\mathcal{P}}, \mathcal{I}_{\mathcal{P}}, \mathcal{D}_{\mathcal{P}}, \mathcal{R}_{\mathcal{P}})$$

where $\mathcal{C}$ is the set of classes, $\mathcal{H}_\mathcal{C}$ is the class hierarchy, and $\mathcal{I}_\mathcal{C}$ is the set of class instances; $\mathcal{P}$ is the set of properties, $\mathcal{H}_\mathcal{P}$ is the property hierarchy, and $\mathcal{I}_\mathcal{P}$ is the set of property instances; finally $\mathcal{D}_\mathcal{P}$ and $\mathcal{R}_\mathcal{P}$ are, respectively, the set of property domain and range definitions. Hence, a *Multi-temporal Light-weight RDF(S) Ontology* can be defined as a multi-temporal RDF database as:

$$
\begin{aligned}
&O := \\
&\{(C, \texttt{rdf:type}, \texttt{rdfs:Class} \,|\, T) \,|\, C \in \mathcal{C}, T \subseteq \mathcal{T}\} \cup \\
&\{(C, \texttt{rdfs:subClassOf}, C' \,|\, T) \,|\, (C, C') \in \mathcal{H}_\mathcal{C}, T \subseteq \mathcal{T}\} \cup \\
&\{(x, \texttt{rdf:type}, C \,|\, T) \,|\, C(x) \in \mathcal{I}_\mathcal{C}, T \subseteq \mathcal{T}\} \cup \\
&\{(P, \texttt{rdf:type}, \texttt{rdf:Property} \,|\, T) \,|\, P \in \mathcal{P}, T \subseteq \mathcal{T}\} \cup \\
&\{(P, \texttt{rdfs:subPropertyOf}, P' \,|\, T) \,|\, (P, P') \in \mathcal{H}_\mathcal{P}, T \subseteq \mathcal{T}\} \cup \\
&\{(x, P, y \,|\, T) \,|\, P(x, y) \in \mathcal{I}_\mathcal{P}, T \subseteq \mathcal{T}\} \cup \\
&\{(P, \texttt{rdfs:domain}, C \,|\, T) \,|\, \mathrm{dom}(P, C) \in \mathcal{D}_\mathcal{P}, T \subseteq \mathcal{T}\} \cup \\
&\{(P, \texttt{rdfs:range}, C \,|\, T) \,|\, \mathrm{range}(P, C) \in \mathcal{R}_\mathcal{P}, T \subseteq \mathcal{T}\}
\end{aligned}
$$

This definition is useful to understand how ontology changes will be mapped onto manipulation of temporal triples. The general template, which can be used for a transaction which creates a new ontology version, is exemplified in Fig. 1. Such a transaction needs two temporal parameters as inputs: the *ontology selection validity* and the *ontology change validity* (corresponding to schema selection validity and schema change validity in databases [8]). The former (*OS_Validity*) is a valid-time point and is used to select the ontology version —not necessarily the present one— chosen as the starting base, to which ontology changes are applied to produce the new version; the latter (*OC_Validity*) is a valid-time element, that is a disjoint union of valid-time intervals, representing the validity to be assigned to new version resulting form the application of the ontology changes. As far as transaction time is concerned, default conditions are used, since only current ontology versions can be chosen as modification base and the new version must be assigned a [NOW,UC] pertinence.

