Change management and validation for collaborative editing of RDF datasets

Manuel Fiorelli*, Maria Teresa Pazienza, Armando Stellato and Andrea Turbati

Department of Enterprise Engineering,
University of Rome Tor Vergata,
Via del Politecnico 1,
00133 Roma (RM), Italy
Email: fiorelli@info.uniroma2.it
Email: pazienza@info.uniroma2.it
Email: stellato@uniroma2.it
Email: turbati@info.uniroma2.it
*Corresponding author

Abstract: The dynamic and distributed nature of the Semantic Web implies that datasets are often the result of collective participation rather than isolated works. Change management, provenance tracking and validation of changes performed by contributing agents are all requirements of systems for collaborative dataset development. Different scenarios may as well require mechanisms to foster consensus, resolve conflicts between competing changes, reversing or ignoring changes etc. In this paper, we perform a landscape analysis of version control for RDF datasets, emphasising the importance of change reversion to support validation. Firstly, we discuss different representations of changes in RDF datasets and introduce higher-level perspectives on change. Secondly, we analyse diverse approaches to version control. We conclude by focusing on validation, characterising it as a separate need from the mere preservation of different versions of a dataset.

Keywords: change management; change validation; collaborative editing; RDF; resource description framework; provenance tracking; version control; change representation; change reversion.

Reference to this paper should be made as follows: Fiorelli, M., Pazienza, M.T., Stellato, A. and Turbati, A. (2017) 'Change management and validation for collaborative editing of RDF datasets', *Int. J. Metadata, Semantics and Ontologies*, Vol. 12, Nos. 2/3, pp.142–154.

Biographical notes: Manuel Fiorelli, PhD is a Research Associate at the University of Rome Tor Vergata, where he carries on research and teaching in the fields of Knowledge Representation and Knowledge-based Systems. He complements his research activity on large-scale semantic interoperability with an interest in the engineering challenges associated with the realisation of software systems in his area of interest. He is author of about 20 publications on workshops, conferences and journals in the fields of Semantic Web, Natural Language Processing and related areas. He participated in the EU-funded project SemaGrow and in the W3C Community Group Ontology-Lexicon. He is currently participating in R&D projects funded by the ISA^2 programme of the European Commission: PMKI and VocBench.

Maria Teresa Pazienza is Full Professor in the Enterprise Engineering Department at the University of Rome Tor Vergata, and Head of the AI Research Group at Tor Vergata University (http://art.uniroma2.it/). She coordinated for several years the NLP working group of the Italian Association for Artificial Intelligence, is in the editorial board of a few International journals, received for two times the IBM Faculty awards for her researches on natural language processing and conceptual knowledge engineering. She is author or coauthor of more than 200 scientific papers. Prof. Pazienza is member of several cooperation activities between the University of Tor Vergata and national/international scientific institutions for jointly developing researches and technologies. She is also involved in technology transfer activities for Italian companies. She is currently involved in initiatives on Big Data.

Armando Stellato, PhD, is Researcher at the University of Rome Tor Vergata, where he carries on research and teaching in the fields of Computer Programming, Knowledge Representation and Knowledge Based Systems. He is author of more than 80 publications on conferences and journals in the fields of Semantic Web, Natural Language Processing and related areas and has been member of the program committees of more than 90 international scientific conferences and workshops. His main interests cover Architecture Design for Knowledge Based Systems, Knowledge Acquisition and Onto-Linguistic interfaces, for which he participated in several EU funded projects (such as Crossmarc, Moses, Cuspis, Diligent, NeOn, INSEARCH, SCIDIP-ES,

AgInfra, SemaGrow) and international research initiatives, such as the W3C OntoLex Community Group. He is currently leading – under a project funded by the ISA2 program – the development of VocBench: an Application for Collaborative Management of RDF Vocabularies.

Andrea Turbati, PhD, is a Research Associate at the University of Rome Tor Vergata. His research interests span across Knowledge Representation and Knowledge Based Systems. He is author of ~20 publications in the Semantic Web area. During his Ph.D. he worked on ontology learning and population from unstructured content. His Ph.D. thesis was about the design and development of CODA (Computer-aided Ontology Development Architecture), an architecture and a framework that extends the unstructured information management framework UIMA to support the generation of RDF data. He is one of the developers of Semantic Turkey, a platform for Knowledge Acquisition and Management, and VocBench (mainly covering its interaction with Sematic Turkey) - a collaborative web-based, multilingual, editing and workflow tool that manages thesauri, authority lists and glossaries using SKOS-XL. He has also contributed to the EU-funded project SemaGrow.

1 Introduction

The Semantic Web (Berners-Lee et al., 2001; Shadbolt et al., 2006) evolved into a global data space (Heath and Bizer, 2011) of interlinked datasets spanning a multitude of topics. Practices for collaborative development of RDF datasets are often justified by the need to subdivide the effort between multiple contributors, often with different competencies in order to cover specific parts of a larger domain or different aspects of its representation (e.g. lexical, conceptual, logical). Actually, Tudorache et al. (2008) argued that true collaboration depends on the possibility for contributors to discuss and work together to a certain extent. Indeed, the effort required to reconcile the occasionally conflicting perspectives of different contributors is an investment, the return of which is more shareable content. In fact, potential or even actual users may be involved in the development process, thus raising ever more the chances of reuse and interlinking.

Collaborative and iterative development processes for dataset development clearly need methodologies and systems to manage changes to a dataset. Our use of the expression change management is not related to management science (Wikipedia contributors, 2017), which is concerned with problem-solving and decision making in managerial contexts, while our focus is on defining, modelling and handling change to RDF datasets in the context of collaborative editing processes. Furthermore, it is necessary to consider that users of a dataset have a different perspective than its developers. Users are interested in changes between published versions of a dataset. On the other hand, developers are interested in the stream of contributions to a dataset, irrespectively of whether they will be part of a new version of the dataset. Developers certainly benefit from mechanisms to record individual changes, discuss them and, depending on the scenario, get a formal acceptance workflow. Conversely, a continuously evolving dataset may be more difficult to use, as resources can be deleted and, in general, their semantic description can be changed in a backward incompatible manner (e.g. an individual is turned into a class). Versioning benefits those users who are interested in a stable access to specific versions of a dataset. OWL 2 (W3C, 2009) supports versioning of ontologies – which are identified through an ontology IRI – by introducing the notion of version IRI that can be used to identify, locate and, then, import a specific version of an ontology in a series.

Datasets on the Semantic Web encompass both factual and conceptual knowledge, as they can be classified as ontologies, thesauri and other Knowledge Organisation Systems (Hodge, 2000), other than mere data. In general, this spectrum of possibilities is managed through a layered approach: at the structural level, these differences are simply ignored by treating these diverse sources as RDF (W3C, 2004) datasets, while higher levels may deal with the semantics of specific modelling languages and applications.

In this paper, we performed a landscape analysis of the field of version control for RDF in the context of collaborative processes for dataset development. The next sections will introduce different facets of version control, including provenance, change discussion, validation and policy enforcement. We observed much interest in the dynamics associated with the identification and efficient storage of changes, possibly allowing querying different snapshots of a dataset. We contended, however, that rejection of changes is an equally, or perhaps more, important facet of version control, which is supported to a different extent by existing approaches. In fact, we observed that some notable systems for collaborative development adopted a slightly different perspective on the problem, focusing on changes rather than snapshots of a dataset.

