Tracking Changes During Ontology Evolution

Natalya F. Noy[1], Sandhya Kunnatur[1], Michel Klein[2], and Mark A. Musen[1]

[1] Stanford Medical Informatics, Stanford University,
251 Campus Drive, x-215, Stanford, CA 94305, USA
{noy, kunnatur, musen}@smi.stanford.edu
[2] Vrije University Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
Michel.Klein@cs.vu.nl

Abstract. As ontology development becomes a collaborative process, developers face the problem of maintaining versions of ontologies akin to maintaining versions of software code or versions of documents in large projects. Traditional versioning systems enable users to compare versions, examine changes, and accept or reject changes. However, while versioning systems usually treat software code and text documents as text files, a versioning system for ontologies must compare and present *structural* changes rather than changes in text representation of ontologies. In this paper, we present the PROMPTDIFF ontology-versioning environment, which address these challenges. PROMPTDIFF includes an efficient version-comparison algorithm that produces a structural diff between ontologies. The results are presented to the users through an intuitive user interface for analyzing the changes that enables users to view concepts and groups of concepts that were added, deleted, and moved, distinguished by their appearance and with direct access to additional information characterizing the change. The users can then act on the changes, accepting or rejecting them. We present results of a pilot user study that demonstrate the effectiveness of the tool for change management. We discuss design principles for an end-to-end ontology-versioning environment and position ontology versioning as a component in a general ontology-management framework.

1 Need for Ontology Versioning

Ontologies constitute an integral and important part of the Semantic Web. For the Semantic Web to succeed, it will require the development and integration of numerous ontologies, from general top-level ontologies, to domain-specific and task-specific ontologies. For this development to happen on the scale that the Semantic Web success requires, ontology development will have to move from the purview of knowledge engineers with artificial intelligence training to the purview of web developers and domain experts. The WYSIWYG HTML editors are in large part responsible for the pervasiveness of today's web, enabling people without formal hypertext training to create web pages and put them on the web. In the same vein, easy-to-use graphical ontology editors that use simple visual metaphors and hide many of the syntactic and semantic complexity behind intuitive interfaces will be an essential component of the success of the Semantic Web.

As ontology development becomes a more ubiquitous and collaborative process, support for *ontology versioning* becomes necessary and essential. This support must enable users to compare versions of ontologies and analyze differences between them.

Furthermore, as ontologies become larger, collaborative development of ontologies becomes more and more common. Ontology designers working in parallel on the same ontology need to maintain and compare different versions, to examine the changes that others have performed, and to accept or reject the changes. In fact, this process is exactly how authors collaborate on editing software code and text documents.

Change tracking in Microsoft Word, while far from perfect, has significantly facilitated collaborative editing of Word documents. Users can easily see the changes between document versions, determine which text was added or deleted, and accept or reject the changes. Intuitive interfaces and tool support must make ontologies just as easy to maintain and evolve as Word documents. Rather than reinvent the wheel, we can build on the successes of tried and tested technologies, adapt them to the wold of ontologies and in fact improve on these technologies using the additional structural knowledge that we have in ontology definitions.

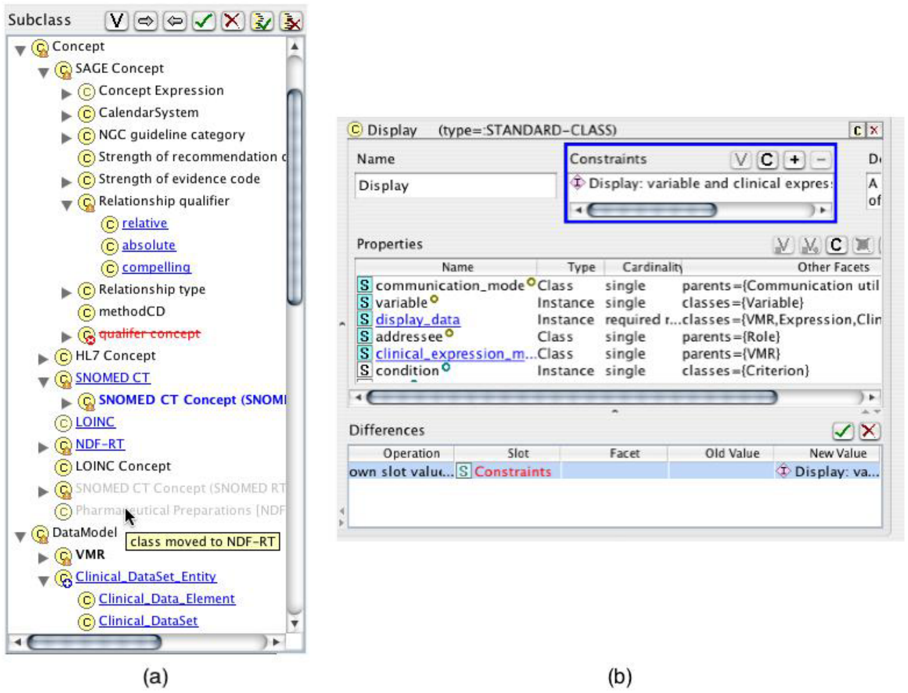
In an ontology-versioning environment, given two versions of an ontology, users must be able to: (1) examine the changes between versions visually; (2) understand the potential effects of changes on applications; and (3) accept or reject changes (when an ontology is being developed in a collaborative setting).