In Fig. 1, statements 1 and 7 are SQL-like syntactic delimiters for the transaction body. As a first operation (2), a temporary (non-temporal) RDF graph is created to be used as work version of the ontology. Such graph is then populated (3) with the RDF triples making up the work version, extracted as a *snapshot query* from the temporal ontology (i.e., the triples whose temporal pertinence contains *OS_Validity*×{NOW}, with the timestamp projected out). Then, the required sequence of (non-temporal) ontology changes is applied to the work version. When the new ontology version is ready, it must be loaded into the temporal ontology with the desired time pertinence *OC_Validity*×[NOW,UC]. To this end, the contents of the temporal ontology within the time window *OC_Validity*×[NOW,UC] are deleted (4), in order to make room for the new version in the time domain, and the triples making up the work version are inserted as temporal triples into the temporal ontology with the assigned timestamp *OC_Validity*×[NOW,UC] (5). After that, the temporary work version is no longer necessary and can be discarded (6). Notice that, whereas statements 2 and 6 are "standard" (i.e., non-temporal) SPARQL/Update instructions [22], statements 3, 4 and 5 are temporal T-SPARQL operations as defined in [5], [6].

Adopting the template in Fig. 1, schema changes are applied on the work version, which is a traditional (non-temporal) RDF(S) ontology and, thus, there is no need to introduce *temporal* schema change operations. Hence, as a set of possible schema changes, we could even consider the operations made available by an existing ontology editor [21]. Differently from other approaches with a strong logic-based ground (e.g., [11], [13]), our choice is to follow the simpler approach previously used for database schema versioning (e.g., [1], [8]) by considering the set of schema changes in Tab. I: the proposed operations are *primitive*, as each of them acts on a single element of an RDFS ontology and none of them can be decomposed in terms of the others, and make up a complete set. Completeness can easily be checked by verifying that any arbitrarily complex RDFS ontology can be built from scratch (or completely destroyed) via the execution of a suitable sequence of ontology change primitives.

With this approach, we remit in fact the management of the resulting ontology version validity to the responsibility of the designer. For instance, the validity rule R8 enforced by the approach in [13], stating that "the domain of a property is unique", would be too limiting and, thus, unacceptable with respect to the requisites of several application domains. Notice that, in the formalization of some real world component, which is fruit of some human (i.e., arbitrary or at least non completely rational) activity, like the legal domain, a correctly designed ontology could even be logically inconsistent. For example, in several countries, the primary function of the Supreme/Constitutional Court is to rule conflicts between ordinary norms and constitutional laws. As long as such conflicts exist, the whole corpus of regulations is in fact logically inconsistent and as such must be modelled.

The semantics of the primitives in Tab. I will be defined in the next section, taking care of preservation of class/property hierarchies and typing constraints (like in an object-oriented database schema). Moreover, since instances are part of the ontology definition, we do not have in this framework to deal with the interaction between versioning at intensional and extensional levels, extensively discussed in [8, Sec. 4], arising in databases where schemata and data are possibly versioned along different time dimensions.

We underline that, whereas the proposed operation set could also be used for ontology evolution in a non-temporal setting, where only the current version of the ontology is retained, their usage within the template in Fig. 1 gives rise to full-fledged temporal ontology versioning.

```
1. BEGIN TRANSACTION ;
2. CREATE GRAPH <http://example.org/workVersion> ;
3. INSERT INTO <http://example.org/workVersion> { ?s ?p ?o }
   WHERE { TGRAPH <http://example.org/tOntology> { ?s ?p ?o | ?t } .
           FILTER(VALID(?t) CONTAINS OS_Validity && TRANSACTION(?t) CONTAINS fn:current-date()) } ;

   ⇒ a sequence of ontology changes acting on the (non-temporal) workVersion graph goes here

4. DELETE FROM <http://example.org/tOntology> { ?s ?p ?o } VALID OC_Validity ;
5. INSERT INTO <http://example.org/tOntology> { ?s ?p ?o } VALID OC_Validity
   WHERE { GRAPH <http://example.org/workVersion> { ?s ?p ?o } } ;
6. DROP GRAPH <http://example.org/workVersion> ;
7. COMMIT TRANSACTION
```

Figure 1. Template for a transaction implementing the derivation of a new ontology version.

