

Previewing OWL Changes and Refactorings Using a Flexible XML Database

Christoph Lange and Vyacheslav Zholudev

Computer Science, Jacobs University Bremen,
{ch.lange,v.zholudev}@jacobs-university.de

Abstract. During their lifecycle, ontologies are changed for diverse reasons: their vocabulary is enhanced to enable additional application or annotation possibilities, their expressivity is restricted to speed up reasoning, their internal structure is refactored for alignment with other ontologies or to facilitate maintenance, and many more. Any such change can have serious consequences on applications using an ontology; therefore it has to be done with care. TNTBase is a versioned XML database supporting *virtual documents*: XQuery-based views on XML documents that appear to the user as files. We use this feature in order to preview changes to OWL 2 XML ontologies: any proposed change is first tested in a virtual document, before it is applied to the actual ontology. We demonstrate the flexibility of this approach in several cases of changes, discuss the limitations of working with ontologies on XML level, and propose an integration of TNTBase as a backend with ontology editors.

1 Introduction

Change management and refactoring are important parts of the ontology lifecycle. Originally investigated in software engineering, they are now also gaining more and more attention and software support in ontology engineering (see, e. g., [19, 6]), often within the larger context of ontology evolution [25]. Change management for ontologies has been defined as “the process of performing the changes as well as [...] the process of coping with the consequences of changes” [12]. Refactoring in software engineering is commonly defined as “a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior” [7]; this definition is also valid for ontologies. Typical ontology refactorings include splitting an ontology into several modules, or, conversely, merging multiple ontologies, moving axioms to another module of the same ontology, or rewriting axioms to semantically equivalent but shorter or longer forms (e. g., in description logics, rewriting the two axioms $A \sqsubseteq B, A \sqsubseteq C$ into $A \sqsubseteq B \sqcap C$). Despite the strict definition of refactoring, practical development environments, including Protégé, also subsume restructurings that do change the external behavior of code under “refactoring”, e. g. changing the URI of an ontology or entity [6]. However, well-behaved tools usually try to keep the potential damage done by such a change as low as possible by adapting all ontologies in the current project to it. On the other hand, it is not trivial for a

development environment to estimate the full impact of a change, which means that in practice changes often *do* break other things, such as other ontologies based on the current one, ontology-based software implementations, documents annotated with terms from ontologies, etc. Consider the axiom refactoring example mentioned before, and suppose that A , B , and C are not concepts but roles. Not all description logics support role intersection, so certain reasoners might not support the supposedly equivalent rewriting of two role inclusions into one.

Therefore, it is safe to assume that *any* change may break things. When multiple developers collaborate in a shared repository, this means changes should be made with care, and that they should be tested. We present a repository system that supports testing changes by not making them to the physical ontology files in the first place, but by creating them as *views* on the unchanged physical files. We have implemented this on top of our versioned XML database TNTBase for ontologies in the XML encoding of OWL 2. We discuss common change and refactoring cases and show their realization in TNTBase, also taking larger engineering workflows as a part of the ontology lifecycle into consideration. At the same time, this is an experiment in exploring the advantages and limits of managing OWL ontologies on the level of XML document collections.

2 TNTBase, a Versioned Database for XML

TNTBase is a versioned client-server XML database [27]. Essentially, it consists of two parts: the core and the application-specific layer. Let us briefly discuss the technical foundations we need for refactoring OWL 2 XML ontologies.

2.1 The Core

The xSVN module, which integrates Berkeley DB XML [1] into a Subversion server [23], is the core of TNTBase. DB XML stores HEAD revisions of XML files; non-XML content like PDF files or images, differences between revisions, directory entry lists and other repository information are retained in a usual SVN back-end storage. Keeping XML documents in DB XML allows us to access those files not only via any SVN client, but also through the DB XML API that supports efficient querying of XML content via XQuery [2] and modifying it via XQuery Update [3]. As many XML-native databases, DB XML also supports indexing, which improves performance of certain queries.

