Change Detection in Ontologies Using DAG Comparison

Johann Eder[1] and Karl Wiggisser[2]

[1] University of Vienna, Dep. of Knowledge and Business Engineering
johann.eder@univie.ac.at
[2] Klagenfurt University, Dep. of Informatics-Systems
wiggisser@isys.uni-klu.ac.at

Abstract. Ontologies are shared conceptualizations of a domain. As this domain may change over the time, the ontology has to evolve as well. Additionally, for many applications, it is important to know which version of an ontology was valid at a certain point in time. Several ontology version management systems address this problem. If a user is confronted with different versions of an ontology it is often necessary to identify the changes. We present an efficient graph based approach for change detection between two versions of an ontology based on structural comparisons. The result is a change script transforming the old to the new version. Furthermore, we present an extensive evaluation of the prototype implementation of the change detection system.

1 Introduction

An ontology is *an explicit specification of a conceptualization* [1]. Ontologies are seen as important technique for semantic data processing, and in particular for interoperability. They represent knowledge about a certain real world domain. But as the real world tends to change, the ontologies have to change as well. Knowledge about these changes is mandatory to correctly interpret data or documents which were based on the semantics defined in the changed ontology. Furthermore, the correct comparison of data and documents from different points in time, based on different versions of an ontology is only possible if the differences between these versions are known. E. g. when analyzing the development of unemployment rate in the European Union over the last 30 years one has to be aware that both the European Union and the formula for computing the rates changed considerably over this period of time.

Changes between versions of an ontology might not be explicitly available. Frequently, only the different versions are available, but a change history is missing. To help in this situation is the ambition of the work presented here. In particular, we focus on the following problem: Given two versions of an ontology we want to derive an edit script, i. e. a series of change operations, which is able to transform one version into the other. This edit script is then an explicit representation of changes which occurred between the versions of the ontology.

Our change detection system is based on the structure of the ontology only. One might argue that the important changes in ontologies are changes in the semantics. We assume that every semantic change has to be represented by a structural change, as otherwise two identical representations will have different semantics. There might be structural changes which are not semantic changes. Examples for such changes are representational variations for performance increase. Thus, a fast and reliable algorithm for identifying and describing structural changes is a good start for analyzing the changes in the semantics.

In this paper we present our graph based algorithm for a semiautomatic change detection between two versions of an ontology in detail. Based on an extensive evaluation of the algorithm (Sect. 5) we claim that it is very efficient in terms of both speed and precision.

2 Related Work

In [2,3] we presented a graph based approach for ontology versioning. Incorporating changes in such a temporal ontology is easy if one knows all changes, but can be a very complex task, if the differences are not previously known. This is particularly important for users of ontologies who have access to the latest version, but do not have a representation of the changes since their last download. There are approaches for ontology comparison published, e. g. [4,5,6]. Among them, PromptDiff, as a part of the Protege framework [7], and OntoView, a web based system, are the best known. However, to the best of our knowledge, there are no evaluation figures for these systems published. Ontology matching/alignment/merging systems like GLUE [8], Cato [9] or Chimaera [10], although somehow related to our change-detection problem, in fact address a different issue. They are more intended to find the semantic overlapping of two or more *different independently developed ontologies*, whereas our approach is designed to find changes in *two versions* of the *same ontology.*

In [11], we presented a brief first sketch of our concepts without detailed description of the algorithm. It is an extension of an algorithm, successfully applied for identifying changes in dimension structures of data warehouses [12,13]. The major challenges were the far more complex (data) structure of ontologies and the usage of ontology specific information (in particular various forms of relationships) for further improving the accuracy of the applied heuristics.

Graph matching and graph comparison is a long known problem. Because the graph isomorphism problem is in $\mathcal{NP}$ [14], there are several approaches comparing two graphs and/or determine their edit distance using some heuristics or restricting the data structure. For instance, the approaches presented in [15,16,17,18,19,20,21] are only some of them. We evaluated these algorithms but they all have some shortcomings which make them either completely unusable for our purpose or at least very hard to adapt to our problem. Some of the approaches are defined on undirected graphs and others are missing operations essential for our purpose. If an adaption was possible, main advantages of the algorithms would have vanished.

To the best of our knowledge there is no algorithm which can easily be adapted to calculate the edit operations between two DAGs (directed acyclic graphs) as we need them for our ontology versioning system. So we developed a new algorithm inspired by the tree comparison algorithm of Chawathe et al. [22]. It is built upon the same principles, but with major enhancements to support the comparison of directed acyclic graphs. The renaming detection component is adapted from our previous work in this area [12].

3 Ontology Graphs and Graph Operations

An ontology can be seen as a graph where the concepts are represented by nodes and semantic relations between concepts by edges. A node consists of an unique *id*, a *label* which represents the concept's name, an object holding implementation-dependent *attributes* (e. g. some comment or description of the concept) and a set of *slots*, a concept we will describe later. The ontology's relations are represented by edges. An edge consists of two nodes (*parent* and *child*) and an *edgetype*, which represents the type of the relation. Common ontological relations like generalization (`IS-A`) or aggregation (`PART-OF`) typically build up a *directed acyclic graph* (DAG). Other relations, e. g. `IS-FRIEND-OF`, may also create cycles in the graph. The user may explicitly define edge types as *acyclic*, i. e. not creating cycles in the graph. All other edges are per default treated as *possibly cyclic*, thus may build up cyclic graphs. For our approach, we assume the ontology graph to be a *rooted directed acyclic graph* (RDAG), i. e. a DAG with exactly one node not having any parents.

