An Approach to Detection Ontology Changes

Michal Tury
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies, Slovak University of Technology
Ilkovičova 3, 842 16 Bratislava, Slovakia
misotury@gmail.com

Mária Bieliková
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies, Slovak University of Technology
Ilkovičova 3, 842 16 Bratislava, Slovakia
bielik@fiit.stuba.sk

ABSTRACT

Ontologies change and evolve both on the level of schema and individuals in order to meet requirements of changing world. Changes involve adding, deleting and modifying elements in the ontology both on structural (schema related) and content (individuals related) levels. The changes can lead to incorrect conclusions and can cause malfunction of systems, which use ontology data. In this paper we describe a proposal of the method for automated detection of currentness of presented data, including meta-data, which are represented by an ontology. Method comes out from a detection of changes between different versions of the ontology. Besides identifying modifications in the ontology, the purpose of the method lies also in identifying equivalent elements, thanks to which we are able to track information sources over time. We describe possibilities of automated identification of changes between ontologies using heuristic methods and their realization as a software tool called OntoDiff for comparing ontologies.

Categories and Subject Descriptors

H.5.4 [**Information Interfaces and Presentation**]: Hypertext/Hypermedia; D.2.9 [**Management**]: Software configuration management

General Terms

Algorithms, Design, Experimentation

Keywords

Ontology change, structural change, content change, heuristics, relative text comparison, OntoDiff software tool

1. INTRODUCTION

Nowadays, there exist various studies, which deal with a research in outdating and evolution of the web [3, 6]. According to the results of Online Computer Library Center [10], around 50 % of web pages become unavailable every year. We can assume that the semantic web evolution can progress with the same speed (or even faster) as the standard web. This is also the main reason why it is inevitable to research ways how to ensure that meta-data will be consistent with the content they describe for the whole time of their existence. Ontologies, which are corner stone the semantic web, consist of classes, properties, individuals and relations between these objects. The ontology, which is used for describing a part of the world, can be created by several people. Every human can model (and also perceive) the same part of the world in different ways. Even if the ontology is created only by one human, he can create different versions of the same ontology over time.

Consider for example two versions of an ontology, whereas the only difference between the ontology versions is in the name of a class, which is used by a software agent for information search in the ontology. This agent is developed to support only one version of the ontology, so it cannot work with a newer version, because it does not know the class identifier and is unable to identify particular individuals. But, if the agent knew differences or mappings between classes in different versions of the ontology, it would be possible for it to make the particular class or individuals accessible. In this manner, the agent could work also with other versions of the ontology, not only with ontologies it was created for.

Differences between two versions of an ontology could be simple, like using different names for classes or slots, using different data types and restrictions, but could also be rather complex, like changes in the class hierarchy or in data semantics when changing the application domain. Detecting the ontology changes can be used for checking currentness of data, which are delivered to the user. Let us consider stock exchange news or job offers. The ontology describing listed parts of the world can be – on the semantic level – still the same for certain period of time; it means that employer's salary or stock price always have the same meaning. But on the content level (including the instances), the ontology could change over time more frequently. These changes involve adding new job offers, modification of employer's salary data, changes in company's stock price, etc.

We describe an approach to automated detection of currentness of data (including meta-data) represented by an ontology. We proposed a method that comes out from the identification of changes between different versions of the ontology. Besides identifying modifications in the ontology versions, the purpose of the method lies also in identifying semantically equivalent elements, thanks to which we are

.

able to track information sources over time and reuse software components that work on defined data represented by the ontology in several applications where the ontology has changed. We describe heuristics proposed for automated detection of changes between the versions of an ontology and their validation by means of developed software tool called OntoDiff aimed at comparing ontologies.

The paper is organized as follows. Section 2 describes an overview of our approach to ontology changes detection. Next, we describe change detection between two ontologies at the structural level (Section 3) and also at the content level (Section 4). In Section 5 we present a method for relative text comparison, which is used for the comparison of the ontology versions individuals. Prototype implementation of presented approach – a software tool OntoDiff for comparison ontology versions – is described in Section 6. The paper concludes with a summary of our work and a brief discussion of future research.