TNTBase is realized as a web-application with two different communication interfaces: an xSVN interface and a RESTful interface for XML-related tasks. The xSVN interface behaves like the normal SVN interface – Apache’s *mod_dav_svn* module serves requests from SVN clients – with one exception: When an ill-formed XML file is committed, xSVN aborts the whole transaction. The RESTful [11] interface provides XML fragment access to the versioned collection of documents:

Querying: As every XML-native database, TNTBase supports XQuery, but extends the DB XML syntax by a notion of file system paths to address path-based collections of documents.

Modifying: Apart from modifying documents via an SVN client, TNTBase takes advantage of XQuery Update facilities, and, in contrast to pure DB XML, versions modified documents: A new revision is committed to xSVN whenever a documents in a TNTBase repository has been changed.

Querying previous revisions: Although xSVN's DB XML back-end by default holds only the latest revisions of XML documents, and others are stored as reverse deltas against them, it is possible to access and query previous versions by additionally providing a revision number to the TNTBase XQuery extension functions. However, previous versions cannot be modified since once a revision is committed to an xSVN repository, it becomes persistent.

2.2 Application Layer of TNTBase

It turned out that many tasks specific to particular XML formats can be done by TNTBase, and that was a reason to derive a separate layer on top of the TNTBase core and augment this layer with format-specific functionalities. Detailed information can be found in [28], but let us briefly describe the major features:

Virtual Documents are essentially “XML database views” analogous to views in relational databases; these are tables that are virtual in the sense that they are the results of SQL queries computed on demand from the explicitly represented database tables. Similarly, TNTBase virtual documents (VD) are the results of XQueries computed on demand from the XML files explicitly represented in TNTBase, presented to the user as entities (files) in the TNTBase file system. Like views in relational databases, TNTBase VDs are editable, and the TNTBase system transparently patches the differences into the original files in its underlying versioning system. Thus, a user does not have to know about the original source of document parts and it allows him to focus only on relevant pieces of information. Again, like relational database views, VD become very useful abstractions in the interaction with versioned XML storage. Technically, VDs are document templates with embedded references to any number of XQueries and instructions how to fill those templates with XQuery results. For example, a VD for a table of contents may consist of a caption, some author information and XQuery that selects all chapter titles from a collection of documents. For more examples and understanding, check out [29]. Furthermore, VDs can be addressed directly in XQuery by calling a dedicated XQuery TNTBase function. Hence they may seamlessly be used in users' XQueries as well as comprise the content of other VDs.

Validation and Presentation: TNTBase provides facilities to integrate format-specific validation and presentation. Simple examples are XSL transformations and schema validation. But sometimes a user needs even more, e.g. extract RDF information upon commit and cache it or retrieve a human-readably rendered representation of a document in semantics-oriented markup. Utilizing the TNTBase plugin API one can write additional modules and inject them into the application layer. Configuration files are also stored in a TNTBase repository, so a user do not require access to the server. Commit-time behavior is defined by the SVN *tntbase:validate* property that can be

assigned to files as well as to whole directories. Pre-commit or post-commit hooks that are automatically generated take care of processing committed information based on the configuration files. In case of pre-commit processing a corresponding plugin has access to the documents that are about to be committed, and may reject a transaction if the collection of committed documents is format-inconsistent, or clashes with existing documents in the repository. Last but not least, URLs that are used to perform validation or presentation are dynamically changed once configuration files are modified.

Custom XQuery modules: A user can write his own XQuery extensions and store them in the repository. Thus it is not necessary to have modules located in the server's file system or remotely. XQuery modules can be referenced inside repository itself, which is particularly useful if the development of XQuery modules is still in progress.