To transform an arbitrary digraph, representing an ontology version, to such a RDAG we perform the following steps: First, we assume that there is one single root in the graph, i. e. there is only one node in the graph, not having any parents. If such a root is not present, we create a new node $root_v$, which becomes the *virtual root* of the graph by creating a *vroot* edge from $root_v$ to each node x in the graph not having any parents yet. Next we eliminate cycles. For that purpose, we assume that every node is connected to the root via a path consisting only of edges defined to be acyclic. This will hold for many ontologies, because they often comprise a generalization hierarchy. For all nodes x not satisfying this requirement, we create a *vroot* relation from the root to x. As a last step, we create the so called *slots*, representing cyclic edges. Each slot has a *name* and a *type*. For each relation from a node *parent* to node *child*, with relation type $edgeType$, where $edgeType$ is not defined as acyclic, we add the slot with name $edgeType$ of type *child* to the node *parent* and remove the edge between *parent* and *child*. With this steps we can transform any graph into a RDAG. When we assume all relations in an ontology to be directed, i. e. we can determine the start and the end of a relation, this transformation is lossless and unique and can be reversed by replacing the slots of each node with the respective edge and removing the *vroot* node and edges.

Figure 1 shows an example for such a transformation. On the left the original cyclic graph is shown. The relations `IS A` and `PART OF` are defined to be acyclic.

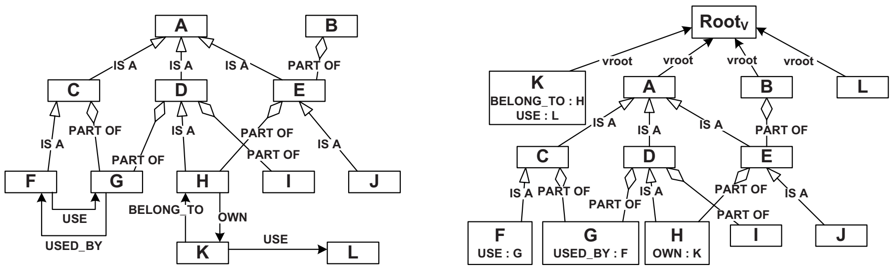

Fig. 1. Transformation of arbitrary graph to RDAG

All other relations may build cycles. There is no single root. In the right, the resulting RDAG is shown. A new node $Root_v$ is created and the nodes A and B are attached to it with a *vroot* relation. Now all nodes except K and L are connected to the root via a path, consisting only of acyclic edges. So these two nodes are also connected with a *vroot* relation. For edges like *use* or *belong_to* slots are inserted to the respective nodes, e. g. K.

With the operations defined below we can transform any two RDAG into each other. We represent the old version of an ontology with the graph v_{old} and a new version of the same ontology with the graph v_{new}. Our goal is to find the differences between two ontology versions in terms of graph operations. We present an algorithm which calculates a so called edit script, which is a sequence of graph operations that transform v_{old} into v_{new}. This edit script acts as representation for the changes between the ontologies and thus enables us to incorporate changes of the ontology into virtually any ontology versioning system, for instance like proposed in [2]. The operations defined on the ontology graph are:

– $InsertNode(name, attributes, slots, parents)$ inserts a new node with the label *name*, the attributes *attributes*, and set of *slots* to all *parents*. The set *parents* holds pairs of nodes and edge types $(parent, type)$, meaning the edge from *parent* to the new node to be of type *type*.
– $DeleteNode(node)$ deletes *node* from the graph. A node can only be deleted, if it does not have any children. With the node, all its incident edges are deleted as well.
– $InsertEdge(parent, child, type)$ creates an edge of type *type* from *parent* to *child*. The new edge must not close a cycle in the graph.
– $DeleteEdge(parent, child)$ deletes the edge from *parent* to *child*.
– $InsertSlot(node, slot)$ adds a new *slot* consisting of name and type to *node*.
– $DeleteSlot(node, slot)$ removes the *slot* from the *node*.
– $UpdateNode(node, attributes)$ changes the attributes of *node* to *attributes*.
– $RenameNode(node, name)$ changes the name of *node* to *name*.
– $ChangeEdgeType(parent, child, type)$ changes the type of the edge between *parent* to *child* to *type*.

4 The Comparison Algorithm

Our graph comparison algorithm is inspired by the tree comparison algorithm of Chawathe et al. [22]. Although the algorithm works quite well on ordered trees, it has some shortcomings for our purpose: (i) It is defined on trees and not on RDAG structures. (ii) The renaming of nodes is not supported. (iii) It depends on the ordering of the nodes' children. (iv) It does not support typed edges.

Our approach is built upon the same principles, especially when calculating the node matching between two graphs, but includes some major changes in order to support the comparison of directed acyclic graphs. The renaming detection component is adapted from our previous work in this area [12]. The algorithm is based on the assumption that ontologies do not change very much from one version to the next. This is also supported by [4].