The fields of software evolution and collaborative document processing have faced these challenges for many years. There is one crucial difference however: In the case of software code and documents, what is usually compared—with only a few exceptions—are *text files*. For ontologies, we must compare the *structure* and *semantics* of the ontologies and not their textual serialization. Two ontologies can be exactly the same conceptually, but have very different text representations. For example, their storage syntax may be different. The order in which definitions appear in the text file may be different. A representation language may have several mechanisms for expressing the same semantic structure. Thus, text-file comparison is largely useless for ontologies.

We have developed the PROMPTDIFF ontology-versioning tool to address these issues. PROMPTDIFF automatically performs structural comparison of ontology versions, identifies both simple and complex changes (such as moving classes, or adding or deleting a tree of classes), presents the comparison results to the user in an intuitive way, and enables the user to accept or reject the changes between versions, both at the level of individual changes and groups of changes.

PROMPTDIFF is a component in the larger PROMPT ontology-management framework [13] and serves an important role in maintaining ontology views or mappings between ontologies [3]. PROMPTDIFF provides an ontology-comparison API that other applications can use to determine for example if the mapping needs to be updated when new versions of mapped ontologies appear.

In this paper, we present our general ontology-versioning architecture, describe the user interface for presenting ontology changes; the mechanism for accepting and rejecting changes and ways to custom-tailor this mechanism. We then present the results of our pilot user study aimed at evaluating the usability of our ontology-versioning tools and discuss how PROMPTDIFF is integrated in the general ontology-management framework.



Name	Type	Cardinality	Other Facets
communication_mode	Class	single	parents={Communication util
variable	Instance	single	classes={Variable}
display_data	Instance	required r...	classes={VMR,Expression,Clin
addressee	Class	single	parents={Role}
clinical_expression_m...	Class	single	parents={VMR}
condition	Instance	single	classes={Criterion}

Operation	Slot	Facet	Old Value	New Value
own slot valu...	Constraints			Display: va...

Fig. 1. Displaying changes in PromptDiff. (a) Class hierarchy changes. Different styles represent different types of changes: added classes are underlined and in blue; deleted classes are crossed out and in red; moved classes are grayed out in their old positions and appear in bold in their new ones; tooltips provide additional information. (b) Individual class changes

2 Scenario: Tracking Changes in an Ontology

Consider a project where multiple authors are working on the same ontology.[1] The lead ontology editor receives a new version of the ontology from one of the authors and has the responsibility of examining the changes and accepting or rejecting them. He starts by opening the new ontology version in the Protégé ontology editor [16] and fires up the PromptDiff engine to compare the new ontology version to the baseline. He then examines the results in the PromptDiff interface, a snapshot of which is shown in Figure 1. This figure presents the differences between the two class hierarchies. The editor can see the classes that were added, deleted, moved, and changed.

He starts by focusing in on the subtree for the *Concept* class, which has a number of changes. The color and the font of the class names correspond to the types of the changes. He can see, for example, that *SNOMED CT* is a new class (its class name is in blue and

[1] The ontology versions that we use in this section and in Section 6 are the actual ontology versions from a large collaborative project in our laboratory, the SAGE project (`http://www.sageproject.net/`). The goal of the project is to provide an infrastructure that will enable encoding and dissemination of computable medical knowledge.

underlined). Its subclass was moved to this position from another location in the class hierarchy (it appears in blue and in boldface). In fact, the editor can see the class grayed out in its old location, as a direct subclass of *Concept*. The class *qualifier concept* was deleted (its name is in red and crossed-out). Tooltips provide additional information, such as a new location for the moved class *Pharmaceutical Preparations*.

In addition to changes to individual classes, the editor can see operation on class subtrees. The status of a subtree is conveyed by the icon that is overlaid with the traditional class icon (©). For instance, when the class *qualifier concept* was deleted, all of its subclasses were deleted as well (the class icon has on overlaid delete icon ⊗). The warning icon ⚠ indicates whether or not the subtree rooted at a class has any changes in it. So, the editor knows that there are additional changes in the subtree of the *NDF-RT* class. At the same time, no special icons on the roots of other subtrees indicate that those subtree or individual classes in the subtrees have not changed at all and the editor does not need to look there.