3 Previewing Changes and Refactorings in TNTBase

As the main contribution made by this paper, we will analyze how a number of common OWL ontology change and refactoring tasks can be realized in TNTBase by means described in section 2. As TNTBase is an XML database, we have to agree on a fixed XML schema for our implementation. We chose OWL 2 XML [15], the straightforward XML encoding of the functional-style syntax [17] of OWL 2, in terms of which the direct semantics is specified [16]. The RDF/XML serialization of OWL is still more widely used but has to sacrifice the elegance of the functional-style syntax to accommodate for restrictions imposed by RDF's graph-based data model. Moreover, RDF/XML is a particularly awkward RDF serialization, as it cannot be validated against an XML schema, and as there are too many alternatives for expressing the same facts (cf. [5]). Committing to the OWL 2 XML serialization is not a conceptual restriction, though, as translations from and to other serializations can be provided independently, e. g. by the OWL API [24]. Thanks to the Subversion-based infrastructure, such a translation can even be integrated almost transparently into TNTBase: A post-commit hook could translate any committed RDF/XML file to OWL XML.

We assume that the ontologies one wants to work with are stored as OWL 2 XML files in a TNTBase repository.¹ The file and directory structure is completely up to the users. We will not *perform* changes in the first place, but *preview* them. Change and refactoring patterns will be implemented as VDs, which can be applied to any existing ontology – an actual physical OWL XML document, or another VD. The latter permits chaining multiple change steps. Our VDs will usually affect relatively small pieces of an OWL XML document, while leaving the rest unchanged. Therefore, their implementation will heavily

¹ We provide an entry point for a sandbox repository at https://tntbase.org/wiki/usecase_ontologies, where you can apply the refactorings presented below, or your own ones, to your ontologies.

rely on the XQuery update facility [3], which allows for concisely expressing changes to XML documents.²

We discuss some typical change and refactoring patterns below. In each case, we give a general description, then describe its implementation for OWL XML, and discuss its impact the current ontology or other ontologies depending on the current one. Changes with a local impact do not affect dependent ontologies, but they may affect the *usage* of the current ontology – e. g. if a syntactic construct is introduced that the reasoner or editor in use does not support. Refactorings in the strict sense of the definition are local – recall the definition in section 1 –, but many changes commonly considered “refactorings” – such as renamings – have non-local effects on dependent ontologies or annotated resources. The larger the impact of a change becomes, the more desirable do we consider evaluating the impact by previewing the change using a VD in TNTBase.

3.1 Renaming Entities

All entities of an OWL ontology – classes, properties, and individuals – are identified by IRIs. Renaming entities is a frequent refactoring task. Such a change affects the ontology O in which an entity is declared, all ontologies importing O , and ultimately all external documents or software using O . The refactoring VD not only has to be applied to O , but to all dependent ontologies. We restrict our investigations to non-distributed repositories and thus to dependencies within the same repository. We have not yet automated the cross-document part of this refactoring; one would have to manually apply the VD to all ontologies in the repository.³

IRIs can be absolute or relative. Relative IRIs are resolved against the base IRI, which defaults to the URI of the ontology document but can be changed using the *xml:base* mechanism. Absolute IRIs can be abbreviated using a *prefix:localname* syntax, where prefixes are defined on the top level of the ontology, e. g. `<Prefix name="foaf" IRI="http://xmlns.com/foaf/0.1/" />`. Renaming such abbreviated IRIs is still quite straightforward, compared to the RDF/XML serializations of OWL, which delegates prefix→IRI mapping to the more involved namespacing mechanism of XML.

3.2 Factoring out Modules

When an ontology grows large and hard to manage, developers often identify modules and factor them out into subontologies. This is, for example, supported by the Module Extraction plugin of the NeOn Toolkit [18]. Common candidates for such subontologies are all sub- or superclasses of a given class, possibly with their instances, with related properties, and other dependent entities. The subontology S factored out should be connected to the original ontology O via an

² The transform functions of XQuery Update, which we use here for clarity, are not yet completely reliable in DB XML. Therefore, our actual implementation explicitly recurses over XML trees, which is less elegant.

