PROMPTDIFF: A Fixed-Point Algorithm for Comparing Ontology Versions

Natalya F. Noy and **Mark A. Musen**
Stanford Medical Informatics, Stanford University, 251 Campus Drive, Stanford, CA 94305, USA
{noy, musen}@smi.stanford.edu

Abstract

As ontology development becomes a more ubiquitous and collaborative process, the developers face the problem of maintaining versions of ontologies akin to maintaining versions of software code in large software projects. Versioning systems for software code provide mechanisms for tracking versions, checking out versions for editing, comparing different versions, and so on. We can directly reuse many of these mechanisms for ontology versioning. However, version comparison for code is based on comparing text files—an approach that does not work for comparing ontologies. Two ontologies can be identical but have different text representation. We have developed the PROMPTDIFF algorithm, which integrates different heuristic matchers for comparing ontology versions. We combine these matchers in a fixed-point manner, using the results of one matcher as an input for others until the matchers produce no more changes. The current implementation includes ten matchers but the approach is easily extendable to an arbitrary number of matchers. Our evaluation showed that PROMPTDIFF correctly identified 96% of the matches in ontology versions from large projects.

Structural Diffs Between Ontologies

Several recent developments have made ontologies—explicit formal specifications of concepts and relations in a domain—almost ubiquitous. Examples include the emergence of the Semantic Web with formal ontologies as its backbone and the development of easy-to-use tools, which significantly lowered the barrier for ontology development. With tools such as Protégé-2000 (2002) for example, ontology development is no longer an enterprise available only to researchers with graduate degrees in AI. In a sense, ontology development in the worlds of e-commerce and the Semantic Web is becoming a counterpart to conventional software engineering.

As a result, ontology developers now face the same problem that software engineers began to encounter long ago: **versioning and evolution**. Tools for managing versions of software code, such as CVS (Fogel & Bar 2001), have become indispensable for software engineers participating in dynamic collaborative projects. These tools provide a uniform storage mechanism for versions, the ability to check out a particular piece of code for editing, an archive of earlier versions, and mechanisms for comparing versions and merging changes and updates.

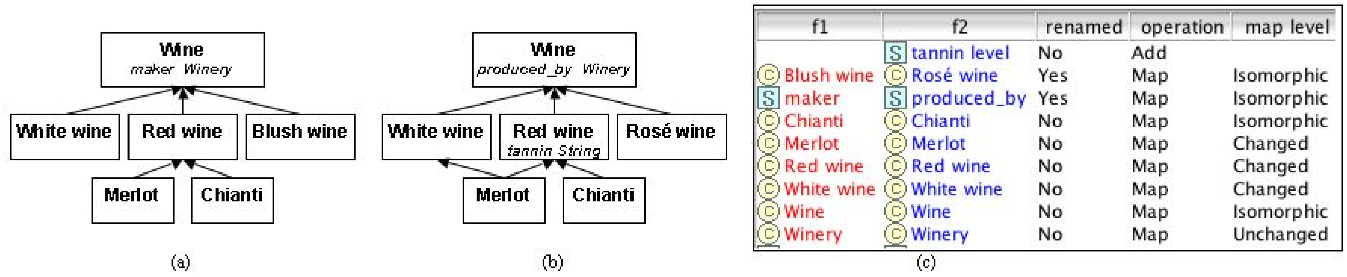
Ontologies change just as the software code does. These changes are caused by changes in the domain itself (our knowledge about the domain changes or the domain itself changes) or in the conceptualization of the domain (we may introduce new distinctions or eliminate old ones). Furthermore, ontology development in large projects is a dynamic process in which multiple developers participate, releasing subsequent versions of an ontology. Naturally, collaborative development of dynamic ontologies requires tools that are similar to software-versioning tools. In fact, ontology developers can use the storage, archival, and check-out mechanisms of tools like CVS with very little change. There are two areas, however, that require new techniques to manage versions of ontologies: (1) representation formalisms to store ontologies and (2) version comparison based on *structure* of the data. To address the first issue, researchers are actively developing representation formalisms, such as RDF and RDF Schema (W3C 2000), OIL (Fensel *et al.* 2000), DAML+OIL (Hendler & McGuinness 2000), and so on. The work we are presenting in this paper addresses the second issue: structure-based comparison of ontologies.

Comparison of versions of software code entails a comparison of text files. Code is a set of text documents and the result of comparing the documents—the process is called a **diff**—is a list of lines that differ in the two versions. This approach does not work for comparing ontologies: 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 are introduced in the text file may be different. A representation language may have several mechanisms to express the same thing. Therefore, text-file comparison is largely useless in comparing versions of ontologies. The PROMPTDIFF algorithm, which we describe in this paper, compares the *structure* of ontology versions and not their text serialization.