Having examined the changes, the editor can now accept or reject them. He agrees with all the changes in the *Concept* subtree, except for the addition of the class *compelling*, a subclass of *Relationship qualifier*. He starts by selecting the class *compelling* and presses the "reject" button (✕). The class is removed from the new version of the ontology. He then selects the *Concept* class and presses the "accept subtree" button (✓) to accept the rest of the changes in the *Concept* subtree. Once he completes the work on this subtree, he uses the "next" button to move to the next change (in this case, the change to the *VMR* class).

After examining class-hierarchy changes, the editor starts examining changes to specific classes. Figure 1b shows the individual changes for the class *Display*. The editor can see that two properties were added to the class, *display_data* and *clinical_expression_model*. These properties are also in blue and underlined, just like the added classes in the hierarchy. He can see changes to class-level properties, such as *constraints* in the table at the bottom of the class definition. When he selects a line in the table, the corresponding property is highlighted. Again, the editor has the option of accepting or rejecting all changes individually or for the entire class.

In fact, if the editor received the new ontology version from the author that he trusts completely, he can simply select the root of the class hierarchy, and accept all the changes in the entire hierarchy tree with one "accept tree" button.

After the editor examines and acts upon all the changes he can save the result of his work as the new baseline version.

3 Design Principles

The scenario illustrates design principles for an ontology-versioning environment. We collected this set of requirements and design principles through our interaction with users of the Protégé ontology-development environment. These are the capabilities the users asked for in mailing lists and in private communications.

Automatic comparison of versions. Given two versions of the same ontology, we must identify what has changed from one version to the next. This identification should be performed at conceptual level, that is, it should be expressed in terms of changes to ontology concepts, such as classes, instances, their properties, and property constraints and values.

Identification of complex changes. The higher the granularity of changes presented to the user, the easier it is for the user to analyze. For example, if a class was moved to a different place in the class tree, we would like to present this change as a move rather than a sequence of an add and delete operation. Similarly, if a whole subtree of classes was deleted from the ontology, we would like to identify it as a subtree delete rather than a sequence of deletes for all classes in the subtree (and constraints and restrictions associated with them).

Contextual presentation of changes. If the user needs to understand or assess a change in an ontology (for example, a deleted class), he should be able not only to see the change itself, but also to see its context. For example, if a class was deleted, the user may want to know where in the class tree was the class located, whether it had any subclasses or instances, what were its properties, and so on.

Navigation among changes. Changes often occur in different and unrelated places in an ontology. Having examined one of the changes, the user must be able to navigate easily to the next change, even if it is in a "remote" part of the ontology.

Access to old and new values. Understanding and assessing changes is impossible without ready access to both old and new values. Just knowing that the class name has changed and its new name is "foo" is not enough: when examining the change, we would like to know what the old value was.

Mechanism for accepting and rejecting changes. There are many scenarios when users need not only to understand the change but also to make a decision on whether or not the change was correct and to be able to accept or reject the change. Collaborative ontology development is probably the best example of such use. In one use case that we considered—the development of the NCI Thesaurus [4]—managers delegate development of particular portions of the ontology to respective editors. The managers then assemble the new edits, verify them, and have the final word on whether the new edits were correct and should go in the production version.

Different levels of granularity for accepting and rejecting changes. When accepting or rejecting changes, users often do not want to go through each change and mark it for acceptance or rejection. For example, a user may want to say that all the changes in a particular subtree are correct and must be accepted or to reject one of the changes in the subtree and accept all the others.

We tried to follow these design principles in implementing the PromptDiff ontology-versioning environment described in Sections 4 and 5.

4 Presenting Ontology Diff

Figure 2 shows the overall architecture of the PromptDiff ontology-versioning system. Two versions of an ontology, V_1 and V_2, are inputs to the system. The heuristic-based algorithm for comparing ontology versions (the PromptDiff algorithm), which we describe in detail elsewhere [11] and briefly in Section 4.2, analyzes the two versions and

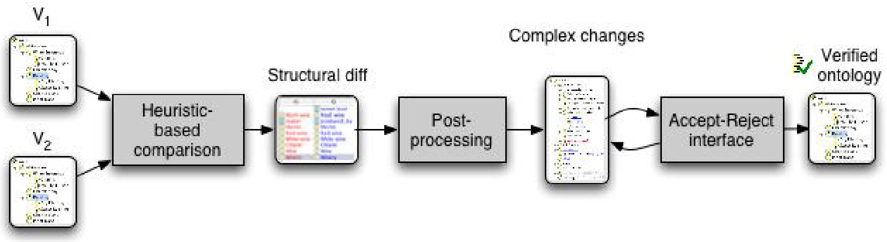


Fig. 2. The architecture of the PROMPTDIFF ontology-versioning system

automatically produces a diff between V_1 and V_2 (we call it *a structural diff* and we define it below). The post-processing module uses the diff to identify complex changes. The results are presented to the user through the intuitive interface (Section 4.3). The user then has the option of accepting or rejecting changes and these actions are reflected in the updated diff (Section 5). We implemented PROMPTDIFF as a Protégé plugin [16].