Changes to the class collection
CREATE_CLASS(*NewClass*)
DROP_CLASS(*Class*)
RENAME_CLASS(*Class, NewName*)
Changes to the property collection
CREATE_PROPERTY(*NewProperty*)
DROP_PROPERTY(*Property*)
RENAME_PROPERTY(*Property, NewName*)
Changes to the class and property hierarchies
ADD_SUBCLASS(*SubClass, Class*)
DELETE_SUBCLASS(*SubClass, Class*)
ADD_SUBPROPERTY(*SubProperty, Property*)
DELETE_SUBPROPERTY(*SubProperty, Property*)
Changes to the property domain and range definitions
ADD_DOMAIN(*Property, NewDomain*)
ADD_RANGE(*Property, NewRange*)
DELETE_DOMAIN(*Property, Domain*)
DELETE_RANGE(*Property, Range*)
CHANGE_DOMAIN(*Property, Domain, NewDomain*)
CHANGE_RANGE(*Property, Range, NewRange*)

Table I
LIST OF PRIMITIVE RDFS ONTOLOGY CHANGES.

D. Definition of RDF(S) Ontology Changes

In this section, we show how the primitive ontology change operations in Tab. I can be defined in terms of manipulation operations acting on the RFD(S) contents of the work version. An SQL-like syntax (which seems a bit more intuitive than SPARQL/Update [22] for SQL-acquainted readers) is used to express INSERT, DELETE and UPDATE statements acting on RDF triples. In the following definitions, for the sake of simplicity, although non explicitly written all the manipulation operations are supposed to work on the named graph <http://example.org/workVersion> when embedded into the transaction template of Fig. 1.

The CREATE_CLASS primitive adds a new class to the set of classes $\mathcal{C}$ and can simply be defined as:

```
CREATE_CLASS(NewClass) :=
   INSERT { NewClass rdf:type rdfs:Class }
```

The DROP_CLASS primitive eliminates a class from the ontology. This means that the class must be removed from the set $\mathcal{C}$ and from the class hierarchy $\mathcal{H}_C$, all the domain and range definitions having the class as target must be removed from $\mathcal{D}_P$ and $\mathcal{R}_P$, respectively, and all the instances of the class must also be removed from $\mathcal{I}_C$. Thus, it can be defined as follows:

```
DROP_CLASS(Class) :=
   DELETE { Class rdf:type rdfs:Class } ;
   INSERT { ?C rdfs:subClassOf ?D } ;
      WHERE { ?C rdfs:subClassOf Class .
              Class rdfs:subClassOf ?D } ;
   DELETE { Class rdfs:subClassOf ?C } ;
   DELETE { ?C rdfs:subClassOf Class } ;
   DELETE { ?P rdfs:domain Class } ;
   DELETE { ?P rdfs:range Class } ;
   DELETE { ?X rdf:type Class }
```

Notice that, before Class can be removed from $\mathcal{H}_C$, if $\{(C, \texttt{Class}), (\texttt{Class}, D)\} \subseteq \mathcal{H}_C$, an explicit inheritance link (C, D) must be added to $\mathcal{H}_C$ in order to maintain the continuity of the inheritance hierarchy. We assume the relation rdfs:subClassOf is not interpreted here as transitive (i.e., it only matches explicitly stored triples), so that only one explicit link is added. In this way, we also produce a consistent state without explicitly computing the transitive closure of the inheritance relation, which would increase the number of stored triples in the work version.

The RENAME_CLASS primitive changes the name of a class in the ontology. This means that the name must be changed wherever the class occurs, that is in $\mathcal{C}$, $\mathcal{H}_C$, $\mathcal{D}_P$, $\mathcal{R}_P$ and $\mathcal{I}_C$. The primitive can be defined as follows:

```
RENAME_CLASS(Class, NewName) :=
   UPDATE { Class rdf:type rdfs:Class }
      SET { NewName rdf:type rdfs:Class } ;
   UPDATE { Class rdfs:subClassOf ?C }
      SET { NewName rdfs:subClassOf ?C } ;
   UPDATE { ?C rdfs:subClassOf Class }
      SET { ?C rdfs:subClassOf NewName } ;
   UPDATE { ?P rdfs:domain Class }
      SET { ?P rdf:domain NewName } ;
   UPDATE { ?P rdfs:range Class }
      SET { ?P rdf:range NewName } ;
   UPDATE { ?X rdf:type Class }
```

```
      SET { ?X rdf:type NewName } ;
```