³ In a well-formed collection, applying a VD to *all* ontologies will not do any harm.

import. Either direction of the import could make sense: (i) If O is intended to be the main ontology reused by other ontologies and applications, O should import S . Here, O just happens to have a modular structure *internally*, but applications need not know that. (ii) Conversely, O can be designed as a *core ontology*, of which S is a domain-specific refinement. Applications in the respective domain would rather use S , which imports O . We have implemented XQuery functions for selecting certain subontologies, such as the subclasses of a given class, and generic functions that factor out a given subontology. Previewing such a change requires two VDs: one for O – removing S from it and possibly replacing it by an import link –, and one for the new ontology document containing S .

3.3 Merging Modules

The inverse operation to modularization is merging several modules back into one, which can be desirable for deploying an ontology, or for compatibility with tools that do not support modular ontologies. Here, we will only consider the easy case that all modules to be merged are disjoint, i. e. that no two modules declare two different entities with the same IRI. Then, merging reduces to merging axioms and removing import links. Multiple ontology modules spread over different files and directories can be addressed by the *tnt:collection* XQuery function. For instance, a part of a query `collection('/onto*//*.owl')` will address all OWL XML ontologies in the child folders of directories having the `onto` prefix.

3.4 Rewriting Axioms

In section 1, we have seen a case of rewriting axioms in a semantics-preserving way. In the migration to OWL 2, there are more such cases. OWL 2 not only introduces new axiom or construct types that require additional expressivity, but also mere syntactic sugar [9]. A prominent example for that is the disjoint union of classes. In OWL 1 one had to state separately the pairwise disjointness of a group of classes D_1, \dots, D_n , and the equivalence $D = D_1 \sqcup \dots \sqcup D_n$.⁴ The *DisjointUnion* axiom of OWL 2 allows for directly stating that D is the disjoint union of D_1, \dots, D_n . In order to make legacy ontologies from the OWL 1 age more readable, or in order to benefit from performance improvements offered by OWL 2 reasoners, it is thus desirable to rewrite disjoint unions. This is a change with a local impact. The XQuery in listing 1.1 rewrites separate declarations of pairwise disjointness and union equivalence into single disjoint union axioms. Listing 1.2 shows how that XQuery can be used for creating a VD specification. Note that we factored out the query itself to a separate module and only reference it from a specification by providing a repository-scoped URI. In section 4, we will

⁴ In the RDF/XML syntax, which was the most common one for OWL 1, there was not even a shorthand notation for expressing pairwise disjointness of more than two classes. In the following, we will, however, not deal with translations between RDF/XML and OWL XML, but assume that that has been done before by a lower-level tool.

see how to apply this VD specification to a concrete input document (for more information on what VD specifications are and how to use them refer to [26, 29])

Listing 1.1. Creating disjoint union axioms

```

module namespace tntx = "http://tntbase.mathweb.org/ns/ores";
declare function tntx:refactor-disjoint-union(
  $doc as document-node() as document-node() {
  copy $tmp := $doc modify (
    (: find equivalent class declarations with two children :)
    for $equiv in $tmp/owl:Ontology/owl:EquivalentClasses[count(*) eq 2
      and owl:ObjectUnionOf (: ... one of which is a union :)
      and *[not(self::owl:ObjectUnionOf)] (: ... and the other one is something else :)
    let $whole := $equiv/*[not(self::owl:ObjectUnionOf)] (: D :)
    let $parts := $equiv/owl:ObjectUnionOf/* (: D1, ..., Dn :)
    (: look for declarations of pairwise disjointness of D1, ..., Dn :)
    for $disjoint in $equiv/../owl:DisjointClasses[fn:deep-equal(*, $parts)]
    return (
      delete node $disjoint, (: delete the disjointness axiom and replace the ... :)
      replace node $equiv with (: ... equivalence axiom by a disjoint union axiom :)
        <owl:DisjointUnion> { $whole, $parts } </owl:DisjointUnion> )
    ) return $tmp };