The rest of the paper is structured as follows. In Section 2, we understand the notion of change in an RDF dataset and how it can be characterised at different levels. In Section 3, we survey existing systems and approaches for version control of RDF datasets. In Section 4, we discuss change validation, and contrast it to versioning, highlighting the commonalities between them as well as the peculiarities of each. Finally, we conclude in Section 5.

This article is an expanded and revised version of the paper (Fiorelli et al., 2017) we presented at the 11th International Conference on Metadata and Semantics Research (MTSR'17). In addition to revising existing

content, we broadened the discussion of some topics, added or replaced (clarification) examples, added the sections on requirements for RDF version control and took into consideration a few additional tools.

2 Understanding change

The term dataset has acquired different meanings within the widespread literature and technical documentation about the Semantic Web and Linked Open Data. In fact, this notion was absent from the specification of RDF 1.0 (W3C, 2004), which only defined the concept of graph as a set of triples. Therefore, two RDF graphs are equal if their associated sets of triples are equal, i.e. contain exactly the same triples (order and multiplicity do not apply to sets). When two graphs contain blank nodes (Hogan et al., 2014), we should in fact talk about graph isomorphism, i.e. equality under an isomorphism that deals with the nameless nature of blank nodes. The SPARQL (W3C, 2008) query language complemented the notion of graph with the concept of dataset, which is a collection of graphs. This notion found its way into the subsequent specification of RDF 1.1 (Cyganiak et al., 2014), which defines a dataset as consisting of one unnamed graph together with zero or more named graphs (each one being a graph associated with an IRI or blank node).

However, in the VoID (Alexander et al., 2011) specification the term dataset is used with a different meaning, to denote "a set of RDF triples that are published, maintained or aggregated by a single provider". A dataset under this definition has a "social" dimension, and therefore is different from a purely mathematical construct, such as an RDF graph. Furthermore, unlike RDF 1.1 and SPARQL, VoID does not base the notion of dataset on the idea of multiple graphs. However, these seemingly different definitions might be closer (or at least share a strong overlap) in practice, since one of the motivations of named graphs (Carroll et al., 2005) was exactly tracking the provenance of RDF data, which is an important facet of the social dimension of the Semantic Web.

Our use of the term dataset will be in between the different meanings described above. Surely, we acknowledge the social dimension of a dataset, as something that is published and maintained by a defined social entity for some purpose. However, we will not investigate social dynamics such as what are the implications of a change in the ownership of a dataset and will focus instead on factual changes to its content only. With this regard, we will mostly assume that a dataset contains a set of triples, but we will occasionally differentiate between the different graphs that compose a dataset.

From the perspective of the users of an editing environment, the different editing operations provided by the environment represent the different types of changes that an RDF dataset can undergo. However, this viewpoint produces an ever-evolving language of changes, which should grow as new operations are added to the editor. Moreover, it would be difficult to reason upon changes expressed in that language: for example, it is impossible to determine in general whether two changes conflict. Furthermore, a change can be reversed only if the editor provides the inverse of the operation that originated the change (i.e. an operation that undoes the effects of the latter). These difficulties are the result of the ad hoc nature of the language used to express and manage changes. Conversely, they disappear when changes to a dataset are reduced to the addition or deletion of triples. This uniform treatment of changes is often acknowledged to Kiryakov and Ognyanov (2002), who claim that the triple is the "smallest directly manageable piece of knowledge". Furthermore, they claim that a triple can only be added or removed, but not modified, because the identity of a triple is solely determined by the identity of its parts. In other words, the RDF data model does not allow relating the deletion of the triple `:s :p :o1` to the addition of the triple `:s :p :o2`, maybe as the change of a property value. Similarly, renaming of a resource is not easy to determine. These higher-level determinations can be based on the operations offered by an editing environment (e.g. the change originated from the operation to replace the value of a property), the human judgement or, as observed by Papavassiliou et al. (2009), some matcher.

At the semantic level, the change of an ontology is similarly determined primarily as the addition or deletion of axioms (Zaikin and Tuzovsky, 2013).

2.1 Blank nodes

Let us consider two datasets d_1,d_2, if we equate them to their associated sets of triples and assume that they do not contain blank nodes, that is to say $d_1, d_2 \subseteq IRI \times IRI \times (IRI \cup LITERAL)$, their difference can be computed by subtracting them as sets. By assuming that d_1 and d_2 are two subsequent versions of the same dataset, we shall define the added triples $\Delta_{add} = d_2 \setminus d_1$ and the removed triples $\Delta_{del} = d_1 \setminus d_2$.

Blank nodes (Hogan et al., 2014) complicate the matter significantly, especially for what concerns the representation of a change (Berners-Lee and Connolly, 2001): blank nodes in fact behave like existentially quantified variables, and as such can be considered like bound variables, which can be renamed at will. In fact, blank nodes should be considered unnamed nodes. Nonetheless, concrete syntaxes and programmatic APIs for RDF usually provide blank nodes with a local identifier, which can be changed each time the dataset is loaded into memory or serialised back to a file.

The need for blank node identifiers in RDF syntaxes arises from the necessity to linearise a model such as RDF that is nonlinear in nature: the purpose of blank node identifiers is to unify all the occurrences of the same blank node in different triples.

Let us consider some examples in the Turtle syntax. In the example below, a pair of square brackets ([]) introduces a fresh new blank node (without the burden of assigning a local identifier to it), the properties of which are represented as a predicate object list

```
[] rdfs:label "a blank node"@en ;
   rdfs:comment "a blank node is …"@en
.
```

Similarly, a predicate object list can be written inside the pair of square brackets, a feature which is particularly useful when the blank node itself is the object of a triple, such as in the following example:

```
ex:SelfSustaining rdfs:subClassOf [
   a owl:Restriction ;
   owl:onProperty ex:sustain ;
   owl:hasSelf true
] .
```

When a blank node occurs as the object of two triples, there is no choice but to use a blank node identifier, like in the example below:

```
ex:john foaf:knows _:aBnode .
ex:alice foaf:knows _:aBnode .
```

These identifiers are scoped to the RDF document in which they occur, so the situation is not much better than in the previous examples using the syntax [].

The instability of blank node identifiers hurts our ability to compare datasets via set differences. Let us consider the following dataset consisting of a single triple:

```
_:b1 rdfs:label "hello" .
```

If a triple is added to the previous dataset, a new serialisation is produced. Since blank nodes have only local scope, new local identifiers are generated:

```
_:b2 rdfs:label "hello" .
_:b2 rdfs:label "world" .
```

Völkel and Groza (2006) observed that a conservative algorithm for comparing RDF datasets never equates two blank nodes from different files. Formally, $d_1, d_2 \subseteq (IRI \cup BNODE) \times IRI \times (IRI \cup BNODE \cup LITERAL)$, such that $blankNodes(d_1) \cap blankNodes(d_2) = \varnothing$, where $blankNodes$ is a function retuning the set of blank nodes mentioned in an RDF dataset.

In our case, the conservative difference consists of a triple deletion and two triple additions. In other words, when we compare two versions of a dataset, the presence of blank nodes may force us to consider larger than necessary differences, which do not capture the actual evolution of the dataset. In the case above, the conservative algorithm suggested to us that the dataset was completely replaced with new content, while a more concise and possibly more accurate interpretation is that an individual triple has been added to the description of the blank node. However, the biggest problems occur when we want to represent the changes themselves, for communicating them, and eventually applying them to another copy of the dataset. Indeed, the assertion 'add triple _:b1 rdfs:label "world"' is not well-defined, unless we assume that _:b1 will be the local identifier of the blank node at the time the change is applied.