4.1 The Longest Common Subsequence

A *Subsequence* of a string is any string obtained by deleting zero or more symbols from the given string. A *Common Subsequence* of two strings A and B is a subsequence of both [24]. The *Longest Common Subsequence*(LCS) of two strings A and B is a common subsequence of A and B such that there is no common subsequence of A and B which contains more symbols. Note that the LCS is not necessarily unique, but there can be different common subsequences of maximal length. Efficient algorithms calculating the LCS are for instance given in [24,25].

The concept of subsequences can easily be extended from strings to sequences of objects of any type. For that purpose we also need to specify a comparison function, which determines whether two elements stemming from either of the sequences are equal. We define the function LCS as follows: $LCS(A, B, equal)$, where A and B are sequences of objects of the same type and $equal(a, b)$ is a function which decides the equality of the objects a and b and returns either `true` or `false`. The function returns a sequence of object pairs $\langle(a_1, b_1), \ldots, (a_n, b_n)\rangle$ with the following properties: (i) $a_i \in A$ and $b_i \in B$ (ii) $equal(a_i, b_i) =$ `true` (iii) $\langle a_1, \ldots, a_n\rangle$ is a subsequence of A and $\langle b_1, \ldots, b_n\rangle$ is a subsequence of B (iv) There is no longer sequence of object pairs which fulfils (i)–(iii).

We use this LCS-function to efficiently compare sequences of graph nodes during node matching (see Sec. 4.3).

4.2 Node Matching

The first step, when comparing two graphs is to find a good matching between them, i. e. finding nodes which represent the same concept in both graphs.

As for ontologies the concept's name often acts as key, we defined the node's name to be the primary key for matching nodes. That means two nodes cannot match if their names differ. Furthermore, we do not expect nodes to change their hierarchical position within the graph, i. e. leaf nodes will rarely become inner nodes and vice versa. The third assumption we act on is that the attributes and descendants, i. e. edges and slots, will not change very much. Thus, we define a function $similar(x, y)$ returning `true` or `false`, which compares two concepts x and y to take into account the following properties (compare [22,26]):

1. $x.name = y.name$: Two concepts can only match if their name is equal.
2. The function $compare(x, y)$ compares all attributes of x and y for similarity and returns a number between 0 (no similarity) and 1 (identical).
3. $commonSlotsRatio(x, y) = \frac{commonSlots(x,y)}{maxSlots(x,y)}$, where $commonSlots(x, y)$ is the number of slots appearing in both, x and y. The function $maxSlots(x, y)$ is the maximum number of slots of x and y. Thus, $commonSlotsRatio(x, y)$ returns a value between 0 and 1. If none of the nodes contains slots, the function is defined to return 1.
4. $commonLeavesRatio(x, y) = \frac{commonLeaves(x,y)}{maxLeaves(x,y)}$, with $commonLeaves(x, y)$ calculating the number of common, i. e. matched, leaf descendants of x and y and $maxLeaves(x, y)$ gives the maximum number of leaf descendants of x and y. So $commonLeavesRatio(x, y)$ returns a number between 0 and 1. If one of the nodes is a leaf, the function is defined to return 1.

The user may configure the influence for each of these similarity measures, depending on the expected changes. For instance, if the ontology's structure has remained stable but the comments for many concepts changed, the administrator can pay more attention to common slots and leaves than to attributes. Only the criterion of equal names is mandatory. The function $similar(x, y)$ returns `true` iff the two concepts have the same name and each of the above comparison functions returns a value greater than the user defined threshold.

4.3 Matching Algorithm

Chawathe et al.'s matching algorithm relies on the ordering of children. Ontologies do not have such an ordering, but a defined order dramatically reduces the complexity during the matching. Therefore, we first sort the nodes' children alphabetically by their name. Then, for each of the graphs we build a list of leaves, traversing the graph from left to right. From these two node lists we build the Longest Common Subsequence, with the function $similar(x, y)$ as equality check. This gives a set of matchings $\mathbb{M}$. A matching is a pair of nodes (n_i, n_j), with $n_i \in v_{old}$ and $n_j \in v_{new}$ which represent the same concept in both versions. We do the same for all inner nodes and add the resulting pairs to $\mathbb{M}$. As a last step during matching calculation, we build a list of still unmatched nodes for each of the graphs and run the LCS algorithm again. The alphabetic sorting of nodes will significantly reduce the effort for the LCS. Figure 2 shows the pseudocode for the matching calculation.

The first run of LCS will match all similar leaves. The second run of LCS does the matching of all similar inner nodes, which depends on the matchings of leaves. The third run of LCS will match nodes, which can either be inner nodes or leaves. The execution order takes into account our assumption that inner nodes seldom will become leaf nodes and vice versa. This approach is a heuristic one and errors may occur. Thus, in production environments, the administrator

```
Function calculateMatching($v_{old}, v_{new}$)
1. Let matching set $\mathbb{M} = \emptyset$;
2. Let $L_o(L_n)$ be the list of leaves, when traversing $v_{old}(v_{new})$ from left to right;
3. Let $\mathbb{M} = \mathbb{M} \cup LCS(L_o, L_n, similar)$
4. For each unmatched node $x$ in $L_o$ if there is an unmatched node $y$ in $L_n$ with
   $similar(x, y)$: $\mathbb{M} = \mathbb{M} \cup \{(x, y)\}$;
5. Repeat steps 2 – 5 for inner nodes;
6. Repeat steps 2 – 5 for all still unmatched nodes;
7. Let the user acknowledge and correct $\mathbb{M}$
8. Return $\mathbb{M}$;
```

Fig. 2. Pseudocode for the calculation of the matching set

must have the possibility to modify the results of the algorithm, i. e. break up matchings or establish matchings not detected by the system.