In this work, we mainly consider ontologies expressed in RDF Schema and therefore consisting of classes, their properties, constraints on properties, property values, and instances [18]. All these elements are also present in other representation formalisms such as OWL and we have used PROMPTDIFF with ontologies specified in OKBC, RDFS, and OWL.

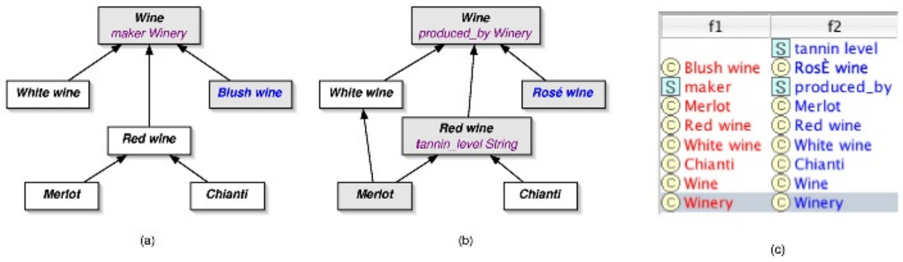
4.1 Structural Diff

We will now introduce the definition of diff between ontology versions. Suppose that we are developing an ontology of wines. In the first version (Figure 3a), there is a class $Wine$ with three subclasses, $Red\ wine$, $White\ wine$, and $Blush\ wine$. The class $Wine$ has a property $maker$ whose values are instances of $Winery$. The class $Red\ wine$ has two subclasses, $Chianti$ and $Merlot$. Figure 3b shows a later version of the same ontology fragment. Note the changes: we changed the name of the $maker$ property to $produced\_by$ and the name of the $Blush\ wine$ class to $Rosé\ wine$; we added a $tannin\ level$ property to $Red\ wine$; and we discovered that $Merlot$ can be white and added another superclass to the $Merlot$ class. Figure 3c shows the differences between the two versions in a table produced automatically by PROMPTDIFF.[2] The columns in the table are pairs of matching frames from the two ontologies. Given two versions of an ontology O, V_1 and V_2, two frames F_1 fromV_1 and F_2 fromV_2 **match** if F_1 became F_2.

Similar to a diff between text files, the table in Figure 3c presents a **structural diff** between ontology versions.

Definition 1 (Structural diff). *Given two versions of an ontology O, V_1 and V_2, a* **structural diff** *between V_1 and V_2, $D(V_1, V_2)$, is a set of concept pairs $\langle C_1, C_2\rangle$ where:*

[2] PROMPTDIFF puts additional information in the table, such as the level of changes, and specific types of changes between a frame and its image [13]



f1	f2
	S tannin level
C Blush wine	C RosÈ wine
S maker	S produced_by
C Merlot	C Merlot
C Red wine	C Red wine
C White wine	C White wine
C Chianti	C Chianti
C Wine	C Wine
C Winery	C Winery

Fig. 3. Two versions of a wine ontology (a and b). Several things have changed: the property name for the $Wine$ class, the name of the $Blush\ wine$ class, a new property was added to the $Red\ wine$ class, the $Merlot$ class has another superclass. (c) The PROMPTDIFF table showing the difference between the versions

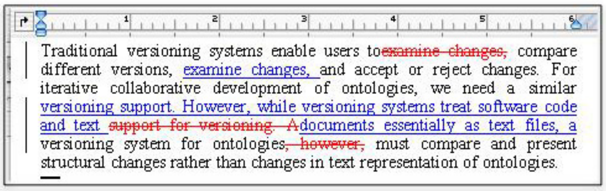
- *$C_1 \in V_1$ or $C_1 = null$; $C_2 \in V_2$ or $C_2 = null$*
- *C_2 is an* **image** *of C_1 (***matches** *C_1), that is, C_1 became C_2. If C_1 or C_2 is null, then we say that C_2 or C_1 respectively does not have a match.*
- *Each frame from V_1 and V_2 appears in at least one pair.*
- *For any frame C_1, if there is at least one pair containing C_1, where $C_2 \neq null$, then there is no pair containing C_1 where $C_2 = null$ (if we found at least one match for C_1, we do not have a pair that says that C_1 is unmatched). The same is true for C_2.*

4.2 The PromptDiff Algorithm

The PROMPTDIFF algorithm for comparing ontology versions consists of two parts: (1) an extensible set of heuristic matchers and (2) a fixed-point algorithm to combine the results of the matchers to produce a structural diff between two versions. Each matcher employs a small number of structural properties of the ontologies to produce matches. The fixed-point step invokes the matchers repeatedly, feeding the results of one matcher into the others, until they produce no more changes in the diff.

Our approach to automating the comparison is based on two observations: (1) When we compare two versions of the same ontology, a large fraction of concepts remains unchanged (in fact, in our experiments, 97.9% of concepts remained unchanged) and (2) If two concepts have the same type (i.e., they are both classes, both properties, etc.) and have the same or very similar name, one is almost certainly an image of the other.