```

Listing 1.2. VD specification for disjoint union axioms

```

<tnt:virtualdocument xmlns:tnt="http://tntbase.mathweb.org/ns">
  <tnt:skeleton>
    <Ontology xmlns="http://www.w3.org/2002/07/owl#">
      <tnt:xqinclude>
        <tnt:query name="disj.xq"/>
        <tnt:return><tnt:result/></tnt:return>
      </tnt:xqinclude>
    </Ontology>
  </tnt:skeleton>
  <tnt:query name="disj.xq">
    import module namespace tntx = 'http://tntbase.mathweb.org/ns/ores'
      at 'tntbase:/modules/refactor/disjoint-union.xq';
    tntx:refactor-disjoint-union(tnt:doc($ontology-path))
  </tnt:query>
  <tnt:params><!-- parameter declarations, with default values that can be overridden -->
    <tnt:param name="ontology-path"> <!-- on creating a VD from this specification -->
      <tnt:value>/ontologies/test-ontology.owl</tnt:value>
    </tnt:param>
  </tnt:params>
</tnt:virtualdocument>

```

3.5 Lowering Expressivity

OWL 2 has several sub-profiles [14]. They allow for efficient reasoning, as they correspond to less expressive logics than *SR_{OIQ}*, which covers the full expressivity of OWL 2. There are *SR_{OIQ}* reasoners, but reducing the expressivity of

an ontology may be desirable in order to benefit from a more efficient reasoner. For example, the “QL” profile can be implemented on top of an SQL database. If an existing ontology is more expressive than the desired profile, certain complex axioms and constructs will have to be removed. Among the axiom types that OWL 2 QL does not support, there are declarations of functional, inverse functional, and transitive properties. These are easy to identify, as they are represented by XML elements on the top level of the ontology. Some constructs are not allowed in certain places, such as existential quantification in the subclass position (e.g. $\exists R.C \sqsubseteq D$). Other constructs, such as unions of classes, are completely forbidden. When lowering the expressivity of an ontology, some forbidden axiom types, or axioms containing forbidden constructs have to be stripped entirely, whereas others can be weakened. Stripping is easy to implement using XQuery Update; one would simply delete nodes that satisfy a certain node test. Weakening usually requires adding a number of new axioms to the ontology. For example, the union $\bigsqcup_{i=1}^n C_i$ is the smallest common superclass of C_1, \dots, C_n and can therefore be weakened to a class C with $C_i \sqsubseteq C$ for each $i = 1, \dots, n$. Another forbidden construct is the complement of a class in subclass expressions, e.g. $\neg A \sqsubseteq B$. This can be weakened by introducing a $C \sqsubseteq B$, where $A \sqcap C \sqsubseteq \perp$, i.e. where A and C are disjoint, which is allowed in OWL 2 QL.

3.6 Stripping Axiom Annotations

OWL 2 introduced annotation of axioms, not just of entities of an ontology. Axiom annotations do not yet enjoy wide tool support. For example, Protégé supports them, whereas the NeOn Toolkit doesn't. They are particularly cumbersome to handle when an OWL ontology is represented in RDF (cf. [21]). Thus, we have implemented a change pattern that strips annotations from axioms. Fortunately, all kinds of annotations are easy to handle in the XML syntax. They are simply given as an optional sequence of initial child elements of an axiom:

```
<ClassAssertion>
  <Annotation>
    <AnnotationProperty abbreviatedIRI="rdfs:comment"/>
    <Literal>TNTBase is a database</Literal>
  </Annotation>
  <Class abbreviatedIRI="dbpedia:Database"/>
  <NamedIndividual IRI="http://tntbase.org/tntbase"/>
</ClassAssertion>
```