2. OVERVIEW OF ONTOLOGY CHANGE DETECTION

Research in the area of ontology evolution covering change detection and management is currently highly active. There still remain open issues so appropriate tools for managing ontology evolution efficiently (from the point of human view, i.e. automatically) are still missing [4].

Some of existing approaches are devoted to automatic capture of changes [9] and some are more concerned with ontology evolution and change propagation [11]. Various classifications of ontology changes are described in literature, e.g. in [7] the authors consider elementary changes (modifications to one single ontology entity), composite changes (modifications to the direct neighborhood of an ontology entity) and complex changes (modifications to an arbitrary set of ontology entities). Similarly, the authors in [6] consider basic changes related to a single ontology entity and complex changes that group basic changes into logical units. These approaches consider primarily changes at the level of ontology schema definition. Detection of changes on both schema definition and ontology individuals are interconnected and it is important to cover both. Such view has been presented recently in [2] for measuring similarity within and between ontologies where three levels on which the similarity between two entities (concepts or instances) can be measured are defined: data layer, ontology layer, and context layer that cope with the data representation, ontological meaning and the usage of these entities, respectively.

We consider changes not only at the level of the ontology schema itself (we call them structural level changes alike in [8]), but also changes at the level of individuals (we call them content changes). In the case of structural level changes we inspect parts of the ontology which define classes, their slots, domains, ranges, relations between classes, restrictions, etc. Content level is represented by individuals of classes. Inspecting on the content level changes includes not only checkup of consistency of individuals with classes they are instances of, but also detecting changes in the context of classes, values of their slots, a comparison individuals between two versions, and identifying identical, modified, added and deleted individuals.

During the checkup of currentness of information two versions are inspected. Between these versions, it is necessary to identify changes and identify parts of versions which are identical (or similar); it means to determine the mappings between these parts. We understand the process of identifying changes between two versions as a process of finding modified, added and deleted parts.

Several approaches can be used for the change detection:

- finding identical elements in ontologies – the simplest approach based on comparing structures and objects and on finding compliances,
- heuristic methods – heuristics extend previous technique by rules, which help deduce new conclusions from imperfect or incomplete information on changes,
- methods of computational intelligence, like neural networks, genetic algorithms that are trained for particular cases.

Mentioned methods allow us to identify differences between two versions, whereas these differences can have diverse character. Figure 1 depicts an example of identified changes between two ontology versions on the structural level. The relation between a class and a subclass is represented with full arrow. Dashed arrows represent mapped classes; it means their semantics in both versions is considered equivalent. For example, the class *hardware* in version V_1 is mapped to the class *hardware* in version V_2. We can also see that mappings exist not only between identical classes, e.g., classes *IT* in version V_1 and *information technologies* in version V_2 have the same semantics, but their names are different. Class *software* in version V_1 is marked with a dagger because in reference to version V_2 this class is removed. Alike, classes *domain* and *networks* are marked with grey color, because in reference to version V_1, they are new in version V_2.

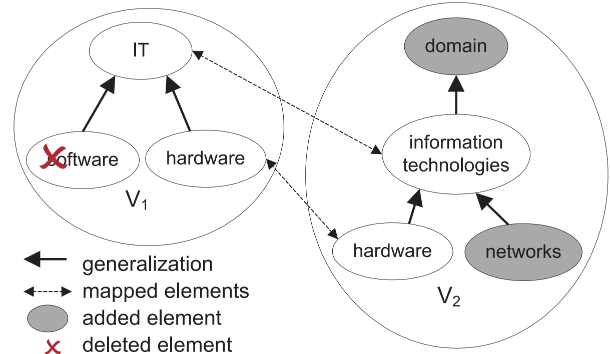

Figure 1: Example of detected changes between two ontology versions on the structural level.