Obviously, the execution of

```
   RENAME_CLASS(ex:C,ex:D)
```

is not equivalent to the sequence:

```
   DELETE_CLASS(ex:C) ;
   CREATE_CLASS(ex:D)
```

because, in the former case, the instances of class `ex:C` are preserved and assigned to `ex:D` (and also instances of properties having `ex:C` as domain or range are preserved), whereas, in the latter, instances are discarded.

The `CREATE_PROPERTY` primitive adds a new class to the set of properties $\mathcal{P}$ and can simply be defined as:

```
CREATE_PROPERTY(NewProperty) :=
   INSERT { NewProperty rdf:type rdf:Property }
```

The `DROP_PROPERTY` primitive eliminates a property from the ontology. This means that the property must be removed from the set $\mathcal{P}$ and from the property hierarchy $\mathcal{H}_P$, all the domain and range definitions having the property as source must be removed from $\mathcal{D}_P$ and $\mathcal{R}_P$, respectively, and all the instances of the property must also be removed from $\mathcal{I}_P$. Thus, it can be defined as follows:

```
DROP_PROPERTY(Property) :=
   DELETE { Property rdf:type rdf:Property } ;
   INSERT { ?P rdfs:subPropertyOf ?Q }
      WHERE { ?P rdfs:subPropertyOf Property .
              Property rdfs:subPropertyOf ?Q } ;
   DELETE { Property rdfs:subPropertyOf ?P } ;
   DELETE { ?P rdfs:subPropertyOf Property } ;
   DELETE { Property rdfs:domain ?C } ;
   DELETE { Property rdfs:range ?C } ;
   DELETE { ?X Property ?Y }
```

As for classes, the deletion of the property in the middle of a inheritance path requires the insertion of a new explicit inheritance link to $\mathcal{H}_P$ before the property is removed, in order not to break the path into two stumps (we also assume the relation `rdfs:subPropertyOf` is not interpreted here as transitive, so that only one explicit link is added).

The `RENAME_PROPERTY` primitive changes the name of a property in the ontology. This means that the name must be changed wherever the property occurs, that is in $\mathcal{P}$, $\mathcal{H}_P$, $\mathcal{D}_P$, $\mathcal{R}_P$ and $\mathcal{I}_P$. The primitive can be defined as follows:

```
RENAME_PROPERTY(Property, NewName) :=
   UPDATE { Property rdf:type rdf:Property }
      SET { NewName rdf:type rdf:Property } ;
   UPDATE { Property rdfs:subPropertyOf ?P }
      SET { NewName rdfs:subPropertyOf ?P } ;
   UPDATE { ?P rdfs:subPropertyOf Property }
      SET { ?P rdfs:subPropertyOf NewName } ;
   UPDATE { Property rdfs:domain ?D }
      SET { NewName rdfs:domain ?D } ;
   UPDATE { Property rdfs:range ?D }
      SET { NewName rdfs:range ?D } ;
   UPDATE { ?X Property ?Y }
      SET { ?X NewName ?Y }
```

The `ADD_SUBCLASS` primitive is used to add a new inheritance link to the class hierarchy $\mathcal{H}_C$ and can simply be defined as:

```
ADD_SUBCLASS(SubClass, Class) :=
   INSERT { SubClass rdfs:subClassOf Class }
```

The `DELETE_SUBCLASS` primitive is used to remove an inheritance link from the class hierarchy $\mathcal{H}_C$ and can simply be defined as:

```
DELETE_SUBCLASS(SubClass, Class) :=
   DELETE { SubClass rdfs:subClassOf Class }
```

