Detecting Changes in Ontologies via 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] Alps Adria University Klagenfurt, Dep. of Informatics-Systems
wiggisser@isys.uni-klu.ac.at

Abstract. Ontologies are shared conceptualizations of a certain domain. As such domains may change, also changes in the ontologies have to be considered, as otherwise there is no way to state which knowledge was valid at an arbitrary point in time. Various ontology version management systems deal with this problem. But often, the differences between two versions of an ontology are not known, thus, it is not possible to incorporate changes into the versioning system. In this paper we present our graph based ontology change detection approach, an evaluation of the prototypic implementation and a comparison to existing change detection systems.

1 Introduction

According to Gruber [1], an ontology is *an explicit specification of a conceptualization*. Ontologies nowadays are often used to represent knowledge about a certain real world domain, e. g. curricula at a university. But as the real world tends to change, the ontologies have to change as well. In [2] we presented a graph based approach for ontology versioning. Incorporating changes in such a temporal ontology is easy if one knows the changes, but can be a very complex task if the differences are not exactly known. Furthermore, ontology development is a very federated process, thus changes are often made by many different people. Collecting and integrating all these changes is a highly complex task. In other situations, differences may be known, but the task of manually tagging them is very long lasting and error prone. In this paper we present our semiautomatic graph based approach for comparing two versions of an ontology.

There are some approaches for ontology comparison, e. g. PromptDiff [3], as part of the Protégé framework or OntoView [4], a web based system. We compare these approaches to our algorithm in Sect. 3.

2 Comparing Ontology Graphs

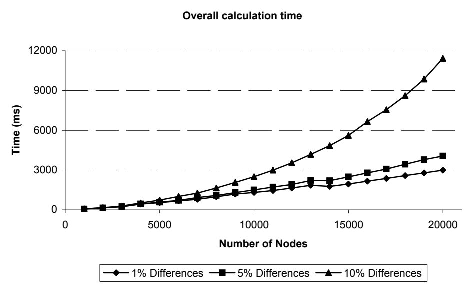
Generally speaking, ontologies can be represented by arbitrary graphs. The nodes of the graph represent the concepts and the edges represent the relations between the concepts. When looking more closely on such graphs, we can see that many ontologies comprise a generalization hierarchy, which typically builds a directed acyclic graph (DAG). Now comparing two DAG is much easier than comparing two arbitrary graphs. Thus, we based our ontology comparison on the change detection in rooted directed acyclic graphs (RDAG), which are DAG with exactly one node not having any parents.

We can obtain such a RDAG from any graph as follows: We define two types of edges for our ontology graph: *acyclic edges*, for which it is guaranteed that they do not create a cycle, for instance *is-a* or *part-of*. We call all other edges, which are not defined to be acyclic, *cyclic edges*. Such an edge can, for instance, be *is-friend-of*. If a single root does not exist yet, we insert a new node *virtualRoot* and connect it to all nodes not having any parents with an acyclic *virtualEdge*. To represent cyclic edges we use so called *slots*. These are attributes assigned to a node, holding the edge type and the node on the other end of the edge. Edges, which are transformed to slots are removed from the graph. When a node is disconnected from the graph during this process, we connect it to the *virtualRoot* via a *virtualEdge*. Thus, when finished, we have a RDAG representing exactly the same ontology as the initial graph.

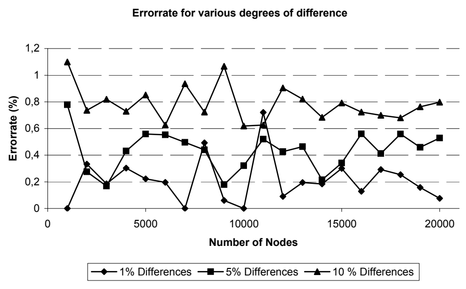
In our approach, the old and the new version of the ontology are represented each by one RDAG. We now want to find an edit script, which transforms the old graph into the new graph, hence holds a representation for the changes. We use the following operations for the graph: *Insert* and *Delete* of nodes, edges and slots, *Update* and *Rename* of nodes, and *Update* of edge type. Our RDAG comparison is inspired by the tree comparison algorithm of Chawathe et al. [5].