Of course, this matching order does not guarantee the best matching, i. e. the matching with the minimum differences. Consider two concepts a and a' within the same ontology version which have the same label. They are quite similar to each other such that $similar(a, a')$ returns `true`, but they are not equal. Now in version 1 of the ontology, when traversing the graph, they appear in order a and later a' but in the traversal of version 2 they appear in order a' and then a. As we build the LCS of these traversal sequences and $similar(a, a')$ gives `true`, a from version 1 will be matched to a' from version 2 and vice versa. Thus, this is not the best matching. But as we assume that such situations seldom occur and the calculation a perfect matching is considered to be in $\mathcal{NP}$ [14], we think the result is reasonably good with respect to the gained performance.

4.4 Renaming Detection

As the name of a concept is the primary matching criterion, all nodes that cannot be matched could possibly have been renamed. So in the next step, we try to find pairs of nodes, which differ in their names but are so similar with respect to their attributes and structure that they may represent the same concept nonetheless.

Theoretically, each unmatched node from the old graph could have been renamed to any unmatched node in the new graph. To reduce complexity, we only consider node pairs under matched parents as possible renamings. But as this rule may foreclose many renamings to be detected, when, for instance, a new prefix is added to every node, in this phase we also consider possible renamed parents as matched parents. So the renaming of node $w \in v_{old}$ to $x \in v_{new}$ can be detected iff there is at least one parent of w that is matched to a parent x or possible renamed to a parent of x. Pairs of nodes, which are possibly renamed are stored in the set *possibleRenamings*. We sort this set according to the hierarchical position of the node such that pairs containing only leaves come first, then the pairs containing only inner nodes, and last the mixed pairs.

Next we calculate the similarity for all possibly renamed pairs, again using the function $similar(w, x)$, but now of course neglecting the different node name. If the similarity of a pair is greater than a user defined threshold, it is considered as

```
Function calculateRenamings($v_{old}, v_{new}, \mathbb{M}$)
1. Let edit script $\mathbb{E} = \emptyset$; likelyRenamings $\mathbb{L} = \emptyset$; unlikelyRenamings $\mathbb{U} = \emptyset$
2. Let possible renamings $\mathbb{P} = \{(w,x) | \nexists(w,\_) \in \mathbb{M} \wedge \nexists(\_,x) \in \mathbb{M} \wedge \exists u \in w.parents, v \in$
   $x.parents : (u,v) \in \mathbb{M} \vee (u,v) \in \mathbb{P}\}$;
3. Reorder $\mathbb{P}$: Leaf pairs $\rightarrow$ Inner node pairs $\rightarrow$ Mixed Pairs;
4. For all pairs $(w,x) \in \mathbb{P}$
   (a) If $similar(w,x) : \mathbb{L} = \mathbb{L} \cup \{(w,x)\}$;
   (b) Else $\mathbb{U} = \mathbb{U} \cup \{(w,x)\}$;
5. Let the user acknowledge and correct the renamings;
6. For each renaming pair $(x,y)$ the user acknowledged
   (a) $\mathbb{M} = \mathbb{M} \cup \{(w,x)\}$;
   (b) $\mathbb{E} = \mathbb{E} \cup \{RenameNode(w, x.name)\}$;
7. Return $\mathbb{E}$;
```

Fig. 3. Pseudocode for the calculation of the node renamings

a *likely renaming*, otherwise we call it an *unlikely renaming*. Of course, each node can only appear in one likely renaming. In case that a node is contained in more than one pair with adequate similarity, the pair with the highest similarity is chosen to be the likely renaming, all others become unlikely renamings. As each renaming results in a matching, when comparing inner nodes, likely renamings are treated as common leaves and contribute to the similarity of inner nodes.

This approach is a heuristic one. For instance, for a node that was renamed and attached to totally different parents, no renaming will be detected, but the node will remain unmatched, even after renaming detection. Thus, in production environments, the user will have to acknowledge or correct the detected renamings. For each acknowledged renaming (w, x), a $RenameNode(w, x.name)$ operation is appended to the edit script and immediately applied on v_{old}. The pseudocode for the renaming detection is shown in Fig. 3.

4.5 Comparing Two DAGs

After matching and renaming detection, we have all preliminaries for the change detection algorithm. This is split into five phases, each responsible for finding a particular set of operations. For the description of these phases, we need to introduce the *partner* of a node x, which is the node y to which x is matched. Thus x and y have to stem from different graphs. Each of the operations detected during the comparison is immediately applied to v_{old}. So, during the comparison v_{old} is transformed into v_{new}, and when finished, both graphs are identical.

Insert Phase. Let x be the current node when traversing v_{new} in topological order. If x is not matched yet, it must have been inserted. Let $\mathbb{Y}$ be the set of parents of x combined with the type of the edge to x. Then $\mathbb{Z}$ is the set of partners of the nodes in $\mathbb{Y}$, combined with the respective edge type. As we traverse the graph in topological order, we can be sure that every parent of x has already been visited and thus must have a partner in v_{old}. We now can easily create the appropriate $InsertNode(x.name, x.attributes, x.slots, \mathbb{Z})$ operation.