The `ADD_SUBPROPERTY` primitive is used to add a new inheritance link to the property hierarchy $\mathcal{H}_P$ and can simply be defined as:

```
ADD_SUBPROPERTY(SubProperty, Property) :=
   INSERT
      { SubProperty rdfs:subPropertyOf Property }
```

The `DELETE_SUBPROPERTY` primitive is used to remove an inheritance link from the property hierarchy $\mathcal{H}_P$ and can simply be defined as:

```
DELETE_SUBPROPERTY(SubProperty, Property) :=
   DELETE
      { SubProperty rdfs:subPropertyOf Property }
```

The `ADD_DOMAIN` primitive is used to add a new domain relationship to $\mathcal{D}_P$ and can be defined as:

```
ADD_DOMAIN(Property, NewDomain) :=
   INSERT { Property rdfs:domain NewDomain } ;
   INSERT { ?X rdf:type NewDomain }
      WHERE { ?X Property ?Y }
```

Notice that, in accordance to [18], properties are allowed to have multiple domains and the resources denoted by subjects of triples with predicate *Property* must be instances of all the classes stated by the `rdfs:domain` properties. Hence, a new instance *NewDomain*(x) must be added to $\mathcal{I}_C$ for each instance *Property*$(x, y) \in \mathcal{I}_P$.

The `ADD_RANGE` primitive is used to add a new range relationship to $\mathcal{R}_P$ and can be defined as:

```
ADD_RANGE(Property, NewRange) :=
   INSERT { Property rdfs:range NewRange } ;
   INSERT { ?Y rdf:type NewRange }
      WHERE { ?X Property ?Y }
```

Notice that, in accordance to [18], properties are allowed to have multiple ranges and the resources denoted by objects of triples with predicate *Property* must be instances of all the classes stated by the `rdfs:range` properties. Hence, a new instance *NewRange*(y) must be added to $\mathcal{I}_C$ for each instance *Property*$(x, y) \in \mathcal{I}_P$.

The DELETE_DOMAIN primitive is used to remove a domain relationship of a property. This means that the domain must be removed from $\mathcal{D}_P$ together with all the instances of the property referencing the domain, which must be removed from $\mathcal{I}_P$. The operation can then be defined as:

```
DELETE_DOMAIN(Property, Domain) :=
   DELETE { Property rdfs:domain Domain } ;
   DELETE { ?X Property ?Y }
      WHERE { { ?X rdf:type Domain }
              UNION
              { ?C rdfs:subClassOf Domain .
                ?X rdf:type ?C }
            }
```

In this case, we assume the relation rdfs:subClassOf is interpreted as transitive during the evaluation of the statement, as we must delete all the instances *Property*$(x, y) \in \mathcal{I}_P$, where x is a member of *Domain* or of any of its subclasses along the inheritance hierarchy.

Similarly, the DELETE_RANGE primitive is used to remove a range relationship of a property. This means that the range must be removed from $\mathcal{R}_P$ together with all the instances of the property referencing the range, which must be removed from $\mathcal{I}_P$. The operation can be defined as:

```
DELETE_RANGE(Property, Range) :=
   DELETE { Property rdfs:range Range } ;
   DELETE { ?X Property ?Y }
      WHERE { { ?Y rdf:type Range }
              UNION
              { ?C rdfs:subClassOf Range .
                ?Y rdf:type ?C }
            }
```

Also in this case, we assume the relation rdfs:subClassOf is interpreted as transitive, as we must delete all the instances *Property*$(x, y) \in \mathcal{I}_P$, where y is a member of *Range* or of any of its subclasses along the inheritance hierarchy.

