A Framework for Ontology Evolution in Collaborative Environments

Natalya F. Noy, Abhita Chugh, William Liu, Mark A. Musen

Stanford University, Stanford, CA 94305
{noy, abhita, wsliu, musen}@stanford.edu

Abstract. With the wider use of ontologies in the Semantic Web and as part of production systems, multiple scenarios for ontology maintenance and evolution are emerging. For example, successive ontology versions can be posted on the (Semantic) Web, with users discovering the new versions serendipitously; ontology-development in a collaborative environment can be synchronous or asynchronous; managers of projects may exercise quality control, examining changes from previous baseline versions and accepting or rejecting them before a new baseline is published, and so on. In this paper, we present different scenarios for ontology maintenance and evolution that we have encountered in our own projects and in those of our collaborators. We define several features that categorize these scenarios. For each scenario, we discuss the high-level tasks that an editing environment must support. We then present a unified comprehensive set of tools to support different scenarios in a single framework, allowing users to switch between different modes easily.

1 Evolution of Ontology Evolution

Acceptance of ontologies as an integral part of knowledge-intensive applications has been growing steadily. The word *ontology* became a recognized substrate in fields outside the computer science, from bioinformatics to intelligence analysis. With such acceptance, came the use of ontologies in industrial systems and active publishing of ontologies on the (Semantic) Web. More and more often, developing an ontology is not a project undertaken by a single person or a small group of people in a research laboratory, but rather it is a large project with numerous participants, who are often geographically distributed, where the resulting ontologies are used in production environments with paying customers counting on robustness and reliability of the system.

The Protégé ontology-development environment[1] has become a widely used tool for developing ontologies, with more than 50,000 registered users. The Protégé group works closely with some of the tool's users and we have a continuous stream of requests from them on the features that they would like to have supported in terms of managing and developing ontologies collaboratively. The configurations for collaborative development differ significantly however. For instance, Perot Systems[2] uses a client–server mode of Protégé with multiple users simultaneously accessing the same copy of the ontology on the server. The NCI Center for Bioinformatics, which develops the NCI The-

[1] http://protege.stanford.edu
[2] http://www.perotsystems.com

saurus[3] has a different configuration: a baseline version of the Thesaurus is published regularly and between the baselines, multiple editors work asynchronously on their own versions. At the end of the cycle, the changes are reconciled. In the OBO project,[4] ontology developers post their ontologies on a sourceforge site, using the sourceforge version-control system to publish successive versions. In addition to specific requirements to support each of these collaboration models, users universally request the ability to annotate their changes, to hold discussions about the changes, to see the change history with respective annotations, and so on.

When developing tool support for all the different modes and tasks in the process of ontology evolution, we started with separate and unrelated sets of Protégé plugins that supported each of the collaborative editing modes. This approach, however, was difficult to maintain; besides, we saw that tools developed for one mode (such as change annotation) will be useful in other modes. Therefore, we have developed a single unified framework that is flexible enough to work in either synchronous or asynchronous mode, in those environments where Protégé and our plugins are used to track changes and in those environments where there is no record of the change steps. At the center of the system is a *Change and Annotation Ontology* (CHAO) with instances recording specific changes and meta-information about them (author, timestamp, annotations, acceptance status, etc.). When Protégé and its change-management plugins are used for ontology editing, these tools create CHAO instances as a side product of the editing process. Otherwise, the CHAO instances are created from a structural diff produced by comparing two versions. The CHAO instances then drive the user interface that displays changes between versions to a user, allows him to accept and reject changes, to view concept history, to generate a new baseline, to publish a history of changes that other applications can use, and so on.

This paper makes the following contributions:

– analysis and categorization of different scenarios for ontology maintenance and evolution and their functional requirements (Section 2)
– development of a comprehensive solution that addresses most of the functional requirements from the different scenarios in a single unified framework (Section 3)
– implementation of the solution as a set of open-source Protégé plugins (Section 4)

2 Ontology-Evolution Scenarios and Tasks

We will now discuss different scenarios for ontology maintenance and evolution, their attributes, and functional requirements.

2.1 Case Studies

We now describe briefly some specific scenarios that we encountered in studying various collaborative projects which members work closely with the Protégé group. Most of these projects focus on developing ontologies in biomedical domain and thus represent scenarios that occur when domain experts develop ontologies collaboratively.

