Managing Change:
An Ontology Version Control System

Timothy Redmond[2], Michael Smith[1], Nick Drummond[3], and Tania Tudorache[2]

[1] Clark & Parsia, LLC
[2] Stanford University
[3] University of Manchester

Abstract. In this paper we present the basic requirements and initial design of a system which manages and facilitates changes to an OWL ontology in a multi-editor environment. This system uses a centralized client-server architecture in which the server maintains the current state and full history of all managed ontologies. Clients can access the current ontology version, all historical revisions, and differences between arbitrary revisions, as well as metadata associated with revisions. This system will be used by many other ontology based services, such as incremental reasoning, collaborative ontology development, advanced ontology search, and ontology module extraction. Taken holistically, this network of services will provide a rich environment for the development and management of ontology based information systems.

1 Motivation and Requirements

We need for a system that manages access to a changing ontology. This requirement is experienced by a variety applications with different stakeholders. An illustrative use case is presented below.

A large distributed organization requires integration and alignment of many heterogeneous data sources and information artifacts. They facilitate such integration by employing one or more expressive OWL ontologies that exist in defined relations to data sources, information artifacts, and an enterprise conceptual model. These ontologies, as a critical infrastructure components, have stakeholders throughout the organization and outside its boundaries. Further, they are developed and maintained concurrently by many parties. Individual stakeholders participate in the ontology engineering process in different ways. Some are primarily consumers, but may make detailed edits to areas of the ontologies critical to them. Others are charged with maintaining high-level ontology coherence and use an integrated ontology development environment, such as Protégé-OWL, to collaborate with similar editors in realtime, leveraging tools to maintain a dynamic view of the ontology. All stakeholders rely on the ontologies being available and consistent across the organization.

This use case illustrates a set of requirements:

Client Performance The network is a potential bottleneck of any distributed or client-server system, but the critical work of ontology development is driven by end users *on the client*. The system we propose uses a client-server architecture, but enables the client to productively work even when the network is unavailable or significantly degraded.

Concurrent Editing Multiple users may suggest changes that conflict with one another. E.g., they may modify the axiom in different ways. Such changes must be identified and conflicts resolved, but in a way that cleanly integrates with existing development workflows. We propose a pluggable conflict management mechanism.

Complete Change Tracking To understand an ontology and be effective developers, ontology editors often need to see the history and evolution of an ontology. We propose a system that makes accessing historical ontology revisions easy and the presentation of changes configurable.

Scalability The ontology engineering efforts most in need of a management system are those responsible for the development and curation of the largest ontologies available. In recognition of these stakeholders, we consider scalability as a critical factor in any design decision.

Finally, we will focus on an approach that keeps the design and implementation of the service simple. A complex service is more difficult to maintain for developers, administrators and users. Design decisions have been made to favor simplicity whenever possible.

We proceed by briefly surveying related work in the following section. Section 3 describes the ontology management system we are proposing and is followed by a presentation of applications we anticipate being enhanced by the availability of such a system. Finally, we conclude in Section 5.

2 Related Work

Several ontology engineering environments have been extended to address the requirements of collaborative ontology development. Protégé 3[4] includes a remote API so that a client can connect to a centralized server that manages concurrent access to an OWL ontology. There are several distributed applications that use the Protégé 3 implementation as a base to support ontology sharing. It fails to satisfy the requirements detailed above because it requires network availability and does not focus on change management. Sesame[5] [2] incorporates a client-server architecture for distributed manipulation of RDF graphs and has been used as an implementation platform for collaborative ontology development in TopBraid Composer[6]. Sesame does not address our requirements because it requires network availability, it does not include change management, and it enforces an RDF-centric view of OWL which frustrates its application.

Subversion and git are examples of version control systems (VCSs) that are used to manage source code and other resources for distributed authoring. Such

[4] http://protege.stanford.edu

[5] http://www.openrdf.org/

[6] http://www.topquadrant.com/topbraid/composer/

systems are not adequate for ontologies; because they rely on the text changes between the files, an ontology that has not changed at all in structure may be considered to have significant changes. Approaches to using such systems for OWL ontology management often constrain serialization and toolchain options.

There is related work on ontology change management. [1] motivates the need for RDF graph change management tools and discusses implementation. RDF-Utils[7] includes RDF focused tools analogous to traditional diff and patch. [4] presents a semantic diff algorithm for the description logic $\mathcal{EL}$, and is hence applicable to the OWL 2 EL profile. OWLDiff[8] is a tool which includes syntactic diff and merge functionality, and similar, but limited, semantic functionality based on [4].