In fact, there are a few circumstances in which we can safely mention blank nodes:

- changes are stored together with the data (e.g. in a separate named graph), thus the coreference of blank nodes will be always preserved.
- non-standard options found in most RDF parsers and writers are used to preserve blank node identifiers.

In the general case, Berners-Lee and Connolly (2001) observed that the presence of blank nodes turns the task of comparing two RDF datasets into a problem of graph isomorphism. Sequence comparison is notoriously based on the idea of finding a minimal edit script that transforms a sequence into another. Similarly, dataset comparison can be based on the problem of maximum common subgraph isomorphism.

However, if our goal is to represent a change per se, we shall circumvent the nameless nature of blank nodes and identify them by relying or even introducing some identifying information (Seaborne and Davis, 2010). For example, the semantics of an ontology allows to uniquely identify a blank node through a chain of properties (inverse functional or functional). In the Delta (Berners-Lee and Connolly, 2001) ontology, this is accomplished by replacing blank nodes with variables that are unified with the appropriate nodes in the source graph by matching a sort of context pattern. Similarly, Völkel and Groza (2006) suggest to enrich blank nodes with an inverse functional property holding a unique identifier (e.g. a UUID). Another possibility is to avoid the use of blank nodes altogether, by replacing them with globally unique IRIs (i.e. Skolem IRIs). Indeed, the preference for IRIs over blank nodes (and literals) is a best practice in the context of Linked Data (Berners-Lee, 2006), aiming at a network effect through the use of global identifiers. Under this perspective, blank nodes should be mostly used as intermediate nodes in complex structures, e.g. RDF collections and class descriptions.

Auer and Herre (2007) follow a different approach, by constraining the granularity of changes: it is not possible to add/remove individual triples about a blank node, but blank nodes can only be destroyed and recreated as a whole. To that purpose, they introduce the notion of atomic graph, subsequently used to define positive and negative changes. A graph is said to be atomic if it can't be subdivided into two graphs the blank nodes of which are disjoint. A ground graph (i.e. without blank nodes) is atomic if and only if it consists of a single statement. A negative atomic change (deleted triples) is an atomic graph that includes every triple that affects a given blank node, and this must hold for any other blank node introduced transitively. In other words, a negative atomic change identifies relevant blank nodes through a syntactic context consisting of every statement related to those nodes. A positive atomic change (added triples) is an atomic graph that never mentions blank nodes

already in the dataset. We illustrate this approach via an example. Let us consider the following axioms, stating that a wine has a maker and a grapevine:

```
ex:Wine a owl:Class ;
   rdfs:subClassOf _:c1 ;
   rdfs:subClassOf _:c2
.
_:c1 a owl:Restriction ;
   owl:onProperty ex:maker ;
   owl:minCardinality 1
.
_:c2 a owl:Restriction ;
   owl:onProperty ex:grapevine ;
   owl:minCardinality 1
.
```

Suppose that we want to update the ontology above to use a qualified number restriction telling that the maker of a wine is an ex:Winery. With respect to the graph above, we would like to:

- remove the triple _:c1 owl:minCardinality 1
- add the triples _:c1 owl:minQualifiedCardinality 1 and _:c1 owl:onClass ex:Winery

The triple deletion is not atomic, because the dataset contains other statements where _:c1 occurs. Following Auer and Herre, we have to include those statements in a negative atomic change, which essentially deletes the class axiom and the property restriction as a whole. Accordingly, the addition of two triples is no longer sufficient, because the class axiom shall be recreated from scratch. The end result is the compound change below consisting of a negative atomic change followed by a positive atomic change:

```
(
  [NegativeAtomicChange] {
    ex:Wine rdfs:subClassOf _:c1000 .
    _:c1000 a owl:Restriction ;
          owl:onProperty ex:maker ;
          owl:minCardinality 1
    .
  }
  ,
  [PositiveAtomicChange] {
    ex:Wine rdfs:subClassOf _:c2000 .
    _:c2000 a owl:Restriction ;
          owl:onProperty ex:maker ;
          owl:minQualifiedCardinality 1 ;
          owl:onClass ex:Winery
    .
  }
)
```

In the negative change, we deliberately used a blank node identifier, _:c1000, that does not occur in the source dataset. The blank node _:c1 is chosen, because if we equate them, then the negative change includes every statement in the source graph about _:c1. Clearly, _:c2 can't be picked, because the negative change does not agree on the value of the property owl:onProperty (ex:maker vs ex:grapevine). The identified subgraph in the source dataset is removed, and replaced with the triples in the positive atomic change. Notably, these triples can't mention the previous blank node (i.e. _:c1000), but instead they shall introduce a new one (i.e. _:c2000). Actually, there would be no point in insisting on recreating the same blank node, because the negative change removed any mention of it from the source graph.

2.2 Representing changes in RDF

RDF is a compelling choice for the representation of changes to an RDF dataset. Actually, we should distinguish two complementary uses:

- represent metadata about changes
- represent the content of a change (i.e. the actual modification of the dataset)

Regarding the first point, RDF is particularly convenient because of its support to the simultaneous use of multiple vocabularies. Therefore, it is possible to combine different vocabularies to describe diverse facets of a change. Moreover, the description of a change can be based on widespread vocabularies, which can be used to record the creator of a change, the instant it was issued (at the desired resolution), the resources it affects or a textual message describing the change and its motivation. Secondly, the adoption of RDF to describe the changes to datasets allows reusing the same tools and methodologies already applied to data, while also enabling interesting scenarios in which the description of a change references another resource on the Semantic Web. Let us assume, for example, that the creator of a change is represented with resources in the knowledge base of an organisation. Leveraging background knowledge about people in the organisation, it is possible to find changes the creators of which work in a given department. Actually, search criteria can be arbitrarily complex, since the SPARQL query language enables ad hoc searches.

After discussing the use of RDF to represent metadata about changes, we report on different approaches to record their content. In Section 2, we have shown that a change can be conveniently reduced to the addition and removal of triples. Therefore, the crux of the problem lies in the representation of triples and their binding to the change that introduced or removed them. Following Seaborne and Davis (2010), we review different approaches, concluding that reification is generally inefficient and that there is a need for some construct to explicitly represent graphs.

The Delta ontology can be used in conjunction with N3 (Berners-Lee and Connolly, 2011) (a superset of RDF), to

compactly represent changes, through the possibility to quote graphs and use them as components of a triple. Additionally, variables can be used to select a resource based on some identification property (see Section 2.1).