[3] `http://nciterms.nci.nih.gov/NCIBrowser/`
[4] `http://obo.sourceforge.net`

Perot Systems[5] provides technology solutions for organizations. In one of their projects, they are developing an ontology for a Hospital Enterprise Architecture using Protégé. There are almost 100 editors involved in ontology development. These editors use a client–server version of Protégé. The ontology resides on a central server that all editors access remotely. Changes made by one user are immediately visible to everyone. Therefore, there is no separate conflict-resolution stage or maintenance of various archived versions.

Another team with which we work closely is the team developing the NCI Thesaurus at the NCI Center for Bioinformatics [1]. NCI regularly publishes *baseline* versions of the NCI Thesaurus. Users access the Thesaurus through an API or browse it using the NCI's Terminology Browser.[6] Intermediate versions between the baselines are for internal editing only. Currently, each editor edits his own copy of the Thesaurus and checks in his changes regularly. The tools then merge the changes from different editors and identify conflicts. A conflict in this context is any class that was edited by more than one person. A curator then examines the merged ontology and the changes performed by all the editors, resolves the conflicts, and accepts or rejects the changes.

Currently, NCI is switching to producing the NCI Thesaurus directly in OWL and to using Protégé for this purpose. In the new workflow, editors will use the Protégé client–server mode and access the same version of the ontology. Therefore, the merging step will no longer be needed. The curation step will still be present, as the curator still needs to perform quality control and to approve the changes.

The Open Biomedical Ontology repository[7] (OBO) is our final example of an approach to ontology development. OBO is a Web site established by the Gene Ontology Consortium to enable biologists to share their ontologies. The OBO site is a sourceforge site that serves many biological ontologies and vocabularies. Ontology developers post successive versions of their ontologies in the repository, without posting a specific list of changes between the versions. Developers use different tools to edit the ontologies in their local environments (e.g., Protégé, OBO-Edit, Swoop) and create the ontologies in different languages. In many cases, the tools do not create any record of changes and when they do create such a record, the authors do not make it accessible to the outside world when they post their ontology in a repository.

2.2 Attributes of collaborative development

Analyzing various scenarios (like the ones described in Section 2.1), we identified several dimensions that we can use to classify scenarios for collaborative ontology evolution. Depending on the values for each of these dimensions, projects have different functional requirements, which we discuss in Section 2.3

Synchronous vs asynchronous editing In synchronous editing, collaborators work on the *same* version of an ontology that resides on a server accessible to all members of the team. Changes made by one editor are immediately visible to others. Thus, the possibility of conflicts is reduced or, essentially, eliminated since users know what changes others have made to the concept they are editing. With proper transaction

[5] `http://www.perotsystems.com`
[6] `http://nciterms.nci.nih.gov/NCIBrowser`
[7] `http://obo.sourceforge.net`

support, where operations are wrapped in a transaction and two users cannot, for example, overwrite each others' values without knowing about it, conflicts are technically impossible. In asynchronous editing, collaborators check out an ontology or a part of it and edit it off-line. Then they merge their changes into the common version. In this case, conflicts may occur and need to be resolved during the merge.

Continuous editing vs periodic archiving In continuous editing, there is no separate step of archiving a particular version, giving it a name, making it accessible. In this case, the only version of an ontology that exists is the latest one, and any roll-back is performed using an undo operation. Alternatively, versions can be archived periodically, and one can roll back to any of the archived versions.

Curation vs no curation In many centralized environments, new versions of an ontology do not get published externally until a designated curator or curators had a chance to examine all the changes and accept or reject them, and to resolve conflicts. In these scenarios, after editors perform a set of edits, a separate curation step takes place. By its nature, curation almost always happens in the scenarios with periodic versions, as the creation of a new archived version is a natural point for the curation step.

Monitored vs non-monitored In monitored editing, the tools record the changes and, possibly, metadata about these changes, making the declarative description of changes available to other tools. In non-monitored development, the tools either do not log the changes, or these records are not readily available to other tools. For instance, any case where successive versions are published in a repository without change logs being available could be considered non-monitored editing.

2.3 Functional Requirements for Collaborative Ontology Development

We start by discussing the functional requirements that are relevant for all modes of ontology evolution and maintenance. Essentially in any collaborative project users request the following features:

- *Change annotations* that explain the rationale for each change, and refer to citations or Web links that precipitated the change.
- *Change history for a concept* that describes all the changes that were made to the concept, who performed the changes, when, what was the rationale for the change. Programmatic access to such information is particularly important as tools may rely on earlier versions of the concept definition and they must be able to process the changes and adjust to them.
- *Representation of changes* from one version to the next, including information on
  - which classes were changed, added, deleted, split, merged, retired;
  - which elements of each class definitions where added, deleted, changed;
  - who edited each change (possibly simply a list of editors that have altered a class since the previous version; ideally, a more detailed account, with complete trace of how each editor changed the class, with old and new values, etc);
  - which classes have more than one editor.
- Definition of *access privileges* where each author can edit only a particular subset of an ontology, thus avoiding clashes with other editors.