3 System Description

This section describes the Ontology Management System, first focusing on a description of the data being managed, then on the architecture components.

3.1 Managed Resources

The managed data element at the core of the Ontology Management System is an OWL ontology. A managed ontology exists in a collection of revisions, each associated with a unique revision identifier.

By slightly simplifying the ontology structure defined in [5], a specific revision of a managed ontology can be described by a name and a collection of axioms. As such, the difference between an ontology at a given revision and the same ontology at an arbitrary revision can be described by a *changeset*, defined as a set of edit operations, each element of which is one of the following:

1. An axiom addition
2. An axiom removal
3. An ontology name change

One can consider the information content of a managed ontology as a directed acyclic graph in which the nodes represent revisions and edges represent the changeset that exists between two revisions. Paths in this graph are a sequence of changesets which transform the ontology from the revision at the origin node to the revision at the terminal node. Paths can be named and managed by the Ontology Management System. By specially naming one path and appending to it as new revisions are created, one can use its terminus to track a "current" view of the managed ontology. By naming arbitrary paths, one can represent alternative views the ontology (just as branches are used in traditional source control management systems). The graph in Figure 1 depicts such a representation.

It is noteworthy that because the changeset between two revisions can be calculated, and similarly a revision can be constructed from another revision and

[7] http://sourceforge.net/projects/knobot/

[8] http://krizik.felk.cvut.cz/km/owldiff

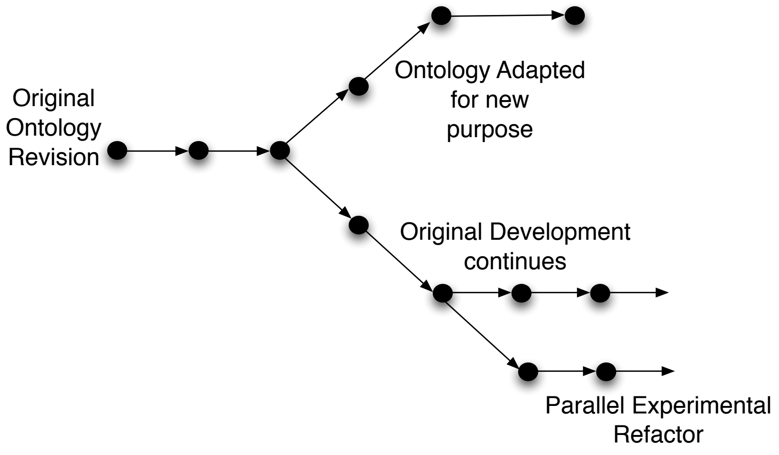

Fig. 1. Example Graph Representation of a Managed Ontology

a changeset, the full information content of a manged ontology can be expressed using revisions exclusively, changesets exclusively, or through some combination of both. This quality is exploited in the system design.

Finally, we extend this conceptualization slightly to allow the association of arbitrary metadata with edges in the graph. This extension is useful to associate editor intent with a changeset, to identify tool specific patterns in changesets[9], and to aggregate related changesets.

3.2 Architecture Components

The system proposed adopts a centralized client server architecture, depicted in Figure 2. A single server is designed to concurrently service multiple clients. This section provides more detail on the architecture by describing the client-server communication and the design of the client and server components.

Client-Server Communication We begin the architecture presentation by discussing the contents of client-server communication in order to provide a context for component designs. All communication in the system is initiated by clients and can be characterized as one of three request types. The request types,

[9] As an example, the NCI EditTab is a custom tool developed for editing large thesauri of terms. It enforces a specific editor workflow and limits the types of operations users can perform in accordance with a role based access control policy. Common edit operations in this tool, such as term “retirement” are specific to the notion of workflow it adopts but can easily be tracked and managed usefully by associating tool specific metadata with changesets.