Update Phase. Let x be the current node when traversing v_{new} in topological order and w its partner. If the attributes of x and w differ, we append an $UpdateNode(w, x.attributes)$ operation to the edit script.

Slot Changing Phase. Let x be the current node when traversing v_{new} in topological order and w its partner. For every slot s_n contained in x but not in w, we append an $InsertSlot(w, s_n)$ to the edit script. For every slot s_o contained in w but not in x, we append an $DeleteSlot(w, s_o)$ to the edit script.

Edge Changing Phase. Let x be the current node when traversing v_{new} in topological order and w its partner. Let $\mathbb{Y}$ be the set of parents of x, $\mathbb{V}$ be the set of parents of w, each of them combined with the respective edge type from parent to child. We now have to check, whether every node in $\mathbb{Y}$ has a partner in $\mathbb{V}$ and vice versa, and whether all edges are of the correct edge type. For every edge e from $y \in \mathbb{Y}$ to x where the partner of y is not in $\mathbb{V}$, we append an $InsertEdge(y.partner, w, e.type)$ to the edit script. For every edge from $v \in \mathbb{V}$ to w where the partner of v is not in $\mathbb{Y}$, we append a $DeleteEdge(v, w)$ to the edit script. For every edge e_o from $v \in \mathbb{V}$ to w where exists an edge e_n from $v.partner \in \mathbb{Y}$ to x and $e_o.type \neq e_n.type$ we append an $ChangeEdgeType(v, w, e_n.type)$ to the edit script.

Delete Phase. Let w be the current node when traversing v_{old} in post-order. If w is not matched, it has been deleted. Thus, we append a $DeleteNode(w)$ operation to the edit script.

Complete Algorithm. Figure 4 shows the pseudocode for the complete change detection algorithm (compare [22]). The function $compareOntologies(v_{old}, v_{new})$ takes two versions of an ontology as input and returns an edit script $\mathbb{E}$, which represents the differences between them. First, the Matchings and Renamings are detected. Then, the Inserts, Updates, Edge and Slot changes can be found during one topological graph traversal. The delete phase needs one additional post-order traversal. Every operation that is appended to the edit script, is immediately applied to v_{old}, thus the graph is transformed to be equal to v_{new}.

4.6 Complexity Analysis

After having presented the complete algorithm, we now give a short complexity analysis. Let n be the number of nodes, n_l be the number leaves, n_i the number inner nodes in the graph and d the number of differences between the two version graphs. Let p be the average number of parents of a node and c be the average number of children of a node. Typically $d \ll n$, $p \ll n_i$ and $c \ll n$.

In the matching phase for each of the matching types (leaves, inner nodes, mixed), we have to do an LCS run, which is in $O(n \cdot d)$ and then compare the unmatched nodes with each other, which is in $O(d^2)$. Thus, number of node comparisons is in $O(n \cdot d + d^2)$. But as comparing two inner nodes requires building the intersection of the contained leaves sets, of course $similar(x, y)$ is

```
Function $compareOntologies(v_{old}, v_{new})$
1. Edit script $\mathbb{E} = \emptyset$;
2. Matching set $\mathbb{M} = calculateMatching(v_{old}, v_{new})$;
3. Renamings $\mathbb{R} = calculateRenamings(v_{old}, v_{new}, \mathbb{M})$;
4. $\mathbb{E} = \mathbb{E} \cup \mathbb{R}$;
5. Let $x$ be the current node in topological traversal of $v_{new}$;
   (a) If $x$ has no partner
       i. $\mathbb{Z} = \{(p.partner, type) | (p, x, type)$ is an edge in $v_{new}\}$;
       ii. $\mathbb{E} = \mathbb{E} \cup \{InsertNode(x.name, x.attributes, x.slots, \mathbb{Z})\}$;
       iii. $w = InsertNode(x.name, x.attributes, x.slots, \mathbb{Z})$ applied to $v_{old}$;
       iv. $\mathbb{M} = \mathbb{M} \cup \{(w, x)\}$;
   (b) Else
       i. Let $w$ be the partner of $x$;
       ii. If $w.attributes \neq x.attributes$ $\mathbb{E} = \mathbb{E} \cup \{UpdateNode(w, x.attributes)\}$;
       iii. If $commonSlotsRatio(w, x) \neq 1$
            A. For each slot $s_o$ in $w$ which is not in $x$: $\mathbb{E} = \mathbb{E} \cup \{DeleteSlot(w, s_o)\}$;
            B. For each slot $s_n$ in $x$ which is not in $w$: $\mathbb{E} = \mathbb{E} \cup \{InsertSlot(w, s_n)\}$;
       iv. Let $\mathbb{Y} = \{(y, type_1) | (u, x, type_1)$ is an edge in $v_{new}\}$;
       v. Let $\mathbb{U} = \{(u, type_2) | (u, w, type_2)$ is an edge in $v_{old}\}$;
       vi. For each pair $(y, type_1) \in \mathbb{Y}$ with $u = y.partner$, $(u, type_2) \in \mathbb{U}$ and
           $type_1 \neq type_2$: $\mathbb{E} = \mathbb{E} \cup \{ChangeEdgeType(u, w, type_1)\}$;
       vii. For each pair $(y, type_1) \in \mathbb{Y}$ with $u = y.partner$, $(u, \_) \notin \mathbb{U}$: $\mathbb{E} = \mathbb{E} \cup$
            $\{InsertEdge(u, w, type_1)\}$;
       viii. For each pair $(u.type_2) \in \mathbb{U}$ with $y = u.partner$, $(y, \_) \notin \mathbb{Y}$: $\mathbb{E} = \mathbb{E} \cup$
             $\{DeleteEdge(u, w)\}$;
6. Let $w$ be the current node in a bottom-up traversal of $v_{old}$;
   (a) If $w$ has no partner
       i. $\mathbb{E} = \mathbb{E} \cup \{DeleteNode(w)\}$;
       ii. Delete $w$ from $v_{old}$;
7. Return $\mathbb{E}$;
```