– The ability to *query* an old version using the vocabulary of the new version.
– *A printed summary* of changes and a programmatically accessible record of changes between an old and a new baseline

If ontology editing by multiple authors is **asynchronous**, with each author editing his own copy of the ontology, additional requirements ensure that authors can resolve conflicts afterwards when the different copies are merged back together:

– *Identification of conflicts* between the local version being checked in and the shared version This identification must include treatment of "indirect" conflicts: it is not sufficient to identify concepts that were edited by more than one person. One person may have edited a class A, and another person may have edited its subclass B. While only one person has edited B, there may still be a conflict at B with the changes it inherited from A.
– *Negotiation mechanism to resolve conflicts* if they are caused by someone else's changes. Since check-ins are usually prevented until conflicts are resolved, mechanisms such as chats and emails (integrated with the ontology-development environment) can enable users to resolve conflicts efficiently.

In **continuous editing**, where essentially only a single version of the ontology exists (the current one) and there is no archiving, the main additional requirement is being able to revert back to an earlier version. Since in continuous editing no archives exist, this capability is usually implemented through the undo mechanism and earlier versions are *virtual versions* since they can be generated on demand, but are not explicitly saved. In other words, the following key feature is required for continuous editing:

– An ability to *roll-back* to a particular state of the ontology, for example, to the state of affairs in a particular date.

The following features provide support for **curated editing** by enabling the curator to perform quality control before a new version of an ontology is made public or is used in production:

– A mechanism to *accept and reject* individual changes and groups of changes; groups can be identified structurally (e.g., accept all changes in a subtree) or based on who performed the change and when (e.g., accept all changes from a particular editor);
– Ability to *save the current state of reviewing* (in the middle of the curation process) and to come back to it to continue reviewing the changes;
– *Specialized views of changes*, such as changes from specific editors or views that display only conflicts between editors.

In **non-monitored editing**, there is no explicit record of changes and all that the user has available are two successive versions. Therefore, in this mode there are a number of requirements that deal with comparing the versions and identifying and describing the changes. More specifically, the following features support change management in non-monitored mode:

– *Version comparison* at the structural level, understanding what was added, what was deleted, and what was modified.

– Ability to *post and describe new versions*, specify where the previous version is, and whether the new version is backwards compatible (see, for example, work by Heflin and colleagues on backwards compatibility of ontology versions [4])

3 Components of a comprehensive ontology-evolution system

Figure 1 presents the architecture for an ontology-evolution system that we developed and that provides support for most of the tasks that we have outlined in Section 2. At the center of the system is the *Change and Annotation Ontology* (CHAO) [8]. Instances in this ontology represent changes between two versions of an ontology and user annotations related to these changes. For each change, the ontology describes the following information: its type; the class, property, or instance that was changed; the user who performed the change; the date and time when the change was performed. In addition to change information, we also record annotations on changes. In our implementation (Section 4), we store the annotations as part of the change ontology, but the annotations can be as easily separated into a different ontology with cross-references between the two.[8] Annotations are stored as instances that refer to the change instances (Figure 2). Our representation of annotations extends the Annotea schema.[9] Therefore, the annotations can be saved to an Annotea store and be accessible to applications that process Annotea data.

3.1 CHAO: The Change and Annotations Ontology

Our Change and Annotation Ontology (CHAO) is based on our earlier work [8] and on the work of our colleagues [7]. In this paper, we highlight only the most relevant features of the ontology. We suggest that the interested reader downloads the plugin and examines the ontology that comes with it for specific details.

The ontology contains two main classes, the class `Change` represents changes in the ontology and the class `Annotation` stores related annotations on changes. These two classes are linked by a pair of inverse properties (Figure 2). Subclasses of the `Change` class describe changes at a more fine-grained level. These subclasses include changes to classes such as adding, deleting, or modifying class properties, subclasses, or restrictions; changes to properties; changes to individuals, and changes to a whole ontology, such as creating or deleting classes. There is also a class to group multiple changes into a higher-order change. Changes performed by users are represented as instances of the corresponding subclass of the class `Change`. These instances contain the information describing the change as well as the class, property, or individual to which the change is applied. For example, if a superclass C_{super} is added to a class C_{sub}, we record that a `Superclass_Added` change is applied to the class C_{sub}.