In the following example, we use the Delta ontology to represent that the approximate position of something identified by the mailbox someone@example.org changed from Lazio (an administrative region of Italy) to Rome (the capital of Italy and a city located in Lazio).

```
@prefix dbr: <http://dbpedia.org/resource/> .
@prefix foaf: <http://xmlns.com/foaf/0.1/ > .
{ ?x  foaf:mbox <mailto:someone@example.org> }
 diff:deletion {
    ?x  foaf:based_near dbr:Lazio
 } ;
 diff:insertion {
    ?x  foaf:based_near dbr:Rome
 } .
```

The change above can be represented similarly through a SPARQL 1.1 update:

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX foaf: < http://xmlns.com/foaf/0.1/ >
DELETE { ?x  foaf:based_near dbr:Lazio }
INSERT { ?x  foaf:based_near dbr:Rome }
WHERE { ?x foaf:mbox <mailto:someone@example.org>  }
```

The example regarding the use of the Delta ontology, as well as its translation as a SPARQL 1.1 update aren't really based on RDF.

An alternative approach is to use standard RDF reification to represent added and removed triples as resources (e.g. instances of the class rdf:Statement). Different variations of this approach arises as diverse mechanisms are used to group reified statements and connect them to the resource representing a change. Specifically, the statements can be linked to the change individually, or they can be grouped into an RDF collection, container or other resource. Furthermore, the distinction between addition and deletion can be encoded in the name of the linking property, or in different classes (of statements and groups, respectively). Reification isn't space efficient, because we need at least three triples to reify a statement (for its subject, predicate and object) plus a triple to link the reified statement to the change or to a grouping resource (e.g. the group of added triples and the group of removed triples). In the latter case, other triples are required to link the groups to the change. Furthermore, reified triples are difficult to read in a serialised RDF document.

Recently introduced in RDF 1.1 (but long supported by major triples stores), named graphs can be used in place of standard reification: added and removed triples are asserted in two different named graphs, which are then related to a resource representing the specific change.

Unfortunately, named graphs are quite weak as an isolation mechanism: for example, inference and SPARQL queries (by default) are computed over all graphs. Moreover, there could be interoperability problems with general-purpose RDF management systems, which often rely on named graphs for other purposes (e.g. to store imported ontologies). A solution to these problems may be found in a hybrid approach (Cassidy and Ballantine, 2007) combining named graphs with reification (which notoriously does not entail the assertion of a triple). Another option is to save changes in a separate triple store, so that inference/querying problems can be simply ignored. Another opportunity is an explicit support from triple stores, which should implement quintuples including a component for tracking the lifecycle of the associated quadruple (i.e. a triple plus a graph name).

2.3 Higher-level changes

So far, we have characterised a change to a dataset in terms of the triples it adds or removes. Additions and deletions of triples can be understood as part of the same change, or they can be modeled as separated positive and negative changes, which can be composed into a single complex change. Several works (Noy et al., 2006; Auer and Herre, 2007) suggest that changes can be grouped hierarchically, usually to reflect some higher-level change. Specifically, Auer and Herre first introduce the notion of atomic change (see Section 2.1), and then define (general) changes inductively as sequences of changes. The grouping of changes can be based on the transactional boundaries of an application or the constructs found in a modeling language. For instance, the creation of a collection is a compound change, composed by several lower-level changes for the creation of the resources and the links necessary to the representation of the collection.

In a certain sense, we are adopting a higher-level perspective that is more specifically bound to the modelling vocabulary at hand. Thus, not only we group together changes by differentiating between levels of granularity, but we also recognise different types of changes. For instance, if a dataset encodes an OWL ontology, we may recognise changes such as "addition of a class", "merging of two classes", and so on. Auer and Herre (2007) suggest the classification of composite changes with respect to ontology evolution patterns. These patterns are associated with data migration algorithms, which serve two purposes. On the one hand, they precise the intention of a change to an ontology in terms of the desired changes to the facts. On the other hand, downstream users of an ontology are facilitated in the adoption of a newer version, since they can use the migration algorithms to change the instances they have already described. The ChAO ontology (Noy et al., 2006) is similarly based on the classification of changes to an ontology, but it does not register the affected triples. Klein et al. (2002) use a set of rules to translate low-level triple changes to higher-level changes specific for ontology versioning. Papavassiliou et al. (2009) are similarly concerned with higher-level changes, although their focus is limited to RDF/S knowledge bases. These authors define a language of changes, which should be concise, intuitive and support the unambiguous interpretation of low-level triple changes. They also define an efficient algorithm to recognise such changes starting from the low-level changes. Both Klein et al. and Papavassiliou

et al. propose a divisive approach, while somehow symmetric to the aggregative one of Auer and Herre.

We observe that this higher-level perspective over change has two main benefits. From an intellectual perspective, it is certainly easier to understand the intention of a change, when the purpose of low-level triple modifications has been decoded against a classification of changes. Additionally, it allows searching and filtering changes for different purposes: e.g. to look only at terminological changes, rather than to taxonomical ones. Some scenarios may even require that different types of changes can be proposed and approved by people with different roles.

Papavassiliou et al. (2009) made the interesting observation that a high-level change may have a condition, consisting of unchanged triples that must be asserted in order for the change to be defined. For instance, changing the domain of a property :p from :Male to :Person, can be understood as the higher level change Generalize_Domain(p, Male, Person), only if the original dataset entails that :Male is a subclass of :Person.

3 RDF version control

In the previous section, we discussed the notion of change in an RDF dataset, and how changes can be represented in an economical, robust and intuitive way. However, manual sharing and application of changes require much diligence and are impractical at scale, especially if it is desirable to guarantee a globally consistent history of a dataset.

There is thus a need for methodologies and tools that support the proper management of changes and versions of a dataset. Most works in this area are clearly inspired by version control systems in the software development domain. Unsurprisingly, we observed a change in the referenced systems, from (nowadays) legacy systems such as CVS (http://www.nongnu.org/cvs/) and those losing hype, as in the case of Subversion (https://subversion.apache.org/), to current ones such as GIT (https://git-scm.com/). We also observed an approach (Cassidy and Ballantine, 2007) inspired by Darcs (http://darcs.net/), the distinguishing feature of which is the focus on changes rather than snapshots. These systems transitioned from centralised architectures to decentralised ones, a trend that is only marginally reflected in the relevant works in the area of Semantic Web. In fact, even in the software development world, distribution is exploited to a limited extent, since most development workflows depend on a centralised repository as the single source of truth about the history of a project.

In the domain of RDF evolution, Kiryakov and Ognyanov (2002) observed that an RDF version control system should serve two classes of stakeholders: dataset developers and users. As claimed in the introduction, these two groups have different requirements on the system. These requirements have been collected from the relevant literature and in particular from the work of Noy et al. (2006).

3.1 Requirements for dataset developers

The development of a dataset, especially in collaborative settings, clearly benefits from a record of individual changes and intermediate (unpublished) versions of a dataset. This functionality can be provided by a standalone system or embedded in a (collaborative) editor. There could be different development workflows, and the system may record changes or temporary versions. The acceptance of changes can be subjected to human validation, and it may also undergo other levels of policy enforcement.

3.1.1 Embedded vs. standalone solution