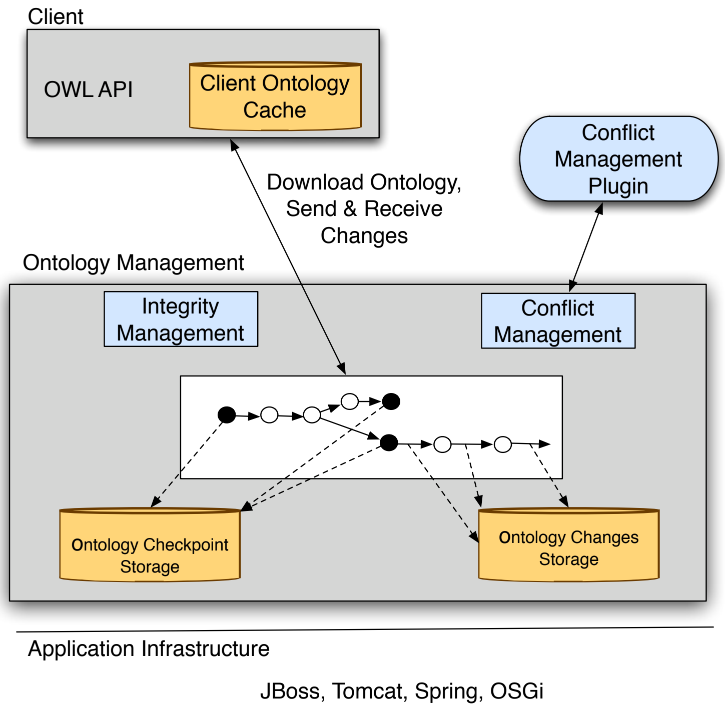
Client
OWL API
Client Ontology Cache
Conflict Management Plugin
Download Ontology, Send & Receive Changes
Ontology Management
Integrity Management
Conflict Management
Ontology Checkpoint Storage
Ontology Changes Storage
Application Infrastructure
JBoss, Tomcat, Spring, OSGi

Fig. 2. Ontology Management System Architecture

two of which are read operations and one of which is a write operation, are listed below.

- GET request for a specific revision
- DIFF request for the changeset between two revisions
- PUT request to create a new revision based on an existing revision and a changeset

Request types GET and DIFF are read only requests. For each, the revision parameters can be identified either by using the unique revision identifier, or by the name of a path, in which case the terminal revision on that path is used. This permits a client to, for example, fetch the latest revision from “trunk” or compare some specific revision with the latest revision from the “refactor branch”. The PUT request is used by a client to modify the managed ontology. It requires a base revision to be identified explicitly by the client, a changeset to be applied to that revision, and the named path on which the changeset should appear. Optionally, additional metadata to be associated with the changeset, may be included. Such requests permit a client to, for example, add a new revision to the “project X branch”. Considered together this collection of request types is designed to emulate behavior common in traditional version control systems.

It is notable that the Put request modifies the managed ontology state using a revision identifier and changeset, not the complete new ontology state. This representation requires a coordination of state between the requesting client and the server components. To increase robustness in this coordination, we anticipate implementing an ontology checksum, i.e., a bit string, much shorter than the typical ontology, that can be derived from the ontology and for which the likelihood of collision between two ontologies is practically zero. For a Put operation, a client would calculate the checksum for the ontology that exists *after* applying the changeset. Thus, the checksum would allow the server to verify that the revision it creates matches the revision intended by the client. The implementation details of the ontology checksum have not yet been chosen. Desiderata include efficient calculation for large ontologies, and incremental update given a changeset. In particular, we anticipate the server component calculating the checksum frequently and want the checksum to increase robustness without significantly impacting performance. The feasibility of choosing a useful checksum that addresses these concerns is being investigated. The use of a checksum in traditional VCS is common, and the implementation of git suggests that if the likelihood of collision is sufficiently low, such a checksum is also useful as a revision identifier.

It is anticipated that most of the calls made by ontology management clients will be Diff and Put requests and note that several existing OWL software frameworks, including the OWL API, already work with data structures similar to changesets.

Finally, we note that the implementation will include additional information in the client requests to support non-core functionality, such as authentication and access control.

Ontology Management Server The server component is the data repository and communication hub of the management system. It is composed of subcomponents responsible for ontology storage, changeset calculation, conflict management, and access control; each subcomponent is discussed below.

Ontology Storage As described in §3.1, for any ontology managed by the ontology management server, the server requires a complete representation of some “root” revision of the ontology and a sufficient collection of changesets to guarantee a path between each revision of the ontology. This data is sufficient to produce the ontology at any revision, as required by Get requests.

This naive approach to storage has several inefficiencies. Most notably, the time required to produce a revision degrades with the number of changesets in the path between the root and target revision. Similarly, if one considers the possibility of data corruption among changesets (e.g., due to hardware failures), dependence on long paths of changesets is undesirable. Clearly, a more effective approach is to maintain intermediate revisions, called *backed revisions* between which shorter paths exist. To produce a specific target revision, the server accesses the backed revision with the shortest path length to the target revision, then traverses the path to the target.