3.2 Generation of the CHAO instances

We must use different mechanisms to generate instances in CHAO, depending on whether editing is monitored or not (Section 2.2). With monitored editing, the tools can gener-

[8] In Figure 1 we show the two parts—changes and annotations—separately.

[9] `http://www.w3.org/2001/Annotea`

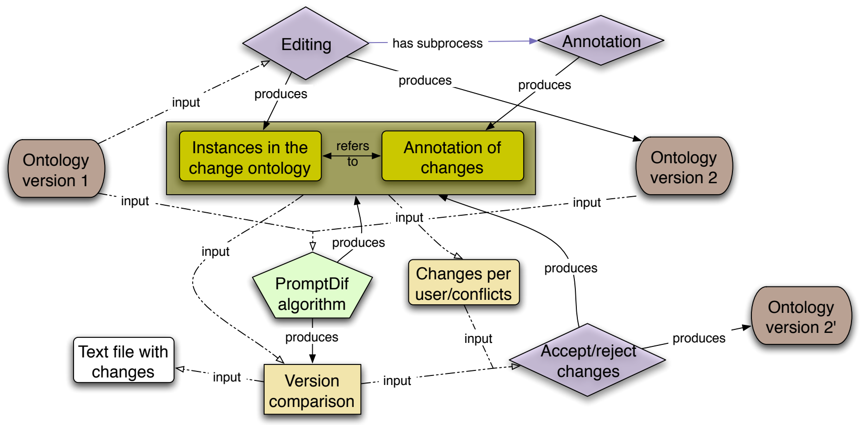


Fig. 1. Components of the ontology-evolution framework. The rounded rectangles represent versions of an ontology. The diamonds represent processes performed by a user. Through the Editing and Annotation processes, an ontology version 1 becomes ontology version 2. The CHAO instances are created as by-product of the editing process. If the CHAO instances are not present, the two versions themselves serve as input to the PROMPTDIFF algorithm that creates a version comparison, from which CHAO instances are generated. The CHAO instances and author, timestamp, and annotation information contained therein are used in the process of accepting and rejecting changes. Each accept or reject operation is recorded as a Boolean flag on the corresponding change instance.

ate CHAO instances as a by-product of the ontology-editing process. Therefore, when users look at a new version of an ontology, they (or the tools they use) also have access to the CHAO instances describing all the changes, and the corresponding annotations.

In many cases, however, particularly in the de-centralized environment of the Semantic Web, tools have access only to successive versions of an ontology, and not to the description of changes from one version to another. In this case, we need to compare the two versions first in order to understand what the differences are. We then can save the differences as instances in CHAO. Naturally, we will not have specific author and timestamp information for each change.

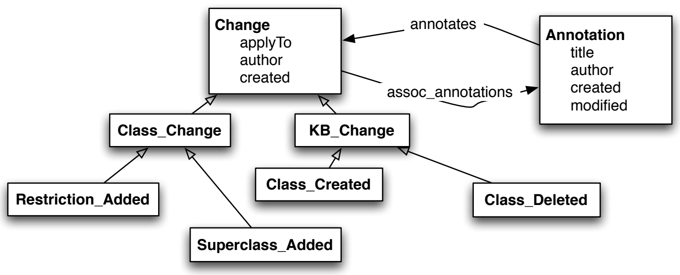


Fig. 2. The `Change` and `Annotation` classes in the change and annotations ontologies, a selection of their properties, and some examples of subclasses of `Change`.

3.3 Using CHAO instances in ontology-evolution tasks

The instances in CHAO provide input for many of the tasks that support functional requirements that we identified in Section 2.3.

Examining changes Many of the scenarios that we discuss involve having a user examine changes between versions. Users need to understand what changed between versions and what changes others have performed. The task is similar to tracking changes in Microsoft Word. The instances in CHAO provide data that inform the display of changes. Some examples of such a display include a table that lists concepts that have been changed and what has been changed for each of the concepts and a tree of classes with deleted, added, or moved classes marked in different fonts and colors and, possibly, underlined or crossed-out (see Figure 3).