An RDF editor can manage the version control system on behalf of the user, whose degree of control can be limited through the enforcement of policies based on other dimensions, e.g. depending on the granularity, each action can be logged as-is, or the user can be allowed to commit a group of actions as a cohesive unit of work. At the opposite end of the spectrum, a standalone solution is only concerned with the management of changes and possibly of versions.

3.1.2 Granularity

When the version control system is embedded inside an editor, user activity can be continually recorded, at the level of detail dictated by the specific application. On the other end, the system may only allow users to save a snapshot of the dataset. The simplest implementation amounts to a sort of export functionality. A slightly more complex approach is a periodic backup facility operating in the background.

3.1.3 Synchronous vs. asynchronous workflow

In a synchronous workflow, all contributors share the same workspace, so that changes performed by anyone are immediately visible to the others. Alternatively, the contributors can work on independent copies of the dataset, while their contributions are merged asynchronously. A version control system may also support separate development lines, which are called branches. The synchronous workflow is typical of many centralised collaborative editing environments, since it may be inconvenient to allocate different workspaces to each contributor. Moreover, the synchronous workflow promotes the practice of continuous integration, while in the asynchronous scenario it depends on how frequently contributions are merged. Continuous integration is a risk-reduction practice, since conflicts between individual contributions can be discovered early on.

3.1.4 Undo of individual changes vs. checkpoint restoration

Let us consider the following evolution scenario in which three classes, in order A, B and C, are added to a dataset. A pair of import/export facilities only allows to restore the state of the dataset after each addition, namely {A,B,C},

{A,B}, {A}. It is not possible to just undo the addition of the class B, leaving the dataset into the state {A, C}. In fact, checkpoints are sufficient to achieve that goal, but we need an additional support to compute the differences between different versions and check whether individual changes can be discarded without negative consequences on other changes.

3.1.5 Tagging

This is the ability to give a name to a particular snapshot of a dataset. This feature eases the reference to meaningful states of the dataset.

3.1.6 Validation

It is not really a requirement on the version control system, but a common usage scenario. Specifically, validation is associated with an explicit approval workflow that is executed in order to accept and consolidate proposed changes.

3.1.7 Policy enforcement

It may be interesting to enforce some access control rules, i.e. establishing who can modify which portion of the dataset. With respect to a dataset about the activities of an organisation, some access rules may limit the possibility to add/remove personnel to members of the HR department, while only project managers could be allowed to update the status of the project they are responsible for. The above can be generalised into the possibility to enforce any policy upon the suggestion of a change. For instance, in the context of a multilingual dataset, we may enforce different requirements on the creation of a new concept: i) a label is provided for each supported language, ii) a label is provided for at least two supported languages, etc.

3.1.8 Communication

The system should foster communication, e.g. to describe a change, to discuss it or even to participate in a voting or other consensus-forming activities.

3.2 Requirements for dataset users

Users of a dataset have a rather different perspective than developers, because users are only interested in the published versions of the dataset, rather than in arbitrary snapshots produced by the development process.

3.2.1 Published versions

It should be possible to reference the latest version of a dataset, as well as to obtain the preceding and subsequent versions of a given version. Usually, the published versions are only a small subset of the ones produced by the development process, therefore users have a much coarser grained perspective than maintainers have. While developers can reasonably reference any intermediate state of the dataset, users benefit from tags that record the released versions.

3.2.2 Lines of backward compatibility

From a user's perspective, a dataset evolution is much more linear than it appears to maintainers. Branching and merging are usually out of the scope of end users, who primarily perceive the evolution of a dataset as a temporally ordered sequence of published versions. In fact, branches could be introduced because of a non-backward compatible change, which effectively starts a new line of backward compatible development.

3.3 Analysis of existing RDF version control systems

We have surveyed some works on version control for RDF datasets, analysing their approaches and, when appropriate, establishing connections or drawing comparisons.

Kiryakov and Ognyanov (2002) propose a unified framework encompassing change-tracking, access-control and metadata management. These authors are often acknowledged for having established the idea that triples are the unit of management. Each edit of the data is individually registered, and specific states of the dataset can be tagged as versions. Individual versions, resources and statements can be described via metadata, at least ideally represented in RDF.

The dataset is associated with an update counter, which is incremented monotonically upon each edit, establishing a logical clock. This counter is used to annotate each triple with its creation and deletion time, making it possible to identify the triples that were valid during a given interval.

When the dataset is persisted in a relational database, the metadata about triples can be recorded efficiently, by extending the representation of the dataset itself. For example, let us assume that statements are stored in a table with columns for their subject, predicate and object. In this case, it is sufficient to add a few columns to that table, in order to store the creation and deletion time of triples as well.

Deleting a triple sets its deletion time, but does not remove it from the repository, whose size can only increase monotonically. However, the growth of the repository can be limited, by purging the history prior to a certain state. The system manages automatically the versioning of imported (read-only) data, as well as inferred triples, by applying techniques of truth maintenance to decide their validity interval. The system, implemented over the OpenRDF Sesame middleware for RDF, now Eclipse RDF4J (http://rdf4j.org/), is intended to support anything that can be encoded in RDF, from factual data to ontologies. It is possible to branch a dataset, but this is quite an onerous operation based on cloning the entire repository.

Völkel and Groza (2006) present SemVersion, a full version control system for RDF (implemented as a pure Java library) that supports all operations commonly found in a version control system, such as checking out, branching, merging and computing differences between states. They represent a change via the TripleSet ontology, which relies on reification to represent added and removed triples. A change contains a link to the preceding one in the branch, and the sequence of changes leading to a given state can be

seen as a delta-compressed representation of that state. This representation is used to compute the difference between two arbitrary states, and to merge two branches. Merging a branch into another fails when the former adds a triple about a resource that was present in the most recent common version of the two branches and that was removed in the receiving branch.

Blank nodes are explicitly supported through the enrichment mechanism described in Section 2.2. Like Kiryakov and Ognyanov, Völkel and Groza focus on RDF at the structural level, on top of which another semantic level can be defined. This further level is characterised by a semantic difference function, which compares two states accounting for inferences enabled by an ontology language. Different notions of semantic conflict can be defined, in addition to the structural level conflict between additions and deletions.

Cassidy and Ballantine (2007) describe a version control system for RDF based on the theory of patches. Starting from an empty dataset, a given version of the dataset can be obtained by applying a sequence of patches (the authors' name for change): $c_1 \cdot c_2 \cdot \ldots \cdot c_n = V_n$. Each patch adds or removes triples, and may be conditioned on unchanged triples. The implementation instruments the Redland (http://librdf.org/) RDF API to intercept any editing operation over the triple store containing the working graph. The patches are stored in a dedicated quad-store, in which each patch is stored in a separate named graph, while individual triples are represented as reified triples. The use of a separate dataset for patches means that the performance of read-only operations over the working dataset are completely unaffected by the tracking system. There is no specific support for blank nodes, nor is there for inference.