The node matching between the graphs is based on the nodes' names, their hierarchical position in the graph, i. e. wether the nodes are leaves or inner nodes, the contained slots, and node attributes, for instance a description of the concept. The renaming detection component is adapted from our previous work [6] in the area of Data Warehousing. Nodes, which are not matched, could have been renamed. As matching and renaming both touch the identity problem, they can only be handled using heuristics. Hence, the user has to acknowledge the results.

After matching and renaming has finished, the change detection takes place. Node inserts and updates, edge changes and slot changes can be detected during one topological traversal of the new version graph. Each node which is not matched yet has been inserted. If a node is matched, we compare it with its counterpart from the old version graph. If the attributes of matched nodes differ, an update has occurred. If two matched nodes have parents which do not match, edges have been inserted or deleted. And if matched nodes have different slots, deletes and inserts of slots have happened. Detecting node deletes needs one extra bottom up traversal of the old version graph, where every still unmatched node is considered as deleted. Each operation is added to the edit script and immediately applied to the old version. Thus, when the algorithm is finished, the old graph has been transformed to be equal to the new graph and the edit



(a) Overall Calculation Time



(b) Error Rate

Fig. 1: Evaluation Results

script contains a list of operations, representing the differences between old and new version.

3 Evaluation

3.1 Evaluation Environment

We implemented our approach with Java 1.5 under Windows XP. All performance tests were done on a Intel Pentium IV with 2.6 GHz and 1GB RAM, of which 512 MB were assigned to the Java Virtual Machine. For evaluating the graph comparison, we generated a set of random graphs, with size from 1 000 nodes up to 20 000 nodes. We tested our approach at difference rates of 1%, 5%, and 10%, meaning, for a graph with 10 000 nodes there were 100, 500, and 1 000 differences, respectively. The differences were also generated randomly. For each graph size and difference rate we did 20 rounds with newly created graphs and calculated the average time and error rate.

3.2 Evaluation Results

In the evaluation of the algorithm there are two major points to look at: *running time* and *correctness*. Figure 1a shows the average overall running time for the algorithm for various difference rates and graph sizes. It can be seen that difference rate has a greater impact on the running time than the graph size.

Figure 1b 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. 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.

3.3 Comparison to other Approaches

The first point to compare between the approaches is the quality of the heuristics, i. e. the error rate of the change detection. Noy and Musen did an evaluation for their PromtDiff algorithm, where they compared ontologies from size 320 to 1900 and a difference rates between 4.4% and 0.3%. They give the precision of their approach with 93%, i. e. 7% of the differences detected by PromptDiff where not correct. Unfortunately, we did not find any numbers on the precision of OntoView. Figure 1b shows the error rate of our approach, with respect to ontology size and difference rate. We can see, that the average error rate is far below 1%.

Second, we would like to compare the time complexity. Figure 1a shows the time calculation time for our approach, with respect to ontology size and difference rate. Unfortunately we did not find any numbers on time complexity, neither for OntoView nor for PromptDiff. So no comparison is possible here.

4 Conclusions

Ontologies holding knowledge from a real world domain are always subject to change. For incorporating such changes into an ontology versioning system, one has to know them. In this paper we presented our approach for detecting changes between two versions of an ontology. We described, how to obtain the rooted directed acyclic graph, needed for our purpose from an arbitrary graph, representing an ontology. Furthermore we presented a sketch of our comparison algorithm. The evaluation of our prototypic implementation gives promising numbers, which outrun the results from existing approaches, like PromptDiff.

References

1. Gruber, T.: A translation approach to portable onotology specification. Knowledge Acquisition **5** (1993) 1993
2. Eder, J., Koncilia, C.: Modelling changes in ontologies. In: Proceedings of On The Move - Federated Conferences, OTM 2004, Springer (2004) LNCS 3292.
3. Noy, N., Musen, M.: Promptdiff: A fixed-point algorithm for comparing ontology versions. In: Proceedings of the Nat'l Conf. on Artificial Intelligence. (2002)
4. 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 International Conference. (2002)
5. Chawathe, S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. (1996) 493–504
6. 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. (2005) 1–10