Accepting and rejecting changes In curated ontology-evolution in particular, but also in other modes, one may want not only to examine changes between two versions, but also to accept or reject these changes (not unlike accepting or rejecting changes in a Word document). We also need to save the state of the reviewing process (i.e, what has been accepted or rejected) so that the user may come back to it. We use CHAO to record the state of accept/reject decisions if the user wants to save a session and come back to it later. Consider a non-monitored editing where CHAO instances are generated by comparing the old and the new version of the ontology. Note that accepting a change does not involve any actual change to the new version, it is just a decision approving a change that has already been made. Therefore, if we start a new session and compare the two versions again (the old one and the version with which the editor had been working), the accepted change will appear as a change again. To avoid this behavior, any time a user accepts or rejects a change, we record the decision in the corresponding instance of CHAO. So, for instance, a change instance can be flagged as "accepted." A rejection of a change is an actual change, and we record this as another instance in CHAO. The link to the annotation instances enables the user to put additional annotations that, for example, explain his acceptance or rejection decision.

Viewing concept history For each change, CHAO contains the information on the concept to which the change was applied and author and timestamp of the change (if the editing was monitored). We can readily process this information to provide concept history. Thus, for each concept, we can present the history of its changes, who performed them and when. Because CHAO links the changes to the annotations where users can describe rationale for and make other comments about each change or a group of changes, we can also provide annotations in the concept history. As a result, the user can see not only what changes where made to the concept, but also the explanation of why the changes were made, and, if available, the discussion about these changes.

Providing auditing information We can compile the information about authors and timestamps for the changes to create auditing information for a curator. For instance, a curator can see which editors have performed changes in a particular time period, how many concepts each editor has edited, how many concepts where edited by more than one editor in a particular time period.

The use of an ontology to record changes and annotations enables us to have a modular system that supports multiple tasks. For instance, regardless of how the CHAO instance are generated (as a by-product of monitored editing or through version com-

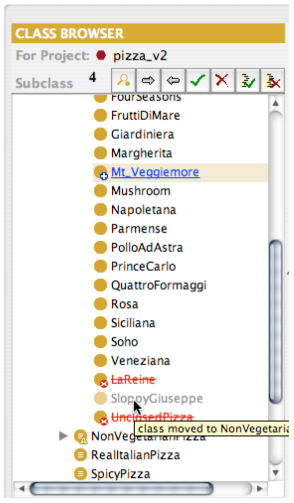


Fig. 3. Comparison of versions in PROMPTDIFF. Classes that were added are underlined; deleted classes are crossed out; moved classes are grayed out.

parison in non-monitored editing), once we have it, we can use it in other tasks, such as presenting concept history, displaying changes between versions, and so on. When the user accepts or rejects a change, we can again use the same set of structures to record his decisions and corresponding annotations. As a result, when a user, for instance, accepts a change to a concept C and records his rationale for this decision, this comment appears in the concept history for C. Furthermore, the use of an ontology to record changes and annotations can potentially enable sharing of this information between ontology-editing tools.

4 Implementation details

We have implemented a system to support the different modes of ontology evolution in a single comprehensive framework that we described in Section 3 as a set of plugins to the Protégé ontology-development environment. The framework is implemented as two related Protégé plugins:

The Change-management plugin provides access to a list of changes and enables users to add annotations to individual changes or groups of changes; when this plugin is activated, the changes are stored as instances in CHAO. Further, this plugin enables users to see a concept history for the class and the corresponding annotations, when they are examining the classes in the standard Classes tab.

The PROMPT **plugin** for ontology management provides comparisons of two versions of an ontology and facilities to examine a list of users who performed changes and to accept and reject changes [9].

In addition, the Protégé environment itself provides many of the facilities necessary to support the scenarios we outlined in Section 2, such as synchronous editing in a client–server mode, transaction support, undo facilities, and other features. The Protégé API provides convenient access to changes as they are performed, to facilities to create and edit ontologies, and to the user interface components that the plugins use.

We now describe these components and where they fall in the overall framework.

4.1 The Change-management plugin

The Change-management plugin to Protégé performs several functions. First, when the user enables the plugin, each ontology change is recorded as an instance in CHAO with the timestamp and the author of the change. Thus, with this plugin in place, the editing is monitored and a declarative record of changes is stored. CHAO instances are saved when the user saves the project. If needed, users can open CHAO in Protégé or access it through the Protégé knowledge-base API, just as they would any other ontology.

Second, users can see the "Changes" tab that provides an overview of changes and corresponding annotations (Figure 4). There are two views for the changes: the "detailed" view shows low-level changes that correspond to each operation in the ontology. The "summary" view (the default) groups the low-level changes into higher-level ones that roughly correspond to the operations that the user performs in the user interface. For instance, when a user creates an OWL restriction for a class, a series of low-level operations happen: an anonymous class is created, property values for this class are assigned, and this class becomes a superclass for the class that it restricts. This set of operations corresponds to a single high-level operation "Restriction added."