The CHANGE_DOMAIN primitive is used to change a property domain definition in $\mathcal{D}_P$ and can be defined as:

```
CHANGE_DOMAIN(Property, Domain, NewDomain) :=
   UPDATE { Property rdfs:domain Domain }
      SET { Property rdfs:domain NewDomain }
```

Analogously, the CHANGE_RANGE primitive to be used to change a property range definition in $\mathcal{R}_P$ can be defined as:

```
CHANGE_RANGE(Property, Range, NewRange) :=
   UPDATE { Property rdfs:range Range }
      SET { Property rdfs:range NewRange }
```

In the last two definitions, we assumed instances of Property are not affected by the domain or range changes. If this is not the case, suitable conversion functions must be supplied, as defined in a given namespace, to correctly propagate the change to instances (e.g., cfn:DomainToNewDomain and cfn:RangeToNewRange for literal data). For instance, in the case of CHANGE_RANGE, this can be done as follows:

```
PREFIX cfn: <http://example.org/conv_funct#>
UPDATE { ?X Property ?Y }
   SET { ?X Property cfn:RangeToNewRange(?Y) }
   WHERE { { ?Y rdf:type Range .
             FILTER(isLiteral(?Y) &&
             cfn:RangeToNewRange(?Y))!="") }
           UNION
           { ?C rdfs:subClassOf Range .
             ?Y rdf:type ?C .
             FILTER(isLiteral(?Y) &&
             cfn:RangeToNewRange(?Y))!="") }
         } ;
DELETE { ?X Property ?Y }
   WHERE { { ?Y rdf:type Range .
             FILTER(isLiteral(?Y) &&
             cfn:RangeToNewRange(?Y))="") }
           UNION
           { ?C rdfs:subClassOf Range .
             ?Y rdf:type ?C .
             FILTER(isLiteral(?Y) &&
             cfn:RangeToNewRange(?Y))="") }
         }
```

If the conversion function is able to produce a significant value (i.e., a non-empty string), the new value is used to update the property instances, also involving range subclasses. Otherwise, the property instances, which cannot be converted, are discarded. This correspond, in the terminology of schema evolution, to a combined deployment of the *coercion* and *filtering* techniques [8]. Notice that, for instance, the execution of:

```
   CHANGE_DOMAIN(ex:P,ex:C,ex:D)
```

is not equivalent to the sequence:

```
   DELETE_DOMAIN(ex:P,ex:C) ;
   ADD_DOMAIN(ex:P,ex:D)
```

because, in the former case, the instances of property ex:P are preserved, if domains ex:C and ex:D are compatible or a conversion function exists, whereas, in the latter, instances are in any case discarded.

III. Conclusion and Future Works

In this work, we added another piece to our proposal, already including [5], [6], [9], which involves the extension to the Semantic Web of temporal data models and query languages developed in decades of temporal database research, by focusing on temporal versioning of light-weight ontologies expressed in RDF(S). To this end, we showed how the multi-temporal RDF data model [5] can easily be used to support RDF(S) ontology versioning. The data model has been equipped with a complete set of primitive ontology change operations, defined in terms of low-level modifications acting on RDF triples. Sequences of such ontology changes can simply be embedded into the transaction template we proposed, to be used by knowledge engineers or maintainers of semantics-based Web resources in order to support full-fledged temporal ontology versioning.

In future research, we will consider the design and prototyping of a query engine supporting the execution of T-SPARQL manipulation operations, which implement the ontology change primitives described in this paper. We will also consider the adoption of suitable multidimensional index and storage structures to efficiently support temporal versioning of light-weight ontologies expressed in RDF(S).

References

[1] J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of SIGMOD Conference*, ACM Press, 1987, pp. 311–322.

[2] C. De Castro, F. Grandi, and M. R. Scalas. Schema Versioning for Multitemporal Relational Databases. *Information Systems*, vol. 22:5, 1997, pp. 249–290.

[3] L. Ding and T. W. Finin. Characterizing the Semantic Web on the Web. In *Proc. of ISWC Conference*, Springer-Verlag, LNCS No. 4273, 2006, pp. 242–257.

[4] S. Gadia. A Homogeneous Relational Model and Query Language for Temporal Databases. *ACM Transactions on Database Systems*, vol. 13:3, 1998, pp. 418–448.