The choice of how many backed revisions should exist is dependent on many factors and optimal tuning requires a multi-dimensional performance analysis that includes consideration of secondary storage capacity and access times, computational burden, and permissible latency when serving client requests. The naive approach is equivalent to maintaining a single backed revision for an ontology. At the opposite extreme, every revision could be a backed revision. We anticipate initial implementation of a heuristic approach in which a backed revision is created every n revisions, where n is a tunable parameter. Should the performance of such an approach prove to be unacceptable, we intend to explore optimization approaches including the use of access patterns to determine the optimal placement of a fixed number of backed revisions within the revision graph. Finally, an administrator may request that a particular revision be backed. This, a type of manual override, may be useful in the case that a particular revision is widely published and has predictably heavy access patterns.

Changeset Calculation To service DIFF requests, the server must calculate the changeset between *any* two revisions. Unsurprisingly, the approaches to changeset calculation are similar to that of storage.

In the naive approach, the server uses the revisions to be compared and performs axiom-by-axiom comparison to produce a changeset. The more sophisticated approach takes advantage of the *mergability* of changesets. A sequence of changesets that constitute a path between two revisions can be merged using algorithm 1 to produce a single changeset. In this algorithm, we use $\mathsf{Add}(.)$ to refer to an axiom addition, $\mathsf{Remove}(.)$ to refer to an axiom removal, and $\mathsf{Name}(.)$ to refer to a name change operation. Each changeset is a collection of these operations. Intuitively, the algorithm merges the changesets by letting add and remove operations “cancel” one another and only includes the final name change. It is notable that behavior for addition of an already present axiom or removal of an absent axiom is undefined. Such a condition should not occur and indicates an error.

Conflict Management Support of the PUT request requires the server to identify conflicting changesets. Simply, if two PUT operations are attempted using the same base revision and path, but different changesets, the possibility for a conflict arises when processing the second operation. It follows that if the base revision referenced in a PUT operation is not the terminus of the referenced path, a conflict may arise if the changeset is not a superset of the changeset between the referenced revision and the terminus. The server requires a conflict management component that identifies conflicts and, optionally, resolves some trivial conflicts. There are several schemes that can be used to identify and deal with conflicts:

- The simplest identification algorithm shifts the responsibility to the client – if a PUT request references a base revision that is not the terminus of the referenced path it is rejected.
- Compare the changeset provided by the client with the changeset between the base revision and the terminal revision and develop a heuristic to refine

Algorithm 1 Merge a sequence of changesets

```
Require: A sequence of changesets, $\boldsymbol{\phi}$ connecting two revisions $r_1$ and $r_2$
Ensure: A single changeset, $\psi$, connecting $r_1$ and $r_2$
$\psi := \emptyset$
for all Changesets $\gamma \in \boldsymbol{\phi}$ do
    for all Edits $\epsilon \in \gamma$ do
        if $\epsilon = \mathsf{Add}(\alpha)$ for some axiom $\alpha$ then
            if $\mathsf{Remove}(\alpha) \in \psi$ then
                $\psi := \psi / \mathsf{Remove}(\alpha)$
            else
                $\psi := \psi \cup \mathsf{Add}(\alpha)$
            end if
        end if
        if $\epsilon = \mathsf{Remove}(\alpha)$ for some axiom $\alpha$ then
            if $\mathsf{Add}(\alpha) \in \psi$ then
                $\psi := \psi / \mathsf{Add}(\alpha)$
            else
                $\psi := \psi \cup \mathsf{Remove}(\alpha)$
            end if
        end if
        if $\epsilon = \mathsf{Name}(\nu)$ for some name $\nu$ then
            for all $\xi$ such that $\mathsf{Name}(\xi) \in \psi$ do
                $\psi := \psi / \mathsf{Name}(\xi)$
            end for
            $\psi := \psi \cup \mathsf{Name}(\nu)$
        end if
    end for
end for
```

the changeset. For example, one might consider independent edits to disconnected parts of an ontology acceptable, but edits involving the same entities unacceptable.

- Implement a locking scheme, such as the one described in [6] and reject all changes that do not abide by the locking rules.

The appropriate mechanism for identification and resolution of conflicts is likely to be site dependent. For this reason, conflict management is a pluggable component of the proposed server. The primary interface to this plug-in is a method that accepts a base revision and two changesets, one provided by the client, one by the change calculation component, and returns an indication of conflict. Optionally, the plug-in may return a refined changeset to be applied at the path terminus.