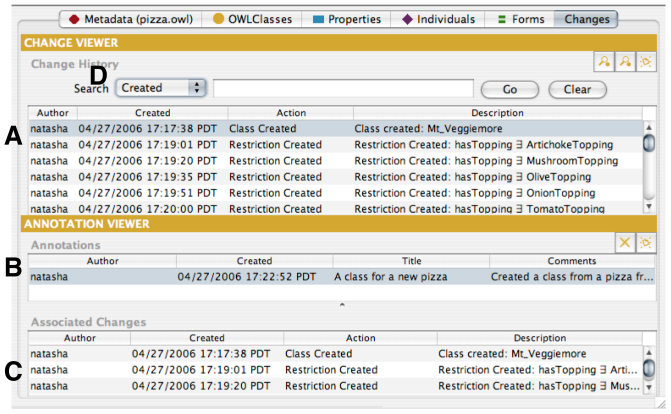
The user can select a change and view annotations for this change and any of the groups of changes that involve the selected change. The user can also provide annotations for the selected change or select a group of changes and provide annotation for the whole group. For example, a user can describe the rationale for adding a set of classes. The tab also provides search capabilities.

In the Classes tab (the standard Protégé tab for browsing and editing classes), an additional menu item—"Change info"—appears when the Change-management plugin is activated. Through this menu, the user can view the concept history and the corresponding annotations for the class selected in the Classes tab. The display is similar to the change and annotation display in Figure 4, but the list of changes is limited to the changes that pertain to the class of interest. The user can also create additional annotations here.

4.2 The PROMPT tab

PROMPT is a Protégé plugin that provides a number of ontology-management functions. Here we discuss only the functionality that is relevant for ontology evolution.

When the user activates the PROMPT tab and instructs it to compare two versions of an ontology, one of the two things happens: (1) If instances of CHAO are present (the editing was monitored), PROMPT uses these instances to compile the changes and to present them to the user. (2) If there are no CHAO instances and the user has only the two version of the ontology, and no record of the changes between them, the PROMPT-DIFF algorithm compares the two versions and creates a *structural diff*, using a set of heuristics to determine what has changed between the versions. It determines, for example, when classes are renamed, when classes are moved in the class hierarchy, or when



Author	Created	Action	Description
natasha	04/27/2006 17:17:38 PDT	Class Created	Class created: Mt_Veggiemore
natasha	04/27/2006 17:19:01 PDT	Restriction Created	Restriction Created: hasTopping ∃ ArtichokeTopping
natasha	04/27/2006 17:19:20 PDT	Restriction Created	Restriction Created: hasTopping ∃ MushroomTopping
natasha	04/27/2006 17:19:35 PDT	Restriction Created	Restriction Created: hasTopping ∃ OliveTopping
natasha	04/27/2006 17:19:51 PDT	Restriction Created	Restriction Created: hasTopping ∃ OnionTopping
natasha	04/27/2006 17:20:00 PDT	Restriction Created	Restriction Created: hasTopping ∃ TomatoTopping

Author	Created	Title	Comments
natasha	04/27/2006 17:22:52 PDT	A class for a new pizza	Created a class from a pizza fr...

Author	Created	Action	Description
natasha	04/27/2006 17:17:38 PDT	Class Created	Class created: Mt_Veggiemore
natasha	04/27/2006 17:19:01 PDT	Restriction Created	Restriction Created: hasTopping ∃ Arti...
natasha	04/27/2006 17:19:20 PDT	Restriction Created	Restriction Created: hasTopping ∃ Mus...

Fig. 4. The Change-management tab in Protégé. The table at the top (A) displays all the changes to the current project. When the user selects a change in the top table, the middle table(B) shows all annotations associated with this change. For selected annotation, the bottom table (C) shows other changes in the group if annotation applied to a group of changes rather than a single change. The user can create new annotations, examine the details of existing ones through this display, and search through annotations (D).

their definitions change. Recall that we designed the architecture for ontology evolution to have CHAO drive all other components in order to have a modularized structure and to have the same components work in different editing modes. Thus, if there is no declarative record of changes in the form of CHAO, PROMPT generates these instances from the results of the PROMPTDIFF algorithm.

One of the unique features of version comparison in PROMPT, is the ability to examine changes using an intuitive interface, based on the Protégé interface for class editing (Figure 3), and to accept and reject changes. In the class hierarchy, the users can see which classes and subtrees were deleted or added, which were changed, moved, and so on. For each class, PROMPT provides a list of changes for that class. Users can also view the old and the new definitions of the class side-by-side. Users then can accept or reject each specific change, all changes for a class, or all changes for a particular subtree.