We use a knowledge model compatible with the Open Knowledge Base Connectivity (OKBC) protocol (Chaudhri *et al.* 1998): an ontology has classes, class hierarchy, instances of classes, slots as first-class objects, slot attachments to class to specify class properties, and facets to specify constraints on slot values.[1] All these elements are also present in other representation formalisms such as RDFS and DAML+OIL (sometimes in a slightly different form). Therefore, our results apply to ontologies defined in these languages as well.

Suppose that we are developing an ontology of wines. In the first version (Figure 1a), there is a class $Wine$ with three subclasses, $Red\ wine$, $White\ wine$, and $Blush\ wine$. The

[1] OKBC also allows procedural attachments and specialized axioms, which we do not consider.



f1	f2	renamed	operation	map level
	S tannin level	No	Add	
C Blush wine	C Rosé wine	Yes	Map	Isomorphic
S maker	S produced_by	Yes	Map	Isomorphic
C Chianti	C Chianti	No	Map	Isomorphic
C Merlot	C Merlot	No	Map	Changed
C Red wine	C Red wine	No	Map	Changed
C White wine	C White wine	No	Map	Changed
C Wine	C Wine	No	Map	Isomorphic
C Winery	C Winery	No	Map	Unchanged

Figure 1: Two versions of a wine ontology (a and b) and the PROMPTDIFF table showing the difference between the versions

class $Wine$ has a slot $maker$ whose values are instances of class $Winery$. The class $Red\ wine$ has two subclasses, $Chianti$ and $Merlot$. Figure 1b shows a later version of the same ontology fragment. Note the changes: we changed the name of the $maker$ slot to $produced\_by$ and the name of the $Blush\ wine$ class to $Rosé\ wine$; we added a $tannin\ level$ slot to the $Red\ wine$ class; and we discovered that $Merlot$ can be white and added another superclass to the $Merlot$ class. Figure 1c shows the differences between the two versions in a table produced automatically by PROMPTDIFF. The first two columns are pairs of matching frames from the two ontologies. Informally, 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. Other columns in the table provide more information about the match. The third column identifies whether or not the frame has been renamed. The last two columns specify how much the frame has changed, if at all (see the next section). Similar to the diff between text files, the table in Figure 1c presents an **ontology diff**. We give a formal definition of the ontology diff in the next section.

The PROMPTDIFF algorithm 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 frames remains unchanged (in fact, in our experiments, 97.9% of frames remained unchanged) and (2) If two frames have the same type (i.e., they are both classes, both slots, etc.) and have the same or very similar name, one is almost certainly an image of the other. Both of these observations are not true if we are comparing two *different* ontologies that came from different sources rather than two versions of the *same* ontology. Consider a class $University$ for example. In two different ontologies, the class may represent either a university campus, or a university as an organization, with its departments, faculty, and so on. If we encounter a class $University$ in two versions of the same ontology, we can be almost certain that it represents exactly the same concept (and because we have a human looking at the results in the end, we can tolerate the ”almost” adverb in that sentence).

At the same time, the tasks of comparing different ontologies (for example, for the purposes of ontology merging or integration) and comparing versions of the same ontology are closely related. 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. We used heuristics that are similar to the ones we present in this paper to provide suggestions in interactive ontology merging (Noy & Musen 2000). However, because in PROMPTDIFF we are dealing with two versions of the same ontology, we can be much more certain about the results the heuristics produce and require significantly less input and verification from the user.

The process of comparing versions of ontologies would have been greatly simplified if we had logs of changes between versions. However, given the de-centralized environment of ontology development today, it is unrealistic to expect that such logs will be available. Many ontology-development tools do not provide any logging capability. Ontology libraries are set up to publish versions of ontologies but not the logs of changes. Representation formats address representation of the ontologies themselves but not changes in ontologies. Therefore, we can expect that the need for comparing versions when a log of changes between them is not available will continue to grow.

In the rest of this paper we describe different heuristic matchers that we used, and the way we combined them in the PROMPTDIFF algorithm. Specifically, this paper makes the following contributions:

- We define the notion of a structural diff between ontology versions.
- We present a set of heuristic matchers for finding a structural diff automatically.
- We present an efficient and extendable fixed-point algorithm that combines the matchers to produce the diff.
- We evaluate the algorithm's performance using versions of large real-world ontologies.