Aside from the known terms like class, slot, individual, we use thereinafter the following:

- element – any part of an ontology for which we are usually looking for its equivalent,
- element type – is used to specify type of the element; it means that it defines if the element is class, slot, individual, etc.,
- equivalent element – element E_2 in version V_2, which has the same semantics as element E_1 in version V_1.

Following the example presented in Figure 1 it is possible to describe identified changes using three operations:

- *adding* – an element occurs in version V_2 and does not occur in version V_1;
- *mapping* – determines which element from version V_1 is semantically equivalent to the element in version V_2; this operation includes also modifying of an element while its semantics remains unchanged;
- *removing* – element occurs in version V_1 and does not occur in version V_2.

These operations can serve for storing differences between two versions of the ontology. It is matter of course that these operations should contain data, which identify versions of the ontology and also data for identifying elements of versions on which operations should apply.

In the case, we use the mapping or removing operation it is sufficient to identify the class, slot or individual. In the case of the adding operation we must also use additional meta-data for specifying added element. Descriptions of operations can be enriched with additional information like purpose, name of authors, time validity; those values have to be defined by a user. It is also desirable to note that the OWL language (commonly used for representing ontologies) supports storing information about class mappings.

Representation of changes can be realized on the ontology level, language level (used for the ontology representation), or on the level of specially defined change representation language. Change representation on the ontology level requires defining classes and slots in a way that we are able to specify directly in individuals which ontology elements are in relations with elements of another version (together with type of the relation). These definitions are part of the ontology and so they become a part of modeled domain indirectly. The main disadvantage of this approach is that the very definition of representation of changes can be liable to changes, so it is necessary to use another tool for finding and managing these changes as the case may be that undesirable infinite loops occur in the ontology.

For representing changes on the ontology representation language level it is possible to use the mapping support already defined in OWL. On the structural level this mapping can be realized using attributes *equivalentClass*, *equivalentProperty* and on the content level using the attribute *sameAs*. These attributes are supported from the OWL Lite version.

The third option is to define new format for describing changes. Because there are practically no limits we have highest count of possibilities in this case. So we can create a new XML based format, or binary format for higher effectiveness or use programming language class hierarchy and transform those classes into another shape. An example of such approach is CDL – Change Definition Language [11].

We define the process of changes identification in such a way that all elements in version V_1 (older version) are declared as removed and all elements in version V_2 (newer version) are declared as added. We look for mappings between elements in both versions. The process of the mappings identification results in the state where in old version removed and mapped elements are identified, and in new version mapped and added elements are detected. This also simulates the modify operation.

3. STRUCTURAL LEVEL CHANGES

The main objective of structural level change identification is to compare two versions of the ontology on the ontology schema level, identify equivalent elements (e.g., equivalent classes, equivalent class properties) and to find differences. These changes have to be submitted to deeper analysis, where it is necessary to find eventual relations between two versions. It means to find out if the change is adding, removing or modifying some element. Similarity between two versions can be expressed by the equivalence relation of their elements. The mapping between elements of two versions of the ontology is established and relationship between these elements is described. In the case of added or removed elements we have to create a description of this transformation.

To detect the changes on the structural level we defined and realized six heuristics for detecting equivalent parts of ontology versions motivated by [9, 8]. As it stands out from the nature of heuristics, the success cannot always be achieved by their application. The detection may be incorrect, i.e. it is possible that application of a heuristic will result to marking two classes as equivalent, but they are not actually semantically equivalent (as we work only on the structural level). That would not be such a big problem, because changes in these classes have to be identified and that way we can detect also internal differences. Despite of that overall semantics of classes can be different. In case this event will occur, the user himself will have to decide, if classes are or are not equivalent.

In the following description of heuristics, we use E for elements, V for versions and C for classes.