3.7 Common Query Patterns

We realized that some functions are reused in many ontology refactoring XQueries. Thus, we distilled them into a shared XQuery module that includes common auxiliary functions. The most common ones are: getting the base IRI of a node, or expanding an IRI using the prefix information and the base IRI in the scope (cf. section 3.1). With standard XQuery – without the update facility –, recursing

over an XML tree and returning subtrees with some changes applied would be another recurring pattern, which can be implemented by a recursive function. The most involved function we have implemented so far is equality: Reducing equality of a set of OWL expressions to equality of XML trees (*fn:deep-equal*), as done in listing 1.1, is a bit naïve, as it hardly takes into account the semantics of OWL. For example, in many constructs, such as *DisjointClasses* or *ObjectUnionOf*, the order of class expressions does not matter. Other cases of semantic equivalence of syntactically different expressions are related to IRIs (cf. section 3.1), which can be written absolutely, relatively, or using prefixes. Therefore, we have implemented OWL expression equality as a recursive XQuery function, which defaults to XML tree equality but handles other cases specially.

4 Integration into the Ontology Lifecycle

Over its whole lifecycle, an ontology will change many times, requiring extensive work with VDs. Applying a single change is straightforward. First, to create a VD one just has to give it a name and associate it with a VD specification path. In this step, query parameters can be associated with a VD; for example, the ontology path parameter in listing 1.2 can be overridden to point to the actual ontology the user wants to refactor. Then content retrieval is done by providing a path of a created VD. In XQueries, e. g. in other VDs, the content of a VD can be addressed by the *tnt:vd* function. From a user perspective, getting a VD is the same as getting a usual file. There is one exception, though: VDs can be obtained only via the RESTful interface of TNTBase. In addition, dynamic parts of a VD (i. e. parts that are results of VD XQueries) are editable. When a user modified a VD, he can commit it back using a RESTful method or a special XQuery function *tnt:submit-vd*. All changes will be propagated to the original documents [28]. Another feature of the RESTful interface is *materializing* of VDs – when refactoring is finished and the content of a VD looks right⁵, a user is able to create a refactored ontology as a physical document in the TNTBase repository based on VD content. Alternatively, the possibility of having editable VDs does not force ontology engineers to materialize all changes: Different developers can keep different views on (sub)ontologies and work with them, without having to adapt to a particular structure of the actual physical ontology.

Our approach should scale well to large ontologies. DB XML indexing facilities might tremendously reduce query time. For instance, we experimented with a collection of 2000 documents and ran filtering queries based on attribute and element names. Adding indexes reduced the timing from 30 seconds to 0.5 seconds. Whereas we do not claim that such speed improvement will be achieved for every query, it gives us a better impression how things may scale in TNTBase. Multiple users can read/write to TNTBase simultaneously – TNTBase secures every user action with a transaction and takes care about deadlocks resolution.

⁵ TNTBase itself does not support the collaborative decision making that is required here. We leave that feature to a future integration of TNTBase into a development environment that, e. g., supports argumentation.