Structural Diff and PROMPTDIFF Table

We define a **structural diff** between two 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 frame pairs $\langle F_1, F_2\rangle$ where:*

- *$F_1 \in V_1$ or $F_1 = null$; $F_2 \in V_2$ or $F_2 = null$*
- *F_2 is an* **image** *of F_1* (**matches** *F_1), that is, F_1 became F_2. If F_1 or F_2 is $null$, then we say that F_2 or F_1 respectively does not have a match.*
- *Each frame from V_1 and V_2 appears in at least one pair.*

- *For any frame F_1, if there is at least one pair containing F_1, where $F_2 \neq null$, then there is no pair containing F_1 where $F_2 = null$ (if we found at least one match for F_1, we do not have a pair that says that F_1 is unmatched). The same is true for F_2.*

Note that the definition implies that for any pair of frames F_1 and F_2, there is at most one entry $\langle F_1, F_2\rangle$.

The structural diff describes which frames have changed from one version to another. However, for a diff to be more useful to the user, it should include not only *what* has changed but also some information on *how* the frames have changed. A PROMPTDIFF table, which results from the PROMPTDIFF algorithm, provides this more detailed information (Figure 1c).

Definition 2 (PROMPTDIFF table) *Given two versions of an ontology O, V_1 and V_2, the* PROMPTDIFF **table** *is a set of tuples $\langle F_1, F_2, rename\_value, operation\_value, mapping\_level\rangle$ where:*

- *There is a tuple $\langle F_1, F_2, rename\_value, operation\_value, mapping\_level\rangle$ in the table iff there is a pair $\langle F_1, F_2\rangle$ in the structural diff $D(V_1, V_2)$.*
- *$rename\_value$ is $true$ if frame names for F_1 and F_2 are the same; $rename\_value$ is $false$ otherwise.*
- *$operation\_value \in OpS$, where $OpS = \{add, delete, split, merge, map\}$*
- *$mapping\_level \in MapS$, where $MapS = \{unchanged, isomorphic, changed\}$*

The operations in the operation set OpS indicate to the user how a frame has changed from one version to the other: whether it was added or deleted, whether it was split in two frames, or whether two frames were merged. We assign a map operation to a pair of frames if none of the other operations applies. The $mapping\_level$ indicates whether the matching frames are different enough from each other to warrant the user's attention. If the $mapping\_level$ is $unchanged$, then the user can safely ignore the frames—nothing has changed in their definitions. If two frames are $isomorphic$, then their corresponding slots and facet values are images of each other, but not necessarily identical images. The $mapping\_level$ is $changed$ if the frames have slots or facet values that are not images of each other. In Figure 1c, the $Red\ wine$ class has changed: It got a new slot. The pair of the $Chianti$ classes is marked as $isomorphic$: Even though the frames themselves have not changed, the frames that they directly reference ($Red\ wine$) have. Our implementation of PROMPTDIFF also provides the information to the user explaining why the table row contains a particular operation or mapping level (not shown in Figure 1c).

PROMPTDIFF Heuristic Matchers

The PROMPTDIFF algorithm combines an arbitrary number of heuristic matchers, each of which looks for a particular property in the unmatched frames. Before describing some of the matchers that we used, we define a **monotonicity principle**, a principle to which all the matchers in PROMPTDIFF must conform.

Definition 3 (Monotonicity principle) *Let M be a matching algorithm and T_1 and T_2 be the* PROMPTDIFF *tables before and after execution of M. Then for every two frames F_1 and F_2, such that $F_1 \in V_1$ and $F_2 \in V_2$, if the pair $\langle F_1, F_2\rangle$ is present in T_1, then $\langle F_1, F_2\rangle$ is present in T_2.*

A matcher conforms to the monotonicity principle iff it does not retract any matches already in the table. It may delete the rows where one of the frames is $null$ by creating new matches. But it must keep all the existing matches.

We now describe some of the heuristic matchers that we used. Note that each of the matchers is fairly simple and the strength of our approach lies in the combination of these matchers. Also, each of the matchers looks at a particular part of the ontology structure: $is-a$ hierarchy, slots attached to a class, and so on.

Each of the matchers in the list is a *heuristic* matcher. Therefore, there can always be an example when the result produced by the matcher is wrong. However, we have examined ontology versions in several large projects, and we have not come across such examples. Therefore, we believe that most of the time the matchers would produce correct results. Furthermore, PROMPTDIFF presents the matching results to a human expert for analysis, highlighting the frames that have changed. The expert can examine the matches for these changed frames to see if they are correct. These frames usually constitute only a small fraction of all the frames in an ontology. Therefore, even for very large ontologies, human experts need to examine only a small number of frames.