If CHAO exists, PROMPT also generates a list of users who changed the ontology. For each user, PROMPT displays the number of concepts he created or modified, and the number of concepts that are in conflict with modification performed by others. Here we define a conflict (something that a curator might want to verify) as one concept modified by more than one user. Curators can also accept or reject all changes performed by a specific user in a single click or all changes that are not in conflict with others.

4.3 Client–Server mode for ontology editing in Protégé

Of the many features of the Protégé core system that support ontology evolution, we focus on the one that specifically addresses one of the modes of ontology editing: synchronous editing by multiple users. The multi-user mode in Protégé uses a client–server architecture, with the ontologies being stored on the machine running the Protégé server. Users then remotely connect to this server using a thick Protégé client. All users editing an ontology always see the same version of it, and changes made by one user are visible immediately to other users. If the Change-management plugin is installed on the server, it records all the changes that happen on the server machine made by different users.

Most of the features described in this Section are already available for beta-testing at the time of this writing as part of the Protégé beta distribution. A small number of features, such as saving to Annotea, are currently in alpha-testing or under development.

5 Related Work

We have developed initial ideas on how a change ontology can drive various ontology-evolution tasks in our earlier work [8]. We have borrowed many ideas on the CHAO structure from there, but extended it with additional pragmatic information such as identification of a concept to which the change applies, author and timestamp of the change, and link to relevant annotations. Thus, in this paper, we have not focused on the representation of changes per se using the detailed work by others in this area [7].

Many ontology-development tools support monitored ontology evolution by recording change logs. However, in most cases, the changes are recorded as sequences of changes rather than a set of instances in a change ontology [6, 11]. While one form of representation can be inferred from the other and vice versa, representation in the form of ontology instances facilitates querying for concept histories and enables attaching annotations to changes.

The SWOOP ontology editor [6] supports an extensive set of annotations, distinguishing between comments, advices, examples, and so on. We currently do not enable users to categorize annotations in different ways, but plan to add such capability in the future. As a practical matter, since our annotation ontology already extends the Annotea RDF Schema, we need to develop only the user interface for such support.

The ontology-evolution support in KAON [13, 3] focuses on the effects of changes. Users can specify their desired strategies for ontology evolution in terms of handling of changes. Recent work [3] focuses on maintaining consistency during evolution for DL-based ontologies. The authors consider various scenarios, such as maintaining consistency as the ontology changes, repairing inconsistencies, or answering queries over inconsistent ontologies.

There is one aspect of ontology evolution and versioning that we do not currently deal with in this work—effects of changes on applications. This analysis requires an understanding of which version of an ontology an application is compatible with, in particular in the context of ongoing ontology development where newer versions may not be consistent. The issue of compatibility is addressed in the MORE framework for reasoning with multi-version ontologies [5] and in the work of Heflin and colleagues on declarative specifications of compatibility between versions [4].

6 Discussion

In summary, the two plugins—the Change-management plugin and the PROMPT plugin—in combination with Protégé's own facilities, provide support for all modes of ontology evolution that we have identified among our projects.

Because Protégé can import ontologies developed by other tools (in OWL, RDFS, and a number of other formats), developers can benefit from the Protégé change-management facilities even if they first develop their ontologies in a different tool. For example, if an OWL ontology was originally developed in SWOOP, the developer can still use PROMPT to compare its versions, accept or reject changes, create a new baseline, and then continue editing in SWOOP.

While we have not yet performed a summative evaluation of our tools, we have run formative evaluations with the developers of NCI Thesaurus. In these evaluations, five NCI Thesaurus editors and a curator used Protégé and some of the tools we described here to edit portions of the Thesaurus in parallel with their regular environment (Apelon's TDE). Mainly, the curator used the PROMPT plugin to examine the changes performed by the editors and to accept or reject them. We also used the PROMPT representation of changes to generate the concept-history output in a database format that the APIs for accessing the NCI Thesaurus expect. Many of the features we discussed here (e.g., side-by-side views of the old and the new version of the class, the ability to save the status of the accept/reject process, the ability to annotate changes and curation decisions, etc.) resulted from usability studies during these evaluations. As the result of the studies, NCI is moving towards switching to Protégé and its change-management plugins as their primary editing environment for NCI Thesaurus.