After identifying the concepts that have not changed, PROMPTDIFF invokes a set of heuristic matchers to match the remaining concepts. One matcher for example looks for unmatched classes where all siblings of the class have been matched. If multiple siblings are unmatched, but their sets of properties differ, another matcher will pick up this case and try to match these classes to unmatched subclasses of the parent's image. Another matcher looks for unmatched properties of a class when all other properties of that class have been matched. There are matchers that look for lexical properties such as all unmatched siblings of a class acquiring the same suffix or prefix. The architecture is easily extensible to add new matchers. With the introduction of OWL, for example, we implemented additional matchers that compared anonymous classes.



Traditional versioning systems enable users to~~examine changes,~~ compare different versions, examine changes, and accept or reject changes. For iterative collaborative development of ontologies, we need a similar versioning support. However, while versioning systems treat software code and text ~~support for versioning. A~~documents essentially as text files, a versioning system for ontologies~~, however,~~ must compare and present structural changes rather than changes in text representation of ontologies.

Fig. 4. Comparing text in Microsoft Word: a new document is presented with parts from the old document crossed out and new parts highlighted.

We describe all matchers that we used in detail elsewhere [13]. We have also shown that our algorithm is extremely accurate: it identifies 96% of matches in large ontologies and 93% of the matches that it identifies are correct [11].

The post-processing step enriches the structural diff with the following information: (1) complex changes, such as class moves and tree-level operations; and (2) diffs between individual matching frames

We combine the information in the structural diff with the hierarchical information to identify complex changes. To identify tree-level changes, we recursively traverse the class hierarchy to determine whether all classes in the tree have the same change operation associated with them. To identify class moves, we look for parents of the class and its image to see if they are images of each other. If they are not, then the class was moved.

Since we need this information only when the user views the corresponding part of the ontology, in practice, for efficiency reasons, the post-processing step is intertwined with the visualization step: we find the complex operations only for the trees and classes that the user has displayed on the screen.

4.3 Visualizing Structural Diff

Our user interface for tracking changes between ontology versions was inspired by the interface that Microsoft Word uses to present changes. Figure 4 shows how Microsoft Word presents the result of comparing two versions of a document. The text that was deleted is crossed out and the added text is underlined. There is also color coding for these two types of changes. Note however, that Word does not identify complex operations, such as move. For example, the phrase "examine changes" in the first sentence was moved to a new location. However, Word shows the phrase as being deleted in one place and added in another.

Visual metaphors. We have described many of the visual metaphors we use in PROMPT-DIFF in Section 2. To summarize, we visualize two types of changes in the class hierarchy: (1) class-level changes and (2) tree-level changes.

For class-level changes, the class-name appearance indicates whether the class was added, deleted, moved to a new location, moved from an old location, or its name or definition was changed. The variations in class-name appearance include color, font (bold or regular), names that are underlined or crossed-out.

We say that there is a tree-level change if all classes in a subtree have changed in the same way: they were all added or all deleted, for example. The changed icon at the subtree root indicates a tree-level operation. In addition, a warning icon ⚠ indicates that the subtree rooted at the class has undergone some changes.

We use the same visual metaphors to indicate changes in properties in individual class definitions. For changes in property values, we use a table that shows both the old and the new values for the properties (Figure 1b). When a user selects a line in the table, the corresponding property is highlighted.

Semantics of Parent–Child Relationship in the PromptDiff Tree. Most ontology editors use the indented-tree display to present a class hierarchy to the user. In the tree, children of a tree node representing a class C are subclasses of C. The PromptDiff tree shows two versions of a class hierarchy in a single tree. Consequently, the exact semantics of the parent–child relationship in a tree is different.

Let V_{old} be an old version of the ontology O and V_{new} be a new version of the ontology O. In a PromptDiff tree, if a tree node N represents a class C_{new} from V_{new}, then its children in the PromptDiff tree are:

1. all subclasses of C_{new} in V_{new}; and
2. all subclasses $subC_{old}$ of $source(C_{new})$ in V_{old} such that $image(subC_{old})$ is *not* a subclass of C_{new}.

As a result, the node N representing C_{new} will have children for all subclasses of C_{new} and for all subclasses of its source that were deleted or moved away from the tree.

If a tree node N represents a class C_{old} from V_{old} (this class will necessarily be a deleted or a moved class), then its children in the PromptDiff are determined as follows. For each subclass $subC_{old}$ of C_{old} in V_{old}:

1. if there is an image of $subC_{old}$ in V_{new}, then $image(subC_{old})$ is a child of the node N (it will be shown as moved, i.e., in grey).
2. if there is no image of $subC_{old}$ in V_{new}, then $subC_{old}$ is a child of the node N (it will be shown as deleted).

5 Accepting and Rejecting Changes

When a user examines changes in the new version V_{new} of the ontology compared to a baseline (old) version V_{old}, he may have the need and the authority to accept or reject the changes. In rejecting a change, the user will be effectively changing V_{new}. For instance, if V_{new} contains a new class C and the user decides to reject the class addition, the rejection will have the effect of deleting C from V_{new}.