In the following descriptions of matchers, F_n denotes a frame of any type, class, slot, facet, or instance; C_n denotes a class; S_n denotes a slot. The heuristic matchers compare two ontology versions looking for the following situations:

1. **Frames of the same type with the same name.** In Figure 1, both ontology versions, V_1 and V_2, have a frame $Wine$, and in both versions this frame is a class. In this situation, the matcher declares that the two frames match. In general, if $F_1 \in V_1$ and $F_2 \in V_2$ and F_1 and F_2 have the same name and type, then F_1 and F_2 match. Frames can be of type class, slot, facet, or instance. In our experiments, this matcher produced an average of 97.9% of all the matches since ontologies usually do not change a lot from one version to the next.
2. **Single unmatched sibling.** In the example in Figure 1, suppose we matched the classes $Wine$, $Red\ wine$, and $White\ wine$ from V_1 to their counterparts with the same names in V_2. Then the $Wine$ class in both versions has exactly one unmatched subclass: $Blush\ wine$ in V_1 and $Rosé\ wine$ in V_2. In this situation, we conclude that the $Rosé\ wine$ class is the image of the $Blush\ wine$ class. In general, if $C_1 \in V_1$ and $C_2 \in V_2$, C_1 and C_2 match, and each of the classes has exactly one unmatched subclass, $subC_1$ and $subC_2$, respectively, then $subC_1$ and $subC_2$ match. We have a similar matcher for multiple unmatched siblings that can be distinguished by their sets of slots.
3. **Siblings with the same suffixes or prefixes.** Taking the example in 1 further, suppose we remove ”wine” from the class name for subclasses of the $Wine$ class (Figure 2a). Therefore, all the names for those subclasses have changed. However, if we observe that they have all changed in the same way—the same suffix has been removed—we can create the corresponding matches any-

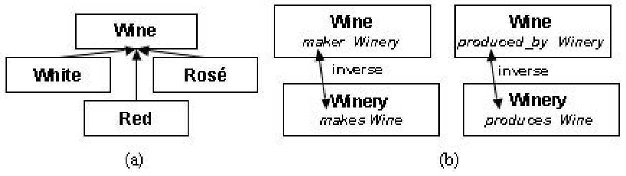


Figure 2: (a) Class names for subclasses of $Wine$ are the same as in Figure 1b except for the "wine" suffix; (b) slots $maker$ and $makes$ are inverse slots in one version; slots $produced\_by$ and $produces$ are inverse slots in another version.

way. In general, if $C_1 \in V_1$ and $C_2 \in V_2$, C_1 and C_2 match, and the names of all subclasses of C_1 are the same as the names of all subclasses of C_2 except for a constant suffix or prefix, then the subclasses match.

4. **Single unmatched slot.** In the example in Figure 1, suppose we matched the class $Wine$ from the first version to its counterpart in the second version. Each of the two classes has a single slot that is so far unmatched: $maker$ and $produced\_by$, respectively. Not only is each of these slots the only unmatched slot attached to its respective class, but also the range restriction for the slot is the same: the class $Winery$. Therefore, we can match the slots $maker$ and $produced\_by$. In general, if $C_1 \in V_1$ and $C_2 \in V_2$, C_1 and C_2 match, and each of the classes has exactly one unmatched slot, S_1 and S_2 respectively, and S_1 and S_2 have the same facets, then S_1 and S_2 match.
5. **Unmatched inverse slots.** If a knowledge model allows definition of inverse relationships, we can take advantage of such relationships to create matches as well. Suppose we have a slot $maker$ in V_1 (at the $Wine$ class in Figure 1), which has an inverse slot $makes$ at the $Winery$ class (Figure 2b); and a slot $produced\_by$ in V_2, which has an inverse slot $produces$. Once we match the slots $maker$ and $produced\_by$, we can match the slots $makes$ and $produces$ because they are inverses of the slots that match. In general, if $S_1 \in V_1$ and $S_2 \in V_2$, S_1 and S_2 match, $invS_1$ and $invS_2$ are inverse slots for S_1 and S_2 respectively, and $invS_1$ and $invS_2$ are unmatched, then $invS_1$ and $invS_2$ match.
6. **Split classes.** Suppose that an early definition of our wine ontology included only white and red wines and we simply defined all rosé wines as instances of $White\ wine$. In the next version, we introduced a $Rosé\ wine$ class and moved all instances corresponding to rosé wines to this new class. In other words, the class $White\ wine$ was split into two classes: $White\ wine$ and $Rosé\ wine$. In general, if $C_0 \in V_1$ and $C_1 \in V_2$ and $C_2 \in V_2$, and for each instance of C_0, its image is an instance of either C_1 or C_2, then C_0 was split into C_1 and C_2. A similar matcher identifies classes that were merged.

Note that each of the matchers in the list considers only the frames that have not yet been matched. Thus, even though potentially each of the matchers will need to examine every tuple in the current PROMPTDIFF table, in practice, each matcher, except for the first one, examines only a very small number of tuples (the ones that have null values either for F_1 or F_2).