To restore the previous version (V_{n-1}), it is sufficient to forget the last change, and replay the full history from the empty dataset. However, the longer the sequence of patches, the more computational demanding this operation is. An alternative is to start from the working state (V_n) and undo the effect of the last change (c_n), by applying its inverse (c_n^{-1}), obtained by swapping additions and deletions.

The theory of patches also supports the undo of intermediate changes in the history. To achieve that, adjacent patches should be commuted, so that the relevant change is moved to the last position, where it can be reversed by applying its inverse. The key observation is that two patches can be commuted freely, unless they conflict. A conflict occurs between patches A and B, when A depends on a triple added by B, or either deletes a triple added by the other. Clearly, if two patches conflict, they cannot be reordered, unless some conflict resolution strategy is employed. By applying the same operations, it is possible to support branching and then merging of different branches.

Im et al. (2012) describe a version management system for RDF based on a relational representation. The system only stores the latest version of graph, plus the delta for reconstructing the previous one. A query against a previous version is rewritten so that it utilises the current graph and all deltas up to the right version. To reduce the response time, Im et al. introduce the notion of aggregated delta, which can be computed in advance between each pair of versions. In this manner, the query should be rewritten using the current graph and only one aggregated delta. Obviously, this approach trades space occupation for response time.

Vander Sande et al. (2013) develop a versioned RDF store on top of an arbitrary quad store, by relying on the mechanism of named graphs to store individual changes. Each change consists of two named graphs (one for the additions and one for the deletions), as well as some metadata describing the change itself. The identifier of a change is externally unique, since it is based on the hash of the change content. That enables a push/pull mechanism similar to one found in distributed version control systems for source code. There is no specific provisioning for blank nodes. The system also supports branching and merging, and conflicts are only determined at the structural level as addition/deletion conflicts. The history can be purged and deltas be rebased to shorten the history. A change is linked to its parent, so that the state of the dataset after each change can be easily reconstructed. Differently from Im et al., queries against an arbitrary version first require that the relevant version is materialised beforehand.

Graube et al. (2014) describe a system similar to the one presented by Vander Sande et al. Both use named graphs to efficiently store deltas, and use RDF to describe individual changes. However, this new work avoids the use of hashmaps and other non-RDF structures. The system is again a standalone versioning system, supporting branching and merging. Its interface is based on an extension of SPARQL with keywords added to reference specific versions. While Vander Sande et al. uses a quad store to version an individual graph, this work uses a quad store to version a collection of graphs (although each graph is versioned independently). Differently from Im et al., queries against an arbitrary version first require that it is materialised in a temporary named graph. To speed-up common scenarios, the latest version of each graph is stored in the respective named graph. Additionally, tagged versions are materialised as well.

Halilaj et al. (2016) observe that some community-driven datasets have already adopted GIT for their version management needs. The perspective of these authors is on vocabulary development, thus they propose a set of best practices to extend (when necessary) GIT to meet the requirements of collaborative vocabulary development.

In fact, they observe that GIT already meets some of them, including flexible workflow support, branching and tagging of versions. Text-based version control systems are based on textual diff algorithms, which are known to have problems with non-linear data such as RDF (Berners-Lee and Connolly, 2001). In particular, Halilaj et al. commit on the use of the Turtle syntax to sidestep the fact that editing tools may produce arbitrary different representations of the same graph because of differences in their writing algorithms.

Other requirements are met by other systems integrated in the GIT ecosystem, such as JIRA (https://www.atlassian.com/software/jira) and other issue tracking systems to support communication and coordination. Halilaj et al. propose the use of OWL2VCS (Zaikin and Tuzovsky, 2013) as a means to compare two versions on a higher level than raw triples. Other requirements can be met by a combination of native

functionality and purpose-built hook, i.e. scripts triggered upon certain events, such as before and after a commit.

Arndt et al. (2016) designed the Quit Store ("Quads in Git"), which implements a SPARQL 1.1 endpoint on top of a GIT repository. In this system, the graphs in the dataset can be mapped to different files in the repository. Like Halilaj et al., they contend that GIT can already track the evolution of RDF datasets suitably serialised. In particular, the Quit Store stores the data in alphabetically sorted N-Quads files. N-Quads is a line-oriented syntax, particularly suitable for GIT, which considers lines as the unit of comparison between textual files. Furthermore, since blank nodes are not supported, sorting guarantees that the serialisation is deterministic, and that differences are not introduced accidentally by a simple reordering of quadruples. Differently from Halilaj et al., users are not intended to edit these serialisations by hand, while they should submit updates to a SPARQL 1.1 endpoint synchronised with the GIT repository. After a SPARQL update, the Quit Store writes back the dataset to files and uses the command `git add --update`, to create a new commit only if some files were actually modified. Like Graube et al., the Quit Store can handle multiple graphs, and furthermore it can handle changes spanning over different graphs.

4 Change validation

In curated datasets, change validation is about reviewing proposed changes to a dataset, in order to reject changes that are deemed wrong, low-quality or otherwise not suitable to make their way into the dataset. Differently, versioning concerns recording and accessing different states of an evolving dataset. Nonetheless, the two activities are related in various ways, since validation can be defined on top of the concepts and even the systems we described for version control.

We first observe that change validation can be implemented on top of any version control system supporting branches. Indeed, Völkel (2006) shows that proposed changes can be allocated to dedicated branches, while accepting a change requires merging the corresponding branch into the main development branch. This workflow is clearly asynchronous in nature, since individual contributions happen in isolated branches of the dataset, which are possibly merged back in case of positive validation. Notably, rejected changes do not affect the history of the main development copy, as it is updated only because of the acceptance of changes. An important downside of this asynchronous workflow is that conflicts between changes by contributors working on independent copies of the dataset are only identified later on, upon the first attempt to merge them.

The notion of conflict is indeed another point of contact between versioning and validation. Despite slight differences between various notions of conflict, there is an agreement on the idea of addition/deletion conflict at the structural level. Let us consider, for example, the three-way merge of two states `B` and `C` originated from a common state `A`. In this case, it should not be possible for either `B` or `C` to remove a triple that is added by the other (Vander Sande et al., 2013). However, such conflicts cannot happen in our three-way merge scenario, if we consider actual triple additions/deletions with respect to the common ancestor `A`. In this setting, neither `B` nor `C` can delete a triple that is added by the other, because either that triple was present in the common ancestor A (thus addition is not possible) or it was not (thus deletion is not possible). Following Völkel and Groza (2006), we will discuss that a similar addition/deletion conflict can happen at a higher level. At the semantic level, we can identify other forms of conflict, such as breaking the consistency and coherency of the ontology being edited. In addition to structural and consistency/coherency conflicts, there is a range of conflicts that sit somehow in the middle, as they depend in part on the specific modelling vocabularies and in part on the applications.

Let us consider the suggestion of a new label for an ontology concept C, that is to say the addition of a triple such as the following:

```
:C rdfs:label "concept C"@en .
```

Furthermore, let us assume that the concept `C` is removed from the ontology before that suggestion is revised. In other words, let us assume that any triple involving that concept is deleted. Intuitively, it should not be possible anymore to accept the label contribution, since the labelled class no longer exists. Unfortunately, from a structural perspective there is no conflict at all, since the contributed triple is not among the triples deleted because of the deletion of the concept C. This problem can be solved by relying on the possibility (e.g. Cassidy and Ballantine, 2007) of adding a condition to a change. Indeed, we may express that the addition of a label depends on the fact that the subject resource exists in the dataset. In our example, the simplest way to require the existence of the concept C is to condition the change on the triple `C rdf:type rdfs:Resource` (assuming the ability of a reasoner to infer that anything that is locally defined is at least a resource). Völkel and Groza (2006) solve a similar problem in the context of three-way merge of two branches: if a branch has deleted any mention of a resource, that resource is considered deleted, therefore the other branch is not allowed to add new mentions of that resource, otherwise producing a higher-level addition/deletion conflict.

Change validation does not necessarily require the use of branches, and it can be implemented in synchronous workflows as well, when all contributors work simultaneously on the same working copy of the dataset. This is a form of continuous integration that reduces the risk of subsequent conflicts, although they are not completely removed. In this synchronous workflow, we no longer maintain separate copies of the dataset for each contribution, but rather we are interested in recording the changes applied to a dataset, so that they can be evaluated and, if necessary, reversed.

Most version control systems for RDF use a delta-compressed representation of the history of a branch, so it is relatively easy to determine what has changed because of a committed change. Auer and Herre (2007) suggest to implement arbitrary change rollback, by checking the compatibility of a change with a version of a graph different from the one the change was originally created.

A change can be reversed easily, by creating a new change, a sort of inverse change, that undoes its effects, by swapping additions and deletions of triples. Actually, the application of the inverse change may fail because of a conflict, when a subsequent change somehow overwrote the one we want to reject. Actually, it is right that the change can no longer be rejected. In fact, it may be the case that a change is not atomic (e.g. it adds two classes), and it may be the case that the conflict arises only because of a part of the change. In such cases, we can exploit the ability (previously discussed) of decomposing a change into cohesive changes, which can be independently rejected.

In this scenario, reversing a change results in another change being registered by the version control system. In a certain sense, a change and its inverse share the same nature, and thus we could even undo the undo of a change, and so on. In fact, we advise in favour of adding some metadata to tell the difference between these two types of change.

As discussed in the previous section, Cassidy and Ballantine (2007) shift the focus from version management to change management, by adopting the theory of patches. The system manages the sequence of patches that lead to the current working state of a dataset. When the changes are not conflicting, they can be reordered freely, therefore the sequence is in fact a set of changes, which can be rejected independently.

A similar interest in change management can be recognised in many collaborative editors, such as VocBench 2 (Stellato et al., 2015), VocBench 3 (Stellato et al., 2017), PoolParty (https://www.poolparty.biz/), TopBraid EVN (https://www.topquadrant.com/products/topbraid-enterprise-vocabulary-net/) and Protégé (Noy et al., 2006). The additions of collaboration features to Protégé have eventually led to its web-based incarnation (Tudorache et al., 2013).

The management of changes clearly presupposes these to be identified in the first place. To this end, there are two main strategies: monitored vs non monitored. Protégé supports both approaches (Tudorache et al., 2008), while VocBench 2/3, TopBraid EVN and PoolParty mainly support the monitored approach. The monitoring of changes consists in recording the changes as they are produced. Usually, it requires the instrumentation of the editing environment or a low-level middleware to intercept and record individual editing actions. Without monitoring, there are only two versions (before and after the modification), and then a comparison function is used to compute the difference. If the second version of the dataset contains numerous modifications, we must break down the triple-based difference into numerous more cohesive changes, so that they can be analysed independently.

A general argument in favour of a monitored solution is its higher efficiency, since comparing two large RDF graphs can be computationally expensive (no wonder that among the above cited tools, the one supporting non monitored changes is meant to support ontology development, while the other two deal with large thesauri). Moreover, the transactional boundaries of an application offer a natural criterion for defining atomicity of changes. In fact, it may be useful to further decompose changes into hierarchies, so that it is possible to analyse the changes at different levels of detail. Another advantage of the monitored approach is that individual changes are identified early on, so that it is possible to annotate them and, in some cases, start discussions about them.

One disadvantage of the monitored scenario is that changes naturally occur in a temporal sequence, and it may happen that subsequent changes are redundant, possibly conflicting. As an example, let us consider a team of maintainers working simultaneously on the same ontology. If the maintainers are in disagreement, a same class might happen to be repeatedly added and deleted. These changes are both conflicting and redundant, and would pollute the list of changes pending for acceptance. It is a matter of policy, whether the system should react to these problematic cases or the application should prevent such cases to occur in the first place.

VocBench, PoolParty and Protégé (in the monitored configuration) share a similar architecture. All users work simultaneously on the same data, while changes are tracked and stored. All of the systems record high-level changes. VocBench 2 models changes as a Java class hierarchy, and stores the objects representing individual changes into a separate relational database. VocBench 3 represents changes as RDF data in a support triple store. In VocBench 2, the addition of a new mutation operation was usually accompanied by the addition of a new class to represent its occurrences in the history. Additionally, it was necessary to add an explicit inverse operation, which can be used by the validation subsystem to undo the effects of an execution of the new operation. Conversely, VocBench 3 represents any change in terms of triple additions and deletions together with metadata telling the name of the operation, its arguments, the user who invoked the operation and the timestamp of the change. Furthermore, the undo of a change is performed uniformly by swapping additions and deletions of triples. PoolParty represents the changes in RDF together with the data in a dedicated named graph, by relying on the ChangeSet (Tunnicliffe and Davis, 2005) ontology. As already discussed, Protégé stores the changes in RDF using the ChAO ontology (like PoolParty), but separately from the data (like VocBench).

TopBraid EVN has a flexible architecture, supporting different approaches to validation. Like other systems, TopBraid EVN records individual changes to a dataset as they are performed by users using the Teamwork ontology. Undo of a change consists in the application of its inverse, which is recorded in the history. Although different users can work concurrently on the same workspace, TopBraid EVN supports the creation of multiple working copies. In a typical scenario, different contributors work on different working copies, which can be frozen for review, when it is the time merge them into the (master) production copy. While the history of a working copy can contain overlapping changes (e.g. a property of a resource is first set to a value and then to a different one), the validator performing the merge can obtain a comparison report containing only the actual differences between the production copy and the working copy (like in the non-monitored scenario).

The change tracking capability of these tools is valuable in its own, but it is interesting mostly because it allows a form of

validation. Protégé has followed the path to integrate many communication and coordination facilities, aimed at forming consensus. For instance, it is possible to annotate changes, discuss them and even vote on a change. VocBench, on the other hand, follows a different approach, because communication is performed externally by means of Wikis, issue management systems, etc... Again, TopBraid EVN is hybrid, since it supports some internal discussion mechanisms, while also offering the integration with external systems.