Accepting a change technically has no implication on the state of V_{new}. It does change the state of the visualization of the diff between V_{old} and V_{new}, since the accepted change should no longer show as a change. For instance, if a user decides to accept the addition of the class C in the example above, the class should no longer show as added (it will now be shown in plain font and without the underline).

As we have shown in the example in Section 2, PROMPTDIFF allows users to accept and reject changes between versions at several levels of granularity:

1. individual changes to property values and definitions
2. all changes in a definition of a class
3. all changes in class definitions and class-hierarchy positions in a subtree rooted at a particular concept

While determining the effects of accepting or rejecting individual operations is relatively straightforward, matters become more complicated as we move from individual changes in properties to class-level changes. Consider for example one of the added classes in the earlier example in Figure 1a, such as $Clinical\_DataSet$. When we accept the addition of the class, do we automatically accept the addition of all of its instances? The same problem arises for an action of rejecting a class delete (such as $qualifier\ concept$). Rejecting a class delete means putting the class back into the new version. Should its instances be reinstated as well? If we reject the addition of the $Clinical\_DataSet\_Entity$ class from Figure 1a, should we also reject the addition of its subclasses and their instances, automatically removing them from the new version?

These issues are similar to the issues discussed by Stojanovic and colleagues [17] in the context of supporting ontology evolution in KAON. When a user deletes a class for example, should its subclasses and instances be deleted as well, or should they become direct subclasses and instances of the parent of the removed class? In the KAON framework, the solution is to enable users to specify evolution strategies to define what happens in each case. However, in the setting of accepting and rejecting changes, the number of strategies to consider is much larger. The examples above show that we must consider not only the status of a single class (added, deleted, moved, etc.) but also various combinations of the class status with the statuses of its subclasses and superclasses.

Our long-term solution to the problem is to classify these strategies and allow users to custom-tailor them. In our current implementation, we implement what we consider to be the most common strategies based on our experience with the users. In our user study (Section 6), all subjects said that the result of accepting and rejecting changes fully met their expectations (all of them gave this question the highest rank of 5). However, since some users may want to custom-tailor this behavior, we are currently working on classifying and identifying the strategies to provide the users with the interface to custom-tailor them.

6 Usability Evaluation

We performed initial user studies to assess the usability of PROMPTDIFF and the ability of users to analyze and verify changes in versions of an ontology. We evaluated both the visualization itself and the mechanism for accepting and rejecting changes. We asked a group of users to compare two versions of an ontology and to accept or reject changes based on a set of rules that we provided.

6.1 Experiment Setup

For the experiment, we presented the subjects with two versions of an ontology describing clinical guidelines and asked them to examine the changes based on a set of provided rules.

Source ontology. We used the actual versions of the clinical-guideline ontology that were recorded in the course of the SAGE project.[3] The ontology contains approximately 400 classes and 300 properties. Between the two versions that we used in the experiment, 3 classes were removed, 2 classes were added, 5 new properties were added and 15 class definitions were changed. In all, there were 47 changes to 20 classes. The changes included moves in the class hierarchy, class name changes, addition of values for class properties, addition of new properties and restrictions.

Rules. We have generated a set of rules for the users to follow in accepting and rejecting changes. The rules covered all the changes that the two ontology versions in the experiment had. Therefore, the experiment was designed to have all users come to the same final version in the end. We made an effort to provide the rules that are formulated around the state of the final version—what should and should not be included—and not in terms of what should be accepted or rejected. Here are some examples of the rules that we gave the users:

- All spelling corrections are ok
- All additions of documentation are ok
- The class $qualifier\ concept$ should not be in the new version
- The $role$ property of $Supplemental\_Material$ class should have the following allowed values: $source, consent\_from, illustration, comment, evidence$

Users. There were 4 users in the study. All of them were experienced users of Protégé and therefore were familiar with knowledge modeling. None of them has used PROMPTDIFF before. Also, none of them was familiar with the SAGE ontology before the experiment.

Training. Prior to the experiment we asked users to follow a brief online tutorial that explained the features of PROMPTDIFF and gave several examples of its use

6.2 Results and Discussion

After we received the resulting ontologies from the users, we compared them with the benchmark ontology that we have prepared. We used PROMPTDIFF to find the changes between the ontologies that the users produced and the benchmark. Our results showed that the users correctly accepted or rejected a change in 93% of the cases. In their comments, users indicated that sometimes they were not sure they were interpreting the change rules correctly and therefore the fact that they did not do what we expected them to do in the remaining 7% of the cases could be partially attributed to that.

On the usability of the tool, the user's satisfaction level was fairly high. We asked the users to rank their answers to a number of usability questions on a scale of 1 to 5 (5 being the highest). Table 1 shows the summary of their responses. In free text comment, users indicated that the navigation through changes (using the next and previous buttons) had bugs in the implementation.