Fig. 4. Pseudocode for comparing two ontology graphs

in $O(n_l)$ for the inner nodes x and y. Thus, the complete matching phase is in $O(n_i \cdot n_l \cdot d + n_l \cdot d^2)$.

All nodes that could not be matched (bound by d) could have been renamed. This gives at most d^2 node pairs to be compared. For similarity determination of inner nodes we again have to compare the common leaves, which is in $O(n_l)$, thus renaming detection is in $O(n_l \cdot d^2)$.

The detection of the remaining graph differences (i. e. Inserts, Deletes, Updates, Edge and Slot changes) requires at least two graph traversals ($\in O(n)$). The number of edit operations to be found here is in $O(d)$. As for the Inserts and Edge Changes for each node its parents have to be checked, the overall complexity here is in $O(n \cdot p + d \cdot p)$.

The overall complexity for our algorithm is the combination of matching, renaming and difference detection and thus $O(n_i \cdot n_l \cdot d + n_l \cdot d^2) \cup O(n_l \cdot d^2) \cup O(n \cdot p + d \cdot p)$. As $O(n_l \cdot d^2) \subset O(n^2 \cdot d + n \cdot d^2)$, $O(n \cdot p + d \cdot p) \subset O(n^2 \cdot d + n \cdot d^2)$ and $O(n_i \cdot n_l \cdot d + n_l \cdot d^2) \subset O(n^2 \cdot d + n \cdot d^2)$, the overall complexity is $O(n^2 \cdot d + n \cdot d^2)$.

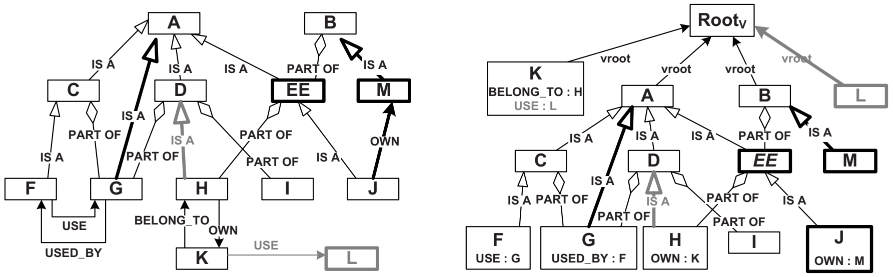

Fig. 5. Graph and RDAG representing the new ontology version

1. *RenameNode(E, "EE")* 2. *InsertNode("M", attributes, ∅, (B, IS_A))*
3. *InsertSlot(J, (OWN, M))* 4. *DeleteSlot(K, (USE, L)))* 5. *InsertEdge(A, G, IS_A)*
6. *RemoveEdge(D, H)* 7. *DeleteNode(L)*

Fig. 6. Edit script between the two versions

4.7 Example Edit Script

Remember the graph and the corresponding RDAG given in Fig. 1. In Fig. 5 we give a subsequent version of this ontology, represented again by the ontology graph and a corresponding RDAG. Elements which have been deleted between the two versions are depicted with strong grey lines. Elements which have changed or have been inserted are depicted with strong black lines. So, when looking at the left graph, we can see that the concept L and the generalization between D and H were deleted. We introduced a new concept M, which is a subconcept of B. Moreover, the concept G became a subconcept of A and a relation of type *own* was introduced between J and M. Finally the concept E was renamed to EE. The same changes can be seen in the RDAG in the right side of Fig. 5. When applying our change detection algorithm, the edit script shown in Fig. 6 is generated, including all differences between the two graphs.

4.8 Structure Versus Semantics

Our approach compares ontology versions solely on the syntactic level, i.e. it performs a structural comparison. With *structural comparison* in this context we address both: (1) Comparing the structure of the graph (i. e. the relations between concepts) and (2) comparing the structure of the concepts (i. e. their name and attributes). So we also have to discuss, whether this mere structural comparison can provide appropriate means to detect ontology changes.

If the semantics of an ontology changes between two versions, then the structure will change as well. Otherwise, there is no representation of the semantic changes and the changes can neither be detected by structural nor by semantic comparison algorithms. If the structure of an ontology (i.e. its representation) changes, then these structural changes might constitute semantic changes. There are cases where the structure is changed without an underlying semantic change,

e. g. for improving the representation for performance reasons, for better understandability, etc. So probably not all structural changes identified by our approach will also constitute semantic changes. Note that we do not compare two independently developed ontologies but rather two different versions of the same ontology. So the general difficulties of schema integration [27], where frequently the same semantical concepts differ considerable in their representations do not apply. This problem is covered by ontology matching techniques as e. g. [8,9,10].