The PROMPTDIFF Algorithm

We combine all the available heuristic matchers (such as the ones we described in the previous section as well as any other available matchers) in the PROMPTDIFF algorithm, a fixed-point algorithm that produces the complete PROMPTDIFF table for two ontology versions. PROMPTDIFF runs all the matchers until they produce no new changes in the table. Because no matcher retracts the results of previous matchers or its own results from previous runs (the monotonicity principle), the algorithm always converges.

Dependency Among Matchers

A simple-minded implementation of such a fixed-point algorithm will run a set of all matchers in sequence until the whole set produces no more changes. However, we can greatly improve the efficiency of the algorithm using the following observations. First, not all matchers use all the information available in the table. For instance, the **single unmatched siblings** matcher never considers matches between slots. The **inverse slots** matcher does not care about the class-matching information. Second, the type of information in the table that each matcher can modify is limited. The **single unmatched sibling** matcher can create new matches between classes but never between slots. The **inverse slot** matcher can create new matches between slots but never between classes. Therefore, if each matcher declares what type of matching information it uses and what type of matching information it modifies, we can create a dependency table among the matchers and use this table to make the implementation more efficient.

Table 1 shows a dependency table for the matchers we described. For each matcher, we specify what type of information—matches between classes or slots—it uses and modifies (we omit facets and instances due to lack of space). Based on this specification, we determine dependency among the matchers (the last column in Table 1). Notice that none of the matchers affects the first matcher, which compares names and types of frames. After we run it once, we do not need to run it again: the results of other matchers will not change its results. Conversely, the **single unmatched slots** matcher uses existing matches between classes and slots and therefore all the matchers that modify this information affect this matcher (in our example, this list includes all matchers). Thus, if any of the matchers changes the PROMPTDIFF table, we need to run this matcher again.

PROMPTDIFF uses the dependency table to determine the order in which it executes the matchers. It keeps a stack of matchers it still needs to run. It starts by putting the matchers that do not affect any other matchers at the bottom of the stack and matchers that are not affected by other matchers at the top. Then it executes matcher M at the top of the stack. If M produced changes in the PROMPTDIFF table, the algorithm adds to the stack all the matchers that depend on M, removing duplicates. It runs until the stack is empty.

Performance Analysis

Given two versions of an ontology, V_1 and V_2, with n and m frames, respectively, we show that PROMPTDIFF converges after a linear number of steps and its running time

Matcher	Uses info about		Modifies info about		Affects matchers
	classes	slots	classes	slots	
1. Same type same name	–	–	+	+	2, 3, 4, 5
2. Single unmatched sibling	+	–	+	–	2, 3, 4
3. Siblings with same suffixes	+	–	+	–	2, 3, 4
4. Single unmatched slot	+	+	–	+	4, 5
5. Unmatched inverse slot	–	+	–	+	4, 5
6. Split classes	+	–	+	–	2, 3, 4

Table 1: Dependency table for the matchers in the paper

is $T_{max}O(max(n,m))$, where T_{max} is the running time for the least efficient of the matchers.

We first estimate the size of the PROMPTDIFF table. Without the split and merge operations, there is at most one tuple for each frame in V_1 and V_2. Thus, the number of tuples is $O(n+m) = O(M)$, where $M = max(n,m)$. We allow splits only of one class into two classes and merges of only two classes into one class. Therefore, even if every frame in V_1 were split into two frames in V_2, the size of the table is still $O(n)$. Similarly, even if every frame in V_2 resulted from a merge of two frames in V_1, the size of the table is $O(m)$. Thus, the table size is $O(M)$ and is linear in the size of the ontologies. When PROMPTDIFF runs, it starts with the table where all the frames from V_1 and V_2 are unmatched. The table contains $n+m$ rows. Performing a similar analysis, we can show that the maximum number of possible monotonic changes to the table is finite and it is limited by $O(M)$. If a matcher produced a change in the table, we will have to run at most c more matchers where c is the total number of matchers in the system, a constant. Therefore, the running time of PROMPTDIFF is $T_{max}O(M)$, where T_{max} is the running time for the least efficient algorithm.

All the matchers that we presented in the previous section, except the first one, are linear in the size of the ontologies. The first matcher runs in $O(M)log(M)$ time but it is executed only once. Therefore, given this set of matchers, the running time is $O(M)log(M) + O(M)O(M) = O(M^2)$.

Evaluation

Empirical evaluation is particularly important for heuristic algorithms, because there is no provable way to verify their correctness. We implemented PROMPTDIFF as a plugin for the Protégé-2000 ontology-editing environment (2002). We then evaluated PROMPTDIFF using ontology versions in two large projects at our department: the EON project,[2] and the PharmGKB project.[3] Both projects rely heavily on ontologies, both use Protégé-2000 for ontology development, and both keep records of different versions of their ontologies. We compared consecutive versions of the ontologies, as well as versions that were farther apart. For each pair of versions, we created the PROMPTDIFF table manually (given that the ontologies contained between 300 and 1900 concepts, it was a onerous process) and compared this manually generated table with the one that PROMPTDIFF produced.