Access Control Use of the ontology management system in an open, distributed environment motivates the presence of an access control component on the server. Minimally, such a component must accept a client request, augmented with metadata such as client authentication information, and return a permit or deny decision. The appropriate set of access control policies is dependent on the deployment environment. The proliferation of declarative access control languages make it most practical to implement access control as a pluggable component. We anticipate implementing a trivial, “permit all” access control plug-in, a basic user and passphrase based implementation, and, in order to support behavior comparable to what is present in Protégé 3 Server, a role based access control plug-in.

Ontology Management Client We anticipate many applications operating as clients in the ontology management system. The minimal requirements for a client are the ability to submit the request types described above. Initially, we intend to implement a command line tool, modeled on common VCS tools such as `svn` and `git`, that allows user driven interaction and permits ontology developers to use the ontology management system with their existing, unmodified toolchain. Additionally, we plan to implement a plug-in modifying Protégé 4 to include client functionality. We anticipate creating a software implementation (written in Java) that each of these initial implementations uses. Further, several components of this software library are expected to be shared with the server implementation, including the calculation of changesets and ontology checksums. Sharing the conflict management implementation will improve usability by letting clients identify conflicts before submitting requests to the server. Similarly, it will be practical to share elements of the access control implementations.

4 Enhanced Applications

We envision that the Ontology Management System will be useful for a wide variety of distributed applications that use ontologies. The two applications initially motivating this work were source control and collaborative ontology editing. The proposed system addresses their requirements; the question of whether the client polls for changes more frequently, as with a collaborative editing system, or less frequently, as in a source control system is a deployment specific detail.

Another common application is the ontology repository, which should support change management, but often employs ad hoc solutions. We anticipate developing ontology repositories that are based on the proposed system for their data store. Other applications which monitor ontology changes using ontology APIs, such as the OWL API, can be adapted to use the system. This may lead to the incorporation of a variety of unanticipated applications operating in a distributed environment.

In addition to the end-user applications described above, the proposed system will support a variety of infrastructure applications. We anticipate adapting an incremental reasoning service currently in production with Pellet and Protégé 3 Server, adapting existing extraction services to allow clients to access and work with smaller extracted portions of an ontology (see e.g., [3, 4]), and modifying Lucene based ontology search mechanisms that allow cross ontology searches for classes based on phonetic or misspelled search criteria. Each of these applications is already in use and includes an application specific implementation of change management. By reusing the infrastructure provided by the Ontology Management System, each will contain less custom code and overall maintainability will improve.

5 Conclusion

We propose an ontology management system designed to facilitate the development and curation of one or more ontologies among a large number of distributed stakeholders. We have discussed the requirements and design criteria of such a system, contrasted our proposal with existing tools, and enumerated some of the applications that we anticipate benefiting from the implementation of such a system. We believe that such a system is required to make ontology engineering a mature discipline and we view it as a critical component of the maturing OWL infrastructure.

6 Acknowledgments

This work was largely motivated by, and incubated during, a series of discussions on the future development path of the tools used to curate the NCI Thesaurus and related enterprise vocabularies. We thank Gilberto Fragoso, Sherri De Coronado, and Bob Dionne for their valuable input into those discussions and the National Cancer Institute for past and ongoing funding of OWL infrastructure development.

References

1. Tim Berners-Lee and Dan Connolly. Delta: an ontology for the distribution of differences between RDF graphs. Available at http://www.w3.org/DesignIssues/Diff.
2. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. *The Semantic Web-ISWC*, 2342:54–68, 2002.
3. Bernardo Cuenca Grau, Ian Horrocks, Yevgeny Kazakov, and Ulrike Sattler. Modular reuse of ontologies: Theory and practice. *Journal of Artificial Intelligence Research*, 31:273–318, 2008.
4. B. Konev, C. Lutz, D. Walther, and F. Wolter. CEX and MEX: Logical Diff and Semantic Module Extraction in a Fragment of OWL. In *4th OWL Experiences and Directions Workshop (OWLED-2008DC)*.
5. Boris Motik, Peter F. Patel-Schneider, and Ian Horrocks. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. http://www.w3.org/TR/owl2-syntax/, 2008.
6. Julian Seidenberg and Alan L. Rector. A methodology for asynchronous multi-user editing of semantic web ontologies. In Derek H. Sleeman and Ken Barker, editors, *K-CAP*, pages 127–134. ACM, 2007.