We can conclude that every semantic change is represented by structural changes. Since semantic comparisons are usually more complex than structural analysis, we expect a facilitation and acceleration of identifying semantic changes if it can be restricted to the structural changes resulting from our change detection procedure. So the edit script which is the output of our algorithm can be input to a procedure for identifying and characterizing changes in the semantics.

5 Implementation and Evaluation

5.1 Evaluation Environment

For evaluating our approach we need two versions of an ontology to compare them. As we do not have enough ontologies with defined differences available, we simulated them by random DAG. The differences between two versions are also generated randomly.

The generated graphs have the following structure: As an ontology often describes – among other relations – a sort of a taxonomy we divide the graph into levels. A graph may have up to seven levels, depending on its size. After creating the nodes for each level, we define hierarchical relations between them. These hierarchical relations create a directed acyclic graphs. Each node can have one ($p = 0.7$), two ($p = 0.15$) or three ($p = 0.15$) parents from the parent or the grandparent level. Due to the layered structure of the graph, we easily can prevent creating cycles in the graph, by only creating edges from higher levels to lower levels. Cyclic graphs are represented by slots. About a quarter of the created nodes get up to five slots.

The number of generated differences is given as percentage of the number of nodes. Thus, for a graph with size 1000 and change rate of 10%, we generate an edit script consisting of about 100 operations. Within the generated edit script the operations are distributed as follows: Insert Node 10% Delete Node 5% Rename Node 5%, Update Node 10%, Insert Edge 15%, Remove Edge 15%, Insert Slot 15%, Remove Slot 15%, and Change Edge Type 10%. As the graphs are representing ontologies, and ontologies tend not to change very much [4], we tested our algorithm with difference rates of not more than 10%.

5.2 Evaluation Results

In the evaluation of the algorithm there are two major points to look at: *calculation time* and *correctness*. Figure 7 shows the average overall calculation

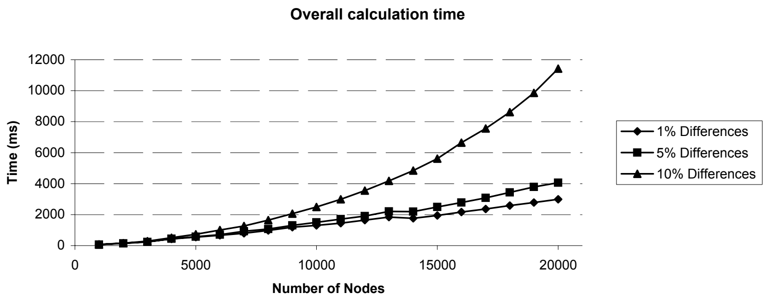

Fig. 7. Overall Calculation Time

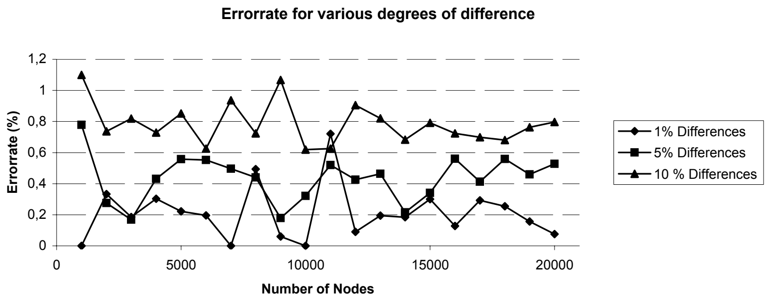

Fig. 8. Percentage of Errors in the Result

time for the algorithm for each difference rate. As expected from the complexity analysis, the chart shows quadratic complexity. Figure 8 shows the percentage of errors in the detected edit scripts with respect to the generated differences. This error rate does not only cover the absolute number of found edit operations but each operation is checked for its correctness against the differences created before the comparison. It can be seen that the error rate grows with the percentage of difference but does not strongly depend on the number of nodes in the graph. We also calculated the overall average error rates which are about 0.21% for 1% differences, 0.43% for 5% differences and 0.78% for 10% differences.

When taking a closer look on the generated errors, we see that many of the wrong operations are a direct result of errors in the matching and renaming detection. Because, if the algorithm does not match two nodes which represent the same concept or matches two nodes which do not represent the same concept, the algorithm will produce a series of operations (insert, delete, update, change edges and slots) for reestablishing what seems to be the correct structure. Thus, when using this approach in a production environment, we have to ask the user to acknowledge all detected matchings and renamings for correctness and therefore can foreclose many of the resulting errors.

6 Conclusion

We presented an approach for computing an edit script which explicitly represents the structural changes between two versions of an ontology. The approach is based on an efficient graph comparison algorithm.

Our approach can be used if only snapshots of an ontology are available but not a change history. It supports ontology administrators and ontology integrators to identify which changes have taken place between the two given versions. The result can, for an example, be applied to mark data where the underlying semantics changed. It can also provide the input for feeding a temporal ontology where several versions of an ontology together with the mappings between them are represented. Also semantic change analysis can be accelerated when it has to consider only the structural changes identified by structural comparison.

We tested the algorithm exhaustively, and honestly, we were surprised by its good performance figures, both in terms of time and precision.

Acknowledgements

We would like to thank Mark Musen (SMI, Stanford) who provided us a version history of the BioSTORM ontology [23] for our experiments.