Table 2 presents characteristics of the source ontologies and the results of our evaluation. Experiments 1, 2 and 3 used the ontologies form the EON project, which had between 314 and 320 frames in each version. Experiments 1 and 2 compared consecutive versions, whereas experiment 3 compared non-consecutive versions: the first version from experiment 1 and the second version from experiment 2. Experiments 4, 5 and 6 used the ontologies from the PharmGKB project, which had between 1886 and 1895 frames in each version. We compared consecutive versions and then the versions that were farther apart. On average, 97.9% of frames in each version remained unchanged. To evaluate the accuracy of our algorithm, we considered the frames that had changed (the remaining 2.1%)—exactly the frames that a user would need to look at. On average, PROMPTDIFF identified 96% of matches between those frames (this measure is similar to *recall* in information retrieval). 93% of the matches that PROMPTDIFF identified were correct (*precision* in information-retrieval terms).[4] More important, *all* the discrepancies between the manual and the automatic results were confined to the rows that had $null$ in one of the first two columns. In other words, when PROMPTDIFF did find a match for a frame, it was always correct. Sometimes, the algorithm failed to find a match when a human expert could find one. A human expert can determine that two frames are similar even if a rule that he applied in a specific case is not sufficiently general to apply in all cases. It can be a significant overlap in a class name (e.g., $Finding$ versus $Physical\_Finding$), similarity in the slot range (e.g., two slots at matching frames that have the same range), and so on. At the same time, the matchers have to use rules that are sufficiently general to apply them to any ontology.

Let us interpret these numbers from the user's point of view. In the last experiment, for example (row 6 in Table 2), 83 frames from V_1 have changed (in fact, names of 67 of those frames were replaced with system-generated names by accident). The PROMPTDIFF result contained 19 unmatched frames. Given that we can trust the matches that PROMPTDIFF generated, we need to examine only these 19 unmatched frames instead of examining all 1886 in V_1, a significantly simpler task (it turned out that 14 of those frames did not have any matches). As a result, even for very large ontologies, users need to examine manually only a tiny fraction of frames—the ones for which PROMPTDIFF did not find any matches. And PROMPTDIFF conveniently shows these frames first in the PROMPTDIFF table.

Note that the performance of the algorithm did not deteriorate when we considered versions that were farther apart. In fact, both recall and precision were better than in the worst of the two cases for consecutive versions: because there were more changed frames, PROMPTDIFF identified a larger fraction of the frames correctly.

We used ten matchers in the experiments. On average, each matcher was executed 2.3 times in each experiment. Each matcher produced a new result at least once.

We have also experimented with executing matches in different order (still subject to the constraints that the dependency table imposes). We could not find cases where the

[2]http://www.smi.stanford.edu/projects/eon

[3]http://www.pharmgkb.org

[4]We use the terms $precision$ and $recall$ in Table 2

Ontologies		# frames in V_1	# frames in V_2	% frames changed from V_1	# frames unchanged from V_1	# of matches for changed frames that PROMPTDIFF identified	# of **correct** matches for changed frames that PROMPTDIFF identified	Recall	Precision
1	EON1 and EON2	314	320	5	98.4%	15	13	93%	87%
2	EON2 and EON3	320	319	1	99.7%	1	1	100%	100%
3	EON1 and EON3	314	319	6	98.1%	16	14	93%	87%
4	Pharm1 and Pharm2	1886	1891	11	99.4%	18	16	94%	89%
5	Pharm2 and Pharm3	1891	1895	71	96.2%	382	372	99%	97%
6	Pharm1 and Pharm3	1886	1895	83	95.6%	400	389	98%	97%
average					**97.9%**			**96%**	**93%**

Table 2: PROMPTDIFF evaluation results

final set of matches would be different depending on the order in which the matchers were executed. A particular match may be identified by a different matcher, but the final set of matches itself remained unchanged. It is likely that as we extend the set of matchers, we will find cases where the execution order of matchers indeed makes a difference.

Related Work

Current research in *ontology versioning* has addressed two issues: (1) identifying ontology versions in a distributed environments such as the Semantic Web (Klein & Fensel 2001) and (2) specifying explicitly logs of changes between versions (Oliver *et al.* 1999; Heflin & Hendler 2000). However, given the de-centralized nature of ontology development, logs of changes may not always be available. Our research complements these efforts by providing an automatic way to compare different versions based on the semantics encoded in their structure when logs of changes are not available.