5 Related Work

Several ontology editors offer client-side refactoring. Protégé natively supports a few basic refactorings [6], whereas for the NeOn toolkit several more sophisticated refactoring plugins have been developed [18]. In contrast, our solution is, to the best of our knowledge, the only one that supports ontology refactoring in a server-side repository. Once a VD specification has been provided for change pattern, any other user can apply it to any ontology and load the result into his favorite development environment for evaluation and testing. We believe that the overall workflow would benefit from a closer client-server integration. The NeOn toolkit seems the most suitable candidate for an integration of our work, as its underlying Eclipse IDE has a Subversion client built in. We would only have to add support for those features that require interaction with TNTBase’s RESTful interface, such as the creation or materialization of VDs. While we have focused on changes and refactorings, the Evolva framework – implemented as another NeOn plugin – addresses the general challenge of ontology evolution [25]. It validates an ontology after each change, checking for consistency, duplication, and time-related inconsistencies. TNTBase can perform schema validation on materialized VDs; more advanced validation can be done by integrating 3rd-party validation plugins. All in all, Evolva dynamically adapts an ontology to a changed environment. We believe that such an evolution framework could be nicely complemented with our database backend. Finally, there are alternatives to querying OWL 2 XML with XQuery (Update): for example, representing an OWL ontology as RDF, querying it with SPARQL, and changing it using the proposed SPARQL-Update [22]. A clear specification of SPARQL under different entailment regimes, such as OWL, is under way [8], but the implications on SPARQL-Update have not yet been investigated. RDF is on a higher abstraction level than XML, which practically means, for example, that different syntaxes of encoding IRIs (cf. section 3.1), which make a difference in XQuery, do not affect a SPARQL query. But the RDF encoding of OWL is contrived in other aspects: Everything has to be broken down to RDF triples, which introduces artificial complexity for n -ary structures that can be represented in a straightforward way in XML. The new SPARQLAS language, however, supports intuitive query formulation in the OWL functional-style syntax [20], which is internally translated to standard SPARQL. So far, it only covers querying ontologies, though, not updating them. The OWL API [24], the technical basis of Protégé and the NeOn toolkit, does not change ontologies by queries but programmatically by Java methods, but makes it convenient to implement refactoring algorithms, as all OWL constructs are represented as Java objects on a semantic level, abstracting from a concrete serialization. On the other hand, the OWL API has to parse a complete ontology into the memory, whereas Berkeley DB XML, the database underlying TNTBase, does not have to do that, and therefore is more scalable.⁶

⁶ Actually, the OWL API is prepared for ontologies stored in databases (cf. [10]). Out of the box it only provides an in-memory representation, but integrating it with a TNTBase backend would be feasible.

6 Conclusion and Future Work

We have showed how TNTBase, a versioned XML database, supports ontology changes and refactorings. As changes and refactorings can break modules or resources that depend on an ontology, we do not immediately apply them, but create them as views on the original ontology, using TNTBase’s virtual documents. We have showed that several common change patterns can be implemented for OWL XML using XQuery at a reasonable cost – even more so now that we provide a module of commonly needed functions. Thanks to XQuery Update, changes can be written down concisely. Applying them to given ontologies is straightforward as long as the ontologies are implemented in single files; otherwise more work is required, which could be automated in future, though. As OWL XML is a direct XML encoding of the OWL functional-style syntax, in terms of which the direct semantics of OWL has been specified, processing on XML level is surprisingly close to processing it on a higher “semantic” level. However, we initially had to implement some of the OWL semantics in XQuery, such IRI expansion and the somewhat more involved equality of expressions. Now, these are part of a reusable XQuery module, to which we will add further functionality, possibly including functions that compute ontology metrics.

Here, we have focused on the repository management and XML querying features of TNTBase, but TNTBase can actually do more. For different document formats, we have shown how to extend TNTBase by XML→RDF translations that are automatically run when committing an XML file, how to integrate an RDF triple store, and how to serve Linked Data – raw RDF, or RDFa embedded into human-readable XHTML documents (think of ontology documentation) [4, 13]. We consider this a beneficial complement to the OWL change and refactoring functionality here. If the OWL ontologies are in a triple store with a reasoner attached, more sophisticated SPARQL(AS) queries will become possible.

With change patterns and ontologies stored in the same repository, TNTBase enables an agile way of collaborative refactoring, where a development team can not only discuss the outcome of a refactoring step, but also easily improve the change patterns. We will explore the potential of this methodology in further case studies with real-world ontologies. So far, implementing a change pattern involves manual XQuery editing, and instantiating requires manual VD administration in the repository. These tasks could be facilitated for ontology engineers by a closer integration of TNTBase with a client-side development environment. The NeOn toolkit can already access Subversion-compatible repositories, and Protégé has been successfully connected to other kinds of repositories. Therefore, a closer TNTBase integration seems feasible, and would take us closer to the goal of editable ontology repositories.