[5] F. Grandi. Multi-temporal RDF Ontology Versioning. In *Proc. of IWOD Workshop*, CEUR-WS, 2009.

[6] F. Grandi. T-SPARQL: a TSQL2-like Temporal Query Language for RDF. In *Proc. of GraphQ Workshop*, CEUR-WS, 2010, pp. 21–30.

[7] F. Grandi. A Personalization Environment for Multi-Version Clinical Guidelines. In A. Fred, J. Filipe, and H. Gamboa, editors, *Biomedical Engineering Systems and Technologies 2010*, Springer-Verlag, CCIS No. 127, 2011, pp. 57–69.

[8] F. Grandi and F. Mandreoli. A Formal Model for Temporal Schema Versioning in Object-Oriented Databases. *Data & Knowledge Engineering*, vol. 46:2, 2003, pp. 123–167.

[9] F. Grandi and M. R. Scalas. The Valid Ontology: A Simple OWL Temporal Versioning Framework. In *Proc. of SEMAPRO Conference*, IEEE Computer Society, 2009, pp. 98–102.

[10] C. Gutierrez, C. A. Hurtado and A. A. Vaisman. Introducing Time into RDF. *IEEE Transactions on Knowledge and Data Engineering*, vol. 19:2, 2007, pp. 207–218.

[11] C. Gutierrez, C. A. Hurtado, and A. A. Vaisman. RDFS Update: From Theory to Practice. In *Proc. of ESWC Conference*, Springer-Verlag, LNCS No. 6644, 2011, pp. 93–107.

[12] C. S. Jensen, C. E. Dyreson (eds.), et al. The Consensus Glossary of Temporal Database Concepts - February 1998 version. In O. Etzion, S. Jajodia, and S. Sripada, editors, *Temporal Databases — Research and Practice*, Springer-Verlag, LNCS No. 1399, 1998, pp. 367–405.

[13] G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides. A Formal Approach for RDF/S Ontology Evolution. In *Proc. of ECAI Conference*, IOS Press, 2008, pp. 70–74.

[14] Y. Theoharis, Y. Tzitzikas, D. Kotzinos, and V. Christophides. On Graph Features of Semantic Web Schemas. *IEEE Transactions on Knowledge and Data Engineering*, vol. 20:5, 2008, pp. 692–702.

[15] P. Mika and H. Akkermans. Towards a New Synthesis of Ontology Technology and Knowledge Management. *Knowledge Engineering Review*, vol. 19:4, 2004, pp. 317–345.

[16] N. F. Noy and M. C. A. Klein. Ontology Evolution: Not the Same as Schema Evolution. *Knowledge and Information Systems*, vol. 6:4, 2003, pp. 428–440.

[17] A. Pugliese, O. Udrea, and V. S. Subrahmanian. Scaling RDF with Time. In *Proc. of WWW Conference*, ACM Press, 2008, pp. 605–614.

[18] RDF Vocabulary Description Language 1.0: RDF Schema. W3C Consortium, http://www.w3.org/TR/rdf-schema/ [retrieved 2011-09-10].

[19] Resource Description Framework. W3C Consortium, http://www.w3.org/RDF/ [retrieved 2011-09-10].

[20] K. Rohloff, M. Dean, I. Emmons, D. Ryder and J. Summer. An Evaluation of Triple-store Technologies for Large Data Stores. In *Proc. of OTM Workshops*, Springer-Verlag, LNCS No. 4806, 2007, pp. 1105–1147.

[21] Semantic Web Tools. W3C Consortium, http://www.w3.org/2001/sw/wiki/SemanticWebTools [retrieved 2011-09-10].

[22] SPARQL Update. W3C Consortium, http://www.w3.org/Submission/SPARQL-Update/ [retrieved 2011-09-10].

[23] J. Tappolet, and A. Bernstein. Applied temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL. In *Proc. of ESWC Conference*, Springer-Verlag, LNCS No. 5554, 2009, pp. 302–322.