The main thrust of research in *database-schema versioning* also uses the assumption that a record of changes between versions is readily available (Roddick 1995). Usually, researchers identify a canonical set of schema-change operations and consider effects of these operations on instance data as it migrates from one version to another (Banerjee *et al.* 1987). Lerner (2000) addresses automatic methods for comparing schema versions. She identifies complex operations such as grouping slots of a class into a different class, which is referenced by the original class and so on. As we try to perform more fine-grained comparison between ontology versions, we will no doubt draw upon her work.

Unlike in the schema-evolution research where the assumption that we have a log of changes is almost universal, the research in *database-schema integration* automates comparison between schemas that originated from different sources. Rahm and Bernstein (2001) survey the approaches that use linguistic techniques to look for synonyms, machine-learning techniques to propose matches based on instance data, information-retrieval techniques to compare information about attributes, and so on. Cupid (Madhavan, Bernstein, & Rahm 2001), for example, integrates many of these approaches in an algorithm that starts by matching leaf concepts in the hierarchy and then proceeding up the hierarchy trees to generate new matches based on matches of the subtrees. In fact, this field can potentially supply many heuristic matchers to integrate in the PROMPTDIFF algorithm. However, none of these algorithms was designed to compare versions of the same schema, but rather different schemas. It would be interesting to see how well they perform in our case. There is one schema-integration algorithm that does rely on source schemas being similar. In designing the TranScm system for data translation, Milo and Zohar (1998) observe that when we need to translate data from an XML document to an object-oriented database, for example, the underlying schemas are often very similar since they describe the same type of data. Therefore, a small number of explicit rules can account for a large number of transformations. TranScm could benefit from many of the simple heuristics we described in this paper, whereas PROMPTDIFF could use some of the TranScm rules as its matchers.

There is a number of large *taxonomies for natural-language processing*, and researchers in that field developed semi-automated techniques for creating mappings between these resources. O'Hara and colleagues (1998) present a heuristic-based approach for finding correspondences between synsets in WordNet (Miller 1995) and concepts in the Mikrokosmos ontology (Mahesh & Nirenburg 1995). The heuristics compare the English representation of terms in both hierarchies, similarity of this representation for ancestors in the hierarchy, overlap in the definition of siblings and children. Daudé and colleagues (2001) used relaxation labeling to map between two versions of WordNet. The authors mainly use the hypernym–hyponym relationships, increasing the weight for a connection if it includes nodes whose hypernym or hyponyms are also connected. They experimented with using both direct and indirect hypernyms and hyponyms. Unlike these approaches, PROMPTDIFF examines not only hierarchical relations but also other relations, such as slot attachment, inverse slots, and instances.

Even though researchers on *semi-automated ontology mapping* also compare disparate ontologies rather than versions of the same ontology, the tools may provide another set of useful extensions for PROMPTDIFF. Ontology-merging tools, such as PROMPT (Noy & Musen 2000), use semantics of links between frames and user's action to produce hypotheses on matching frames. FCA-Merge (Stumme & Mädche 2001) uses a set of shared instances between two ontologies to find candidate matching classes and hierarchical links between them. AnchorPROMPT (Noy & Musen 2001), the articulation engine of the SKAT tool (Mitra, Wiederhold, & Kersten 2000), and Similarity-Flooding algorithm (Melnik, Garcia-Molina, & Rahm 2002) use similarities in the ontology graph structure to suggest candidate matches. Because they use different ontologies, all these algorithms have to be much more "conservative" in their comparisons requiring much more than a simple name and type match between frames to declare that they are similar. However, because the PROMPTDIFF framework is so easily extensible, we can incorporate these algorithms as new matchers in the fixed-point stage and integrate the results. Furthermore, most of the algorithms that we have mentioned either do not use the semantics of links at all or treat only is-a links in a special way. In the matchers that we described in this paper, we have used the semantics of *is-a* links, *instance-of*, slot attachment, slot range, and facet attachment links.

Future Work

In addition to incorporating other comparison algorithms to improve further the accuracy of PROMPTDIFF there are other possible directions to explore.

We can use features present in other formalisms, such as DAML+OIL, in the matchers. For example, we can consider if definitions of disjointness, necessary and sufficient conditions, subproperties, and so on can provide useful clues in the mappings.

We can use the information in the PROMPTDIFF table to generate transformation scripts from one version to another. Computer programs can then use these scripts to migrate instance data (as in schema versioning) or to query one version using another version. Minimizing the number of lossy transformations, which cause values to be lost at intermediate steps, is the main challenge in this task.

In our implementation, we used Java to define matchers. We could use a declarative language like the one described by Abiteboul and colleagues (2001) instead. Declarative specification could enable logic-based inference on rule definitions and their properties.