References

- [1] *Berkeley DB XML*. URL: <http://oracle.com/database/berkeley-db/xml/>.
- [2] *XQuery 1.0: An XML Query Language*. Recommendation. W3C, 2007.

- [3] *XQuery Update Facility 1.0*. Candidate Recommendation. W3C, 2009.
- [4] C. David, M. Kohlhase, C. Lange, F. Rabe, N. Zhiltsov, and V. Zholudev. “Publishing Math Lecture Notes as Linked Data”. In: *ESWC*. 2010. arXiv: 1004.3390.
- [5] I. Davis. *The Sixteen Faces of Eve*. 2005. URL: <http://iandavis.com/blog/2005/09/the-sixteen-faces-of-eve>.
- [6] N. Drummond. *Protege 4.x Menu Actions and Keyboard Shortcuts*. 2009. URL: <http://protegewiki.stanford.edu/index.php?title=Protege4Shortcuts&oldid=6142#Refactor>.
- [7] M. Fowler. *Refactoring Home Page*. URL: <http://refactoring.com>.
- [8] *SPARQL 1.1 entailment regimes*. Working Draft. W3C, 2009.
- [9] *OWL 2: New Features and Rationale*. Recommendation. W3C, 2009.
- [10] M. Horridge and S. Bechhofer. “The OWL API: A Java API for Working with OWL 2 Ontologies”. In: *OWLED*. 2009.
- [11] *JSR 311: JAX-RS: The Java API for RESTful Web Services*. URL: <https://jsr311.dev.java.net/nonav/releases/1.0/index.html>.
- [12] M. Klein. “Change Management for Distributed Ontologies”. PhD thesis. Vrije Universiteit Amsterdam, 2004.
- [13] C. Lange and M. Kohlhase. “A Mathematical Approach to Ontology Authoring and Documentation”. In: *MKM*. Springer, 2009.
- [14] *OWL 2: Profiles*. Recommendation. W3C, 2009.
- [15] *OWL 2: XML Serialization*. Recommendation. W3C, 2009.
- [16] *OWL 2: Direct Semantics*. Recommendation. W3C, 2009.
- [17] *OWL 2: Structural Specification and Functional-Style Syntax*. Recommendation. W3C, 2009.
- [18] *NeOn Toolkit*. URL: <http://neon-toolkit.org>.
- [19] D. A. Ostrowski. “Ontology Refactoring”. In: *IEEE International Conference on Semantic Computing*. IEEE, 2008.
- [20] F. S. Parreiras et al. *SPARQLAS*. URL: <http://code.google.com/p/twouse/wiki/SPARQLAS>.
- [21] *OWL 2: Mapping to RDF Graphs*. Recommendation. W3C, 2009.
- [22] *SPARQL 1.1 Update*. Working Draft. W3C, 2009.
- [23] *Subversion*. URL: <http://subversion.tigris.org/>.
- [24] *The OWL API*. URL: <http://owlapi.sourceforge.net>.
- [25] F. Zablith, M. Sabou, M. d’Aquin, and E. Motta. “Ontology Evolution with Evolva”. In: *ESWC*. 2009.
- [26] V. Zholudev and M. Kohlhase. “Scripting Documents with XQuery: Virtual Documents in TNTBase”. In: *submitted to Balisage: The Markup Conference*. 2010. URL: <http://kwarc.info/kohlhase/papers/balisage10.pdf>.
- [27] V. Zholudev and M. Kohlhase. “TNTBase: a Versioned Storage for XML”. In: *Balisage: The Markup Conference*. Vol. 3. Mulberry, 2009.
- [28] V. Zholudev, M. Kohlhase, and F. Rabe. “A [insert XML Format] Database for [insert cool application]”. In: *XML Prague*. 2010.
- [29] V. Zholudev et al. *TNTBase – Virtual Documents*. 2010. URL: <http://trac.mathweb.org/tntbase/wiki/VD>.