The work on this project was partially supported by the project GATIB - Genome Austrian Tissue bank within the Austrian Genome Program GEN-AU, and by the EU Network of Excellence INTEROP.

References

1. Gruber, T.: A Translation Approach to Portable Onotology Specifications. Knowledge Acquisition 5(2) (1993)
2. Eder, J., Koncilia, C.: Modelling Changes in Ontologies. In: Proc. of On The Move - Federated Conferences. LNCS, vol. 3292, Springer, Heidelberg (2004)
3. Eder, J., Koncilia, C.: Interoperability in Temporal Ontologies. In: Proc. of the Open Interop Workshop on Enterprise Modelling and Ontologies for Interoperability (2005)
4. Noy, N., Musen, M.: PromptDiff: A fixed-point algorithm for comparing ontology versions. In: Proc. of the Nat'l Conf. on Artificial Intelligence (2002)
5. Klein, M., Fensel, D., Kiryakov, A., Ognyavov, D.: Ontology versioning and change detection on the Web. In: Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web, 13th Int'l Conf (2002)
6. Mostowfi, F., Fotouhi, F.: Change in Ontology and Ontology of Change. In: Proc. of Workshop on Ontology Management: Searching, Selection, Ranking, and Segmentation (2005)
7. Noy, N., Kunnatur, S., Klein, M., Musen, M.: Tracking Changes During Ontology Evolution. In: Proc. of the 3rd Int'l Conf. on the Semantic Web (2004)
8. Doan, A., Madhavan, J., Domingos, P., Halevy, A.Y.: Ontology matching: A machine learning approach. In: Staab, S., Studer, R. (eds.) Handbook on Ontologies, pp. 385–404. Springer, Heidelberg (2004)

9. Felicíssimo, C.H., Breitman, K.K.: Taxonomic ontology alignment – an implementation. In: Workshop em Engenharia de Requisitos. pp. 152–163 (2004)
10. McGuinness, D., Fikes, R., Rice, J., Wilder, S.: An environment for merging and testing large ontologies. In: Proc. of the 7th Int'l Conf. on Principles of Knowledge Representation and Reasoning. pp.483–493 (2000)
11. Eder, J., Wiggisser, K.: Detecting Changes in Ontologies via DAG Comparison. In: Proc. of the Open Interop Workshop on Enterprise Modelling and Ontologies for Interoperability (2006)
12. Eder, J., Koncilia, C., Wiggisser, K.: A Tree Comparison Approach to Detect Changes in Data Warehouse Structures. In: Proc. of the 7th Int'l Conf. on Data Warehousing and Knowledge Discovery. pp.1–10 (2005)
13. Eder, J., Wiggisser, K.: A DAG Comparison Algorithm and Its Application to Temporal Data Warehousing. In: Advances in Conceptual Modeling – Theory and Practice, ER Workshops 2006. pp.217–226 (2006)
14. Garey, M., Johnson, D.: Computers and Intractability – A Guide to the Theory of NP-Completeness. W.H. Freeman and Company, New York (1979)
15. Wang, J.T.L., Zhang, K., Chirn, G.W.: Algorithms for approximate graph matching. Information Sciences 82(1-2), 45–74 (1995)
16. Zhang, K., Wang, J., Sasha, D.: On the editing distance between undirected acyclic graphs. Int'l Journal of Foundations of Computer Science 7(1), 43–58 (1996)
17. Messmer, B., Bunke, H.: A new algorithm for error-tolerant subgraph isomorphism detection. IEEE Trans. on Pattern Analysis and Machine Intelligence 20, 493–505 (1998)
18. Shoubridge, P., Kraetzl, M., Ray, D.: Detection of abnormal change in dynamic networks. In: Proc. of Information Decision and Control, IEEE Inc. pp.557–562 (1999)
19. Cordella, L., Foggia, P., Sansone, C., Vento, M.: Perfomance evaluation of the vf graph matching algorithm. In: Proc. of the 10th Int'l Conf. on Image Analysis and Processing. pp.1172–1177 (1999)
20. Hlaoui, A., Wang, S.: A new algorithm for inexact graph matching. In: Proc. of the 16th Int'l Conf. on Pattern Recognition (ICPR'02) - Vol. 4 (2002)
21. Gori, M., Maggini, M., Sarti, L.: Exact and approximate graph matching using random walks. IEEE Trans. on Pattern Analysis and Machine Intelligence 27(7), 1100–1111 (2005)
22. Chawathe, S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. pp.493–504 (1996)
23. Crubzy, M., O'Connor, M., Buckeridge, D., Pincus, Z., Musen, M.: Ontology-centered syndromic surveillance for bioterrorism. IEEE Intelligent Systems 20(5), 26–35 (2005)
24. Myers, E.: An O(N D) Difference Algorithm and Its Variations. Algorithmica 1(2), 251–266 (1986)
25. Bergroth, L., Hakonen, H., Väisänen, H.: New Refinement Techniques for Longest Common Subsequence Algorithms. In: String Processing and Information Retrieval, Proceedings. pp.287–303 (2003)
26. Zhang, L.: On matching nodes between trees. Tech. Rep. 2003–2067, HP Labs (2003)
27. Halevy, A.Y.: Structures, semantics and statistics. In: Proc. of the 13 th Int'l Conf. on Very Large Data Bases.pp. 4–6 (2004)