Elements with the same name

Two elements $E_1 \in V_1$ and $E_2 \in V_2$ are equivalent, if they have the same name and type. This heuristic describes standard situation, when we are looking for basic mappings between two ontology versions. Example of this kind of mapping can be seen in Figure 1 between classes *hardware* both in V_1 and V_2.

One nonequivalent subclass

If two classes $C_1 \in V_1$ and $C_2 \in V_2$ that are equivalent have exactly one subclass, which has no equivalent, then we mark these two subclasses as equivalent. This situation occurs for example in cases, when the name of a subclass has changed.

Several nonequivalent subclasses

Let $C_1 \in V_1$ and $C_2 \in V_2$, C_1 and C_2 are equivalent, $subC_1$ is a subclass of C_1 and $subC_2$ is a subclass of C_2, $subC_1$ has all slots equivalent with $subC_2$, whereas these subclasses have their slots different from other subclasses of their parent class, then $subC_1$ and $subC_2$ are equivalent.

Example illustrating this situation is depicted in Figure 2, where in version V_2 we have added into the class *IT* a new subclass *system administration*. We also have changed the name of one subclass from *hardware* in version V_1 to *hardware/networks* in version V_2. In this case, the class *IT* has two nonequivalent subclasses. Because classes *hardware* in version V_1 and *hardware/networks* in version V_2 have slots with the same names and classes have only different names, we consider them as equivalent. From this reasoning we can also deduce that the class *system administration* is a new class, and thus it represents added element.

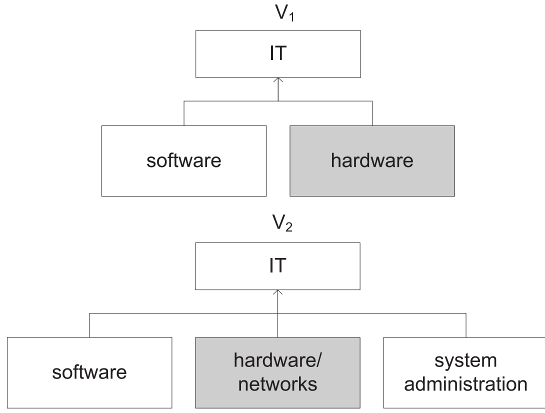

Figure 2: Example of several nonequivalent subclasses.

Change of names with the same prefix or suffix

If $C_1 \in V_1$ and $C_2 \in V_2$ are equivalent and all subclasses of C_1 have the same names with subclasses of C_2 except of constant prefix or suffix, then subclasses C_1 are equivalent with subclasses C_2.

Example of using this heuristic is depicted in Figure 3. Compared to version V_1, the prefix *'-IT'* is added to both subclasses of the class *IT* in version V_2.

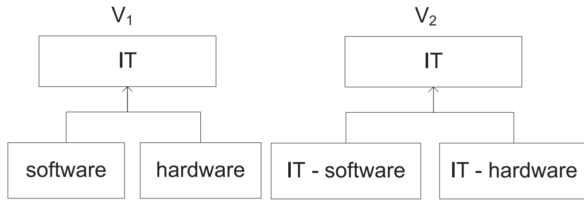

Figure 3: Example of name change by adding the same prefix.

This heuristic can be extended by the notion of "similar names", i.e. we consider equivalent subclasses if their names do not differ considerably (syntactically). For determining similarity we use relative text comparison method described in Section 5. Another extension is in detecting synonyms, which can discover semantically equivalent names of classes regardless their different names.

Nonequivalent parent class

If $C_1 \in V_1$ and $C_2 \in V_2$ have equivalent all subclasses, then C_1 and C_2 are equivalent.

Figure 4 illustrates this case, where the name of parent class has changed from acronym (*IT*) to full name (*information technologies*). However, all subclasses of these classes remained unchanged so it is possible to find the equivalence relation between them. This means that their parent classes can be declared as equivalent too.

One nonequivalent slot

If $C_1 \in V_1$ and $C_2 \in V_2$ are equivalent and both classes have exactly one nonequivalent slot, these slots are considered equivalent.