[3] `http://www.sageproject.net/`

Table 1. Results of the user study. The numbers in the second column are the average value for the users' responses. We asked the users to rank their agreement on a scale of 1 to 5, with 5 being the highest.

Was Prompt easy to use?	4
Was the presentation of changes intuitive?	4.25
Was it easy to navigate through changes	3.75
When you performed accept/reject, did the result meet your expectations	5
Was the granularity provided for accepting/rejecting changes satisfactory?	4.5

These results demonstrate that the users in general had high marks for the usability of the tool. The study has also provided us with many useful suggestions that we plan to implement in the future versions of the tool.

7 Ontology Versioning as Part of Ontology-Management Framework

We treat ontology-versioning as one of the tasks in the set of ontology-management tasks. The domain of multiple-ontology management includes not only versioning, but also creating and maintaining mappings between ontologies; merging ontologies; translating between them; creating views of ontologies; and other tasks [12].

Currently, most researchers treat these tasks as completely independent ones and the corresponding tools are independent from one another. There are tools for ontology merging (e.g., Chimaera [8]) and they have no relation to tools for ontology mapping (e.g., ONION [9]) or ontology versioning (e.g., OntoView [6]). Researchers working on developing formalisms for specifying transformation rules from one version of ontology to another, do not apply these rules to related ontologies that are not versions of each other.

However, many of the tasks in management of multiple ontologies are closely interrelated and have common elements and subtasks. Tools for supporting some of the tasks can benefit greatly from their integration with others. For example, the methods that we develop to help users find overlap between ontologies for the tasks of ontology merging can be used successfully in finding differences between ontology versions [11]. In both cases, we have two overlapping ontologies and we need to determine a mapping between their elements. When we compare ontologies from different sources, we concentrate on *similarities*, whereas in version comparison we need to highlight the *differences*, which can be a complementary process. In a previous study, we used heuristics that are similar to the ones we present in this paper to provide suggestions in interactive ontology merging [10].

Looking at the issue from another angle, PROMPTDIFF proved extremely useful in maintaining declarative ontology mappings and views. Consider for example any definition D of a mapping between two ontologies, O_1 and O_2 that explicitly refers to concepts in O_1 and O_2. Suppose we now have a new version of O_1. We must determine if our old mapping is "dirty" and requires re-definition. We say that a declarative mapping between O_1 and O_2 is "dirty" if it explicitly refers to the concepts in O_1 and

O_2 that were changed. Note however that even if the mapping is not "dirty" after a new version of an ontology is introduced, it may still be incomplete or incorrect given the new information.

We use the PROMPTDIFF API to compare the old and the new version of O_1. PROMPTDIFF informs the user not only whether the mapping is "dirty," but also points him to the part of the mapping that refers to the changed concepts and indicates how much the concept has changed.

8 Related Work

Researchers have proposed several directions in supporting ontology versioning (although the field is still fairly nascent). One direction is based on specifying and using explicit logs of changes. For example, Oliver and colleagues [15] specified a set of changes in controlled vocabularies and described mechanisms for synchronizing different versions (with user's input) based on the change logs.

Stojanovich and colleagues [17], in the context of the KAON suite of tools for ontology management, introduce the notion of an *evolution strategy*: allowing developers to specify complex effects of changes. For example, when a class is deleted from an ontology, whether its subclasses are deleted or become subclasses of its superclass, are two different evolution strategy. KAON uses logs of changes at the granularity of single knowledge-base operations as versioning information.

Ognyanov and Kiryakov [14] propose a formal model for tracking changes in RDF repositories. RDF statements are the pieces of knowledge on which they operate. The granularity level of RDF statements is significantly lower however than the level of class and property changes that we consider here.

All of these approaches however rely on using logs or something similar to trace changes between versions. However, both in the context of de-centralized ontology development on the Semantic Web and in the context of collaborative development of ontologies, logs may not be available or do not contain the necessary change information. While many ontology development tools now provide logs of changes, there is no uniform mechanism for publishing the changes along with ontologies. But even if we do have an agreed-upon ontology of changes [7], many versions of ontologies are likely to exist without explicit change information being made public. Thus, we need tools that compare ontology versions themselves to produce the information about changes at the structural level.

The OntoView tool [6] performs a pairwise comparison of the sets of RDF statements that form the old and new version of the class- and property-definitions in an ontology. In this way, changes in syntax and specific representations of RDF are ignored. OntoView focuses on finding and representing changes between concepts in *different* versions. For example, a concept in the new version may become more general than it was in the old version. These types of changes are complementary to the ones we explore in PROMPTDIFF.