5 Conclusions

We have performed a landscape analysis of RDF version control systems and approaches, focusing on the demands of collaborative and iterative development processes. Under this perspective, the controlled rejection of individual changes is very important, especially in the context of curated datasets, in which proposed changes must undergo an explicit acceptance process.

We observed that change validation is complementary to the need for discrete snapshots of a dataset, and that it can be realised on top of different strategies for version control. We first remarked that in asynchronous workflows validation can be implemented in terms of selective merging of changes into the main development copy of a dataset. Differently, in synchronous workflows changes are already applied to the dataset, therefore validation should be based on an explicit undo mechanism. By shifting the focus of the management from versions to changes, we observed that it is possible to implement this synchronous workflow in a clearer manner. Finally, we observed that this is the path that many collaborative editing environments for RDF have followed.

Acknowledgements

This work was partially funded by the European Commission ISA² programme in the context of the development of VocBench 3 (VB3), which is managed by the Publications Office of the EU under the contract 10632 (Infeurope S.A.).

References

Alexander, K., Cyganiak, R., Hausenblas, M. and Zhao, J. (2011) *Describing Linked Datasets with the VoID Vocabulary (W3C Interest Group Note*). Available online at: http://www.w3.org/TR/void/ (accessed on 16 May 2012).

Arndt, N., Radtke, N. and Martin, M. (2016) 'Distributed collaboration on RDF datasets using git: towards the quit store', *Proceedings of the 12th International Conference on Semantic Systems*, ACM, New York, NY, USA, pp.25–32.

Auer, S. and Herre, H. (2007) 'A versioning and evolution framework for RDF knowledge bases', Virbitskaite, I. and Voronkov, A. (Eds): *Perspectives of Systems Informatics (Lecture Notes in Computer Science)*, Vol. 4378, pp.55–69.

Berners-Lee, T. (2006) *Linked Data.* Available online at: https://www.w3.org/DesignIssues/LinkedData.html

Berners-Lee, T. and Connolly, D. (2001) *Delta: an ontology for the distribution of differences between RDF graphs.* Available online at: https://www.w3.org/DesignIssues/Diff (accessed on 29 March 2016).

Berners-Lee, T. and Connolly, D. (2011) *Notation3 (N3): A readable RDF syntax.* Available online at: https://www.w3.org/TeamSubmission/n3/

Berners-Lee, T., Hendler, J.A. and Lassila, O. (2001) 'The semantic web: a new form of web content that is meaningful to computers will unleash a revolution of new possibilities', *Scientific American*, Vol. 284, No. 5, pp.34–43.

Carroll, J.J., Bizer, C., Hayes, P. and Stickler, P. (2005) 'Named graphs, provenance and trust', *WWW'05: Proceedings of the 14th international conference on World Wide Web*, ACM Press, New York, NY, USA, pp.613–622.

Cassidy, S. and Ballantine, J. (2007) 'Version control for RDF triple stores', in Filipe, J., Shishkov, B. and Helfert, M. (Eds): *ICSOFT 2007, Proceedings of the Second International Conference on Software and Data Technologies, Volume ISDM/EHST/DC*, Barcelona, Spain, 22–25 July, pp.5–12.

Cyganiak, R., Wood, D. and Lanthaler, M. (2014) *RDF 1.1 Concepts and Abstract Syntax.* Available online at: https://www.w3.org/TR/rdf11-concepts/

Fiorelli, M., Pazienza, M.T., Stellato, A. and Turbati, A. (2017) 'Version control and change validation for RDF datasets', in Garoufallou, E., Virkus, S., Siatri, R. and Koutsomiha, D. (Eds): *Metadata and Semantic Research (Communications in Computer and Information Science)*, Vol. 755, Springer, Cham, pp.3–14.

Graube, M., Hensel, S. and Urbas, L. (2014) 'R43ples: revisions for triples – an approach for version control in the semantic web', *Proceedings of the 1st Workshop on Linked Data Quality Co-located with 10th International Conference on Semantic Systems, LDQ@SEMANTiCS 2014*, Leipzig, Germany, 2 September.

Halilaj, L., Grangel-González, I., Coskun, G., Lohmann, S. and Auer, S. (2016) 'Git4Voc: collaborative vocabulary development based on git', *International Journal of Semantic Computing*, Vol. 10, No. 2, pp.167–191.

Heath, T. and Bizer, C. (2011) 'Linked data: evolving the web into a global data space', *Synthesis Lectures on the Semantic Web: Theory and Technology*, Vol. 1, No. 1, pp.1–136.

Hodge, G. (2000) *Systems of Knowledge Organization for Digital Libraries: Beyond Traditional Authority Files*, Council on Library and Information Resources, Washington, DC.

Hogan, A., Arenas, M., Mallea, A. and Polleres, A. (2014) 'Everything you always wanted to know about blank nodes', *Web Semantics: Science, Services and Agents on the World Wide Web*, Vols. 27–28, pp.42–69.

Im, D.-H., Lee, S.-W. and Kim, H.-J. (2012) 'A version management framework for RDF triple stores', *International Journal of Software Engineering and Knowledge Engineering*, Vol. 22, No. 1, pp.85–106.

Kiryakov, A. and Ognyanov, D. (2002) 'Tracking changes in RDF(S) repositories', in Omelayenko, B. and Klein, M. (Eds): *Proceedings of the Workshop on Knowledge Transformation for the Semantic Web KTSW 2002. Workshop W7 at the 15-th European Conference on Artificial Intelligence*, 23 July, Lyon, France, pp.27–35.

Klein, M., Fensel, D., Kiryakov, A. and Ognyanov, D. (2002) 'Ontology versioning and change detection on the web', in Gómez-Pérez, A. and Benjamins, V.R. (Eds): *Knowledge Engineering and Knowledge Management (Lecture Notes in Computer Science)*, Vol. 2473, Springer, Berlin, Heidelberg, pp.197–212.

Noy, N.F., Chugh, A., Liu, W. and Musen, M.A. (2006) 'A framework for ontology evolution in collaborative environments', in Cruz, I., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P. et al. (Eds): *The Semantic Web – ISWC 2006 (Lecture Notes in Computer Science)*, Vol. 4273, Springer, Berlin, Heidelberg, pp.544–558.

Papavassiliou, V., Flouris, G., Fundulaki, I., Kotzinos, D. and Christophides, V. (2009) 'On detecting high-level changes in RDF/S KBs', in Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E. et al. (Eds): *The Semantic Web – ISWC 2009 (Lecture Notes in Computer Science)*, Vol. 5823, Springer, Berlin, Heidelberg, pp.473–488.

Seaborne, A. and Davis, I. (2010) 'Supporting change propagation in RDF', *Proceedings of the W3C Workshop – RDF Next Steps*, 26–27 June, Stanford, Palo Alto, CA, USA.

Shadbolt, N., Berners-Lee, T. and Hall, W. (2006) 'The semantic web revisited',*IEEE Intelligent Systems*, Vol. 21, No. 3, pp.96–101.

Stellato, A., Rajbhandari, S., Turbati, A., Fiorelli, M., Caracciolo, C., Lorenzetti, T. et al. (2015) 'VocBench: a web application for collaborative development of multilingual thesauri', in Gandon, F., Sabou, M., Sack, H., d'Amato, C., Cudré-Mauroux, P. and Zimmermann, A. (Eds): *The Semantic Web. Latest Advances and New Domains (Lecture Notes in Computer Science)*, Vol. 9088, Springer, Cham, pp.38–53.

Stellato, A., Turbati, A., Fiorelli, M., Lorenzetti, T., Costetchi, E., Laaboudi, C. et al. (2017) 'Towards VocBench 3: pushing collaborative development of thesauri and ontologies further beyond', in Mayr, P., Tudhope, D., Golub, K., Wartena, C. and De Luca, E.W. (Eds): *17th European Networked Knowledge Organization Systems (NKOS) Workshop*, Thessaloniki, Greece, 21 September, pp.39–52.

Tudorache, T., Noy, N.F., Tu, S. and Musen, M.A. (2008) 'Supporting collaborative ontology development in protégé', in Sheth, A., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T. et al. (Eds): *The Semantic Web - ISWC 2008 (Lecture Notes in Computer Science)*, Vol. 5318, Springer, Berlin, Heidelberg, pp.17–32.

Tudorache, T., Nyulas, C., Noy, N.F. and Musen, M.A. (2013) 'WebProtégé: a collaborative ontology editor and knowledge acquisition tool for the web', *Semantic Web*, Vol. 4, No. 1, pp.89–99.

Tunnicliffe, S. and Davis, I. (2005) *Changeset.* Available online at: http://vocab.org/changeset/ (accessed on 29 March 2016).

Vander Sande, M., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E. and Van de Walle, R. (2013) 'R&W base: git for triples', in Bizer, C., Heath, T., Berners-Lee, T., Hausenblas, M. and Auer, S. (Eds): *Proceedings of the WWW2013 Workshop on Linked Data on the Web*, Rio de Janeiro, Brazil, 14 May.

Völkel, M. (2006) *D2.3.3.v2 SemVersion – Versioning RDF and Ontologies*, EU-IST Network of Excellence (NoE) IST-2004-507482 KWEB.

Völkel, M. and Groza, T. (2006) 'SemVersion: an RDF-based ontology versioning system', in Isaías, P., Nunes, M.B. and Martínez, I.J. (Eds): *Proceedings of the IADIS International Conference on WWW/Internet*, Murcia, Spain, 5–8 October, pp.195–202.

W3C (2004) *Resource Description Framework (RDF*). Available online at: http://www.w3.org/RDF/

W3C (2008) *SPARQL Query Language for RDF*. Available online at: https://www.w3.org/TR/rdf-sparql-query/ (accessed on 15 November 2017).

W3C (2009) *OWL 2 Web Ontology Language.* Available online at: http://www.w3.org/TR/2009/REC-owl2-overview-20091027/

Wikipedia contributors (2017) *Management Science.* Available online at: https://en.wikipedia.org/wiki/Management_science (accessed on 15 December 2017).

Zaikin, I. and Tuzovsky, A. (2013) 'Owl2vcs: tools for distributed ontology development', in Rodriguez-Muro, M., Jupp, S. and Srinivas, K. (Eds): *Proceedings of the 10th International Workshop on OWL: Experiences and Directions (OWLED 2013) co-located with 10th Extended Semantic Web Conference (ESWC 2013)*, Montpellier, France, 26–27 May.