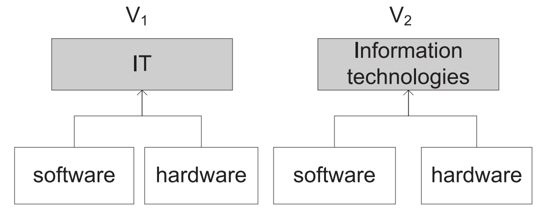

Figure 4: Example of nonequivalent parent class.

This situation occurs when the name of a slot in the class is changed.

According to [9], the simplest heuristic "elements with the same name" is applied in 97.9 % of all cases. Following this number, we can deduce that changes in subsequent versions of ontologies are normally small. From the perspective of effectiveness of change detection this fact should be considered as a very positive. It means that after applying this heuristic other heuristics will have to scan smaller part of the ontology version. On the other hand, it does not mean that other heuristics will seek only 2.1 % part of the ontology. The reason for this lies in fact that when using some heuristic (for example nonequivalent parent class) it is necessary to search also in some already mapped elements, for example to detect mapped subclasses. In spite of this fact we can assume that after searching for elements with the same name, the other heuristics will perform faster, because they will work on relative smaller set of elements.

Besides the way how heuristics determine equivalence between elements, serious impact on successful detection of changes has also the order in which the heuristics are applied on an ontology. As we said, the most frequently applied heuristic is "elements with the same name". So, if we apply this heuristic as first, we can radically reduce the set of classes which enter other heuristics. On the other hand, sooner application of heuristics, which work with more attributes, can lead to more accurate results. The reason is that heuristics can find eventual equivalence of elements on more precise level. Unfortunately, this approach is inefficient, because larger set of elements needs to be compared.

After detection of changes on the structural level, it is possible to further specify identified changes, which can help detecting changes on the content level (the level of ontology individuals). In the case of added or removed element on the structural level, we can accurately specify what should and should not be on the content level. For example, if we add a slot into the class, we know that in new version of the ontology the individuals of this class can have defined values of this new slot.

For easier detection of changes on structural level, we define also a level of element mappings. This level is represented as an attribute of mapping operation and determines if there were identified changes in internal structure of the element. In this way we are able to identify three levels of mapping:

- *without change* – structure of the element and also its name are identical in both versions,
- *isomorphic change* – structure of the element is identical in both versions, but names are different,

- *semantic change* – internal structure of the element is changed.

Mapping of slots has significant impact on values, which they contain in individual versions of the ontology. The value of a slot depends not only on its meaning, but also on the form of expressing this value. For example, if we would have a slot, which defines length, we should distinguish whether the length is given in miles, kilometers, feet or inches. In this case relations between values for each slot mapping should be defined separately. This relation can be defined for example as transformational function from value in old version into the value in new version of the ontology.

4. CONTENT LEVEL CHANGES

The main objective of checkup on the content level is to compare two versions of the ontology, identify equivalent parts, i.e. identical/similar individuals, identical/similar values of slots in individuals, etc. and to find changes between them. Similarly as on the structural level, these changes have to be exposed to a deeper analysis. The main purpose of this analysis is to find eventual relations between both versions. It means to determine whether it is an addition, deletion or modification of some part of the ontology together with an importance of the change.

Change detection on the content level depends on identified changes on the structural level. The basic element of the ontology, which serves as an input for changes detection on this level is a class individual. Because internal structure of the individual comes out from the internal structure of the class, which it is instance of, the identification of changes have to reflect identified changes on the structural level. We already know classes, which are new, deleted or equivalent, so we are able to achieve higher effectiveness of change identification as on the structural level.