Another promising extension is assigning an uncertainty factor to the results of different matchers (i.e., a probability that the result is correct) and integrating the results taking into account these probabilities. Doan and colleagues (2001) use a similar approach to integrate results of machine learners for mapping between ontologies.

Acknowledgments

AnHai Doan provided very useful feedback on the paper. We also thank anonymous reviews for their comments, which greatly improved the paper. This work was supported in part by a contract from the U.S. National Cancer Institute.

References

Abiteboul, S.; Cluet, S.; and Milo, T. 2001. Correspondence and translation for heterogeneous data. *Theoretical Comp. Science*.

Banerjee, J.; Kim, W.; Kim, H.-J.; and Korth, H. F. 1987. Semantics and implementation of schema evolution in object-oriented databases. In *SIGMOD Conference*, 311–322.

Chaudhri, V. K.; Farquhar, A.; Fikes, R.; Karp, P. D.; and Rice, J. P. 1998. OKBC: A programmatic foundation for knowledge base interoperability. In *15th Nat. Conf. on Artificial Intelligence (AAAI-98)*, 600–607.

Daudé, J.; Padró, L.; and Rigau, G. 2001. A complete WN1.5 to WN1.6 mapping. In *NAACL-2001 Workshop on WordNet and Other Lexical Resources*.

Doan, A.; Madhavan, J.; Domingos, P.; and Halevy, A. 2001. Learning to map between ontologies on the semantic web. In *The 11th International WWW Conference*.

Fensel, D.; Horrocks, I.; van Harmelen, F.; Decker, S.; Erdmann, M.; and Klein, M. 2000. OIL in a nutshell. In *12th Conf. on Knowledge Engineering and Management (EKAW-2000)*.

Fogel, K., and Bar, M. 2001. *Open Source Development with CVS*. The Coriolis Group, 2nd edition.

Heflin, J., and Hendler, J. 2000. Dynamic ontologies on the web. In *17th Nat. Conf. on Artificial Intelligence (AAAI-2000)*.

Hendler, J., and McGuinness, D. L. 2000. The DARPA agent markup language. *IEEE Intelligent Systems* 16(6):67–73.

Klein, M., and Fensel, D. 2001. Ontology versioning on the Semantic Web. In *The First Semantic Web Working Symposium*.

Lerner, B. S. 2000. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems* 25(1):83–127.

Madhavan, J.; Bernstein, P. A.; and Rahm, E. 2001. Generic schema matching using Cupid. In *27th Int. Conf. on Very Large Data Bases (VLDB '01)*.

Mahesh, K., and Nirenburg, S. 1995. A situated ontology for practical NLP. In *Workshop on Basic Ontological Issues In Knowledge Sharing*.

Melnik, S.; Garcia-Molina, H.; and Rahm, E. 2002. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *18th Int. Conf. on Data Engineering (ICDE-2002)*. San Jose, CA: IEEE Computing Society.

Miller, G. A. 1995. WordNet: A lexical database for english. *Communications of ACM* 38(11):39–41.

Milo, T., and Zohar, S. 1998. Using schema matching to simplify heterogeneous data translation. In *24th Int. Conf. on Very Large Data Bases (VLDB'98)*, 122–133. New York: Morgan Kaufmann.

Mitra, P.; Wiederhold, G.; and Kersten, M. 2000. A graph-oriented model for articulation of ontology interdependencies. In *Conf. on Extending Database Technology 2000 (EDBT'2000)*.

Noy, N. F., and Musen, M. 2000. PROMPT: Algorithm and tool for automated ontology merging and alignment. In *17th Nat. Conf. on Artificial Intelligence (AAAI-2000)*.

Noy, N. F., and Musen, M. A. 2001. Anchor-PROMPT: Using non-local context for semantic matching. In *Workshop on Ontologies and Information Sharing at IJCAI-2001*.

O'Hara, T.; Mahesh, K.; and Nirenburg, S. 1998. Lexical acquisition with WordNet and the Mikrokosmos ontology. In *COLING/ACL Wordkshop on Usage of WordNet in NLP Systems*. ACL.

Oliver, D. E.; Shahar, Y.; Shortliffe, E. H.; and Musen, M. A. 1999. Representation of change in controlled medical terminologies. *Artificial Intelligence in Medicine* 15:53–76.

Protege. 2002. The Protege project. *http://protege.stanford.edu*.

Rahm, E., and Bernstein, P. A. 2001. A survey of approaches to automatic schema matching. *VLDB Journal* 10(4).

Roddick, J. F. 1995. A survey of schema versioning issues for database systems. *Information and Software Technology* 37(7).

Stumme, G., and Mädche, A. 2001. FCA-Merge: Bottom-up merging of ontologies. In *7th Intl. Conf. on Artificial Intelligence (IJCAI '01)*, 225–230.

W3C. 2000. Resource description framework (RDF).