The database community has addressed similar problems in the area change-tracking in hierarchical databases and XML documents. In both of these cases researchers must compare tree-like structures, which is part of the ontology-comparison as well. However, most formulations of a tree diff problem are NP-hard [19]. Mainly, this result applies to finding a minimal edit script—the minimal set of changes to transform one tree to another. Therefore, researchers addressed variations of this problem, using heuristics and trading some of the minimality for good performance result [1,2]. The approach resonates with our PROMPTDIFF algorithm described in Section 4: while we cannot provably find the minimal set of changes between ontology versions, our heuristic approach finds the changes efficiently and our experimental results show that in practice it also finds the minimal set of changes [11]. In ontology comparison, however, we can and do also use domain knowledge (e.g., class properties, their domains and ranges) to compare hierarchical class structures—the luxury not directly available in database or XML context.

On the visualization side, visualization of changes in XML trees is the work closest to ours. IBM's XML Diff and Merge Tool [5] is a representative example of these tools. After comparing two XML trees, the tool shows the user the nodes that were added, deleted, or changed. It does not however, identify the moves, or highlight specific changes to a node (It just indicates that a node has been changed and it is up to the user to find what the changes are).

9 Conclusions and Future Work

We have presented a system for tracking changes between ontology versions. Our evaluation demonstrated that PROMPTDIFF is effective in helping users analyze and verify changes.

In order to support fully collaborative ontology development, we need to integrate systems such as PROMPTDIFF with version-control systems, such as CVS, which support other components of the versioning process—tracking versions, checking in and checking out of versions, user profiles, and so on. Other ontology-development support for versioning should include tracking changes directly during edits (rather than determining what the changes are by comparing two versions), enabling users to add design rationale during edits, and allowing users to custom-tailor effects of accepting and rejecting complex changes.

PROMPTDIFF can be downloaded at `http://protege.stanford.edu/plugins/prompt/prompt.html`.

Acknowledgments. This work was supported in part by a contract from the US National Cancer Institute This work was conducted using the Protégé resource, which is supported by grant LM007885 from the United States National Library of Medicine.

References

1. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *ACM SIGMOD Int Conf on Management of Data*. ACM Press, 1997.
2. G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *IEEE International Conference on Data Engineering, ICDE*, San Jose, CA, 2002.
3. M. Crubézy, Z. Pincus, and M. Musen. Mediating knowledge between application components. In *Workshop on Semantic Integration at ISWC-2003*, FL, 2003.

4. J. Golbeck, G. Fragoso, F. Hartel, J. Hendler, B. Parsia, and J. Oberthaler. The NCI's thesaurus and ontology. *Journal of Web Semantics*, 1(1), 2003.
5. IBM. XML diff and merge tool, 2004.
6. M. Klein, A. Kiryakov, D. Ognyanov, and D. Fensel. Ontology versioning and change detection on the web. In *13th Int Conf on Knowledge Engineering and Management (EKAW02)*, Sigüenza, Spain, 2002.
7. M. Klein and N. F. Noy. A component-based framework for ontology evolution. In *Workshop on Ontologies and Distributed Systems at IJCAI-03*, Mexico, 2003.
8. D. L. McGuinness, R. Fikes, J. Rice, and S. Wilder. An environment for merging and testing large ontologies. In *Principles of Knowledge Representation and Reasoning (KR2000)*. Morgan Kaufmann, San Francisco, CA, 2000.
9. P. Mitra, G. Wiederhold, and M. Kersten. A graph-oriented model for articulation of ontology interdependencies. In *Conf on Extending Database Technology (EDBT'2000)*, Germany, 2000.
10. N. Noy and M. Musen. PROMPT: Algorithm and tool for automated ontology merging and alignment. In *17th Nat Conf on Artificial Intelligence (AAAI-2000)*, Austin, TX, 2000.
11. N. F. Noy and M. A. Musen. PromptDiff: A fixed-point algorithm for comparing ontology versions. In *18th Nat Conf on Artificial Intelligence (AAAI-2002)*, Edmonton, Alberta, 2002.
12. N. F. Noy and M. A. Musen. The PROMPT suite: Interactive tools for ontology merging and mapping. *Int Journal of Human-Computer Studies*, 59(6), 2003.
13. N. F. Noy and M. A. Musen. Ontology versioning in an ontology-management framework. *IEEE Intelligent Systems*, page in press, 2004.
14. D. Ognyanov and A. Kiryakov. Tracking changes in RDF(S) repositories. In *13th Int Conf on Knowledge Engineering and Management, EKAW 2002*, Spain, 2002.
15. D. E. Oliver, Y. Shahar, E. H. Shortliffe, and M. A. Musen. Representation of change in controlled medical terminologies. *AI in Medicine*, 15:53–76, 1999.
16. Protege. The Protégé project, http://protege.stanford.edu, 2002.
17. L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. In *13th International Conference on Engineering and Knowledge Management (EKAW02)*, Sigüenza, Spain, 002.
18. W3C. Resource description framework (RDF), 2000.
19. K. Zhang, J. T. L. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs and related problems. In *6th Annual Symposium on Combinatorial Pattern Matching*, pages 395–407, 1995.