There are several limitations of our current work that we plan to address in the future. First, we currently assume that when there is a record of changes between two versions, this record is *complete*. In many cases, however, this record will not be complete. For instance, a user may disable the change-management support, perform edits, and then enable it again or he may edit a portion of the ontology in a different tool and then import it back into Protégé. We need to address two questions to deal with incomplete change records: (1) how do we determine automatically that a record is indeed incomplete; and (2) in the case of incomplete change record, can we have a hybrid solution that uses the author, date, annotation, and other pertinent information that is available in the incomplete record and combines it with the record generated by PROMPTDIFF.

Migration of instances from one version of an ontology to the next is a critical issue in the context of evolving ontologies. Some ontology changes, such as creating new classes or properties, do not affect instances; when these changes occur, instances that were valid in the old version are still valid ini the new version. However, a large number of changes may potentially affect instances. In this latter case, some strategies can include having tools take their best guess as to how instances should be transformed, allowing users to specify what to do for a specific class of changes (e.g., similar to evolution strategies [13]), or flagging instances that might be invalidated by changes.

Ontology modularization [12, 2] is also critical for support of ontology evolution, in particular in the asynchronous mode. It will be impractical if the whole ontology is the only unit that users can check out. Rather, editing in asynchronous mode would be much more effective if ontology consists of well-delineated modules that cross-reference one another. Users must be able to check out a specific module rather than a whole ontology.

Finally, consistency checking and ontology debugging [14, 10], while important for ontology development in general, are particularly critical in the collaborative setting. Users must be able to understand what the effects changes performed by others have, to understand the rationale for those changes, and to check consistency of the ontology when all the changes are brought together.

We continue to work with our collaborators that use Protégé in large collaborative ontology-development projects to identify new requirements and modes of collaboration. As it exists today, however, we believe that the environment that we described in this paper is one of the most complete sets of components to support ontology evolution today. And the framework that underlies our implementation provides for flexible and modular development of ontology-evolution support.

Acknowledgments

This work was supported in part by a contract from the U.S. National Cancer Institute. Protégé is a national resource supported by grant LM007885 from the United States National Library of Medicine.

References

1. G. Fragoso, S. de Coronado, M. Haber, F. Hartel, and L. Wright. Overview and utilization of the nci thesaurus. *Comparative and Functional Genomics*, 5(8):648–654, 2004.
2. B. C. Grau, B. Parsia, and E. Sirin. Working with multiple ontologies on the semantic web. In *Third Internatonal Semantic Web Conference (ISWC2004)*, 2004.
3. P. Haase, F. van Harmelen, Z. Huang, H. Stuckenschmidt, and Y. Sure. A framework for handling inconsistency in changing ontologies. In *Fourth International Semantic Web Conference (ISWC2005)*, 2005.
4. J. Heflin and Z. Pan. A model theoretic semantics for ontology versioning. In *Third International Semantic Web Conference*, page 6276. Springer, 2004.
5. Z. Huang and H. Stuckenschmidt. Reasoning with multiversion ontologies: a temporal logic approach. In *Fourth International Semantic Web Conference (ISWC2005)*, 2005.
6. A. Kalyanpur, B. Parsia, E. Sirin, B. Cuenca-Grau, and J. Hendler. SWOOP: A web ontology editing browser. *Journal of Web Semantics*, 2005.
7. M. Klein. *Change Management for Distributed Ontologies*. PhD thesis, Vrije Universiteit Amsterdam, 2004.
8. M. Klein and N. F. Noy. A component-based framework for ontology evolution. In *Workshop on Ontologies and Distributed Systems at IJCAI-03*, Acapulco, Mexico, 2003.
9. N. F. Noy and M. A. Musen. The PROMPT suite: Interactive tools for ontology merging and mapping. *International Journal of Human-Computer Studies*, 59(6):983–1024, 2003.
10. B. Parsia, E. Sirin, and A. Kalyanpur. Debugging OWL ontologies. In *14th Intl Conference on World Wide Web*, pages 633–640, New York, NY, 2005. ACM Press.
11. P. Plessers and O. De Troyer. Ontology change detection using a version log. In *Fourth International Semantic Web Conference (ISWC2005)*, 2005.
12. J. Seidenberg and A. Rector. Web ontology segmentation: Analysis, classification and use. In *15th International World Wide Web Conference*, Edinburgh, Scotland, 2006.
13. L. Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, University of Karlsruhe, 2004.
14. H. Wang, M. Horridge, A. Rector, N. Drummond, and J. Seidenberg. Debugging OWL-DL ontologies: A heuristic approach. In *4th International Semantic Web Conference (ISWC2005)*, Galway, Ireland, 2005. Springer.