Change detection on the content level uses similar heuristics as those used on the structural level. Individuals define values of class slots and these values can contain literals and also URIs of some resources. During the comparison of these values and evaluation of these comparisons the basic question emerges. It is known fact that in case of even the smallest mistake (e.g., typing error) in URI notation, the resource becomes unavailable or references to another, not intended resource (e.g., typing errors in parameters in GET method). In the case of literals (especially text literals), the situation is quite opposite. Here the typing error is generally not interesting for the user and the user does not apprehend its correction as change. The user can remark this error, but in most cases he will understand the original text and is not interested in emphasizing such change. Based on this, we distinguish the type and meaning of compared data.

Definition a mechanism for simple comparison of two literals is trivial. This type of comparison is supported practically in all modern programming languages. When using this type of comparison, we simply take two strings and find out whether they are identical.

We proposed a comparison mechanism that takes into account a threshold value determining the number of differences between two strings. If the number of differences stays under defined threshold value, we can declare these two strings as similar (or relatively identical). We allow users to set up the threshold value, and give them also possibility to visualize in readable form reasons why two strings were declared as relatively identical or not relatively identical.

The question is how we should set the granularity of differences used for relative conformity decisions. Is it characters, words, or paragraphs? It is considerable difference, if we would compare words with threshold value of two characters and whole paragraphs with the same threshold value. The simplest solution is setting the number of characters considered. Another aspect is how "close" are changes situated in text. Thanks to this aspect, we are able to evaluate importance of changes. Another heuristic in this sense is not considering added or removed diacritic marks in text, eventually not considering case sensitivity.

The objective of change detection on the content level in an ontology version is to find added, removed and modified individuals. In so doing, it is also necessary to take into account changes on the structural level, for example changes in slot names, classes, or slot additions. One of the most important objectives is identification of correct mappings of equivalent individuals between two versions. If we are able to define these mappings, we can define which slots and how much have been changed and which data are current.

To identify mappings of individuals between two ontology versions, we proposed three heuristics. In the following description of heuristics we use I for individuals, V for versions and C for classes.

Individuals with the same name

Let two individuals I_1 from class C_1 and $C_1 \in V_1$ and I_2 from class C_2 and $C_2 \in V_2$ and C_1 and C_2 are equivalent. Then I_1 and I_2 are equivalent, if they have identical names.

This heuristic describes standard situation when searching for mapping of individuals between two versions. It is not applicable on anonymous individuals.

Identical values of slots

Let two individuals I_1 from class C_1 and $C_1 \in V_1$ and I_2 from class C_2 and $C_2 \in V_2$ and C_1 and C_2 are equivalent. Then I_1 and I_2 are equivalent, if they have relative identical values of slots and the mapping between classes C_1 and C_2 was detected.

This heuristic is applicable for identification of equivalent anonymous individuals. But it can happen that classes will contain not enough equivalent slots to make decision about individuals similarity. This problem can be partially solved by defining minimal number of slots, which must be mapped in classes C_1, C_2 or minimal number of defined values in individuals I_1, I_2.

One nonidentical value of slot

Let two individuals I_1 from class C_1 and $C_1 \in V_1$ and I_2 from class C_2 and $C_2 \in V_2$ and C_1 and C_2 are equivalent. Then I_1 and I_2 are equivalent, if they have identical values of slots, except one, and the mapping between classes C_1 and C_2 exist.

This heuristic is applicable for identification of equivalent anonymous individuals with exactly one changed slot value. Like in previous heuristic, it is desirable to define minimal number of slots, which will be analyzed.

Described heuristics are used for identification of equivalence between individuals. Slots which have discovered a mapping on the structural level are exposed to additional analysis, where the differences on the slot level, respectively their values are identified. The reason is that only for these

slots we are able to acquire values which is reasonable to compare. The comparison follows the preposition that the user knows, eventually will see, which data have changed and what has been changed.

5. RELATIVE TEXT COMPARISON

In order to detect relative equivalence of two literals, we proposed a method for text differences identification. We identify the differences that are supposed to be semantically significant. For the user it means that only "significant" text changes will be visualized and minor changes like typing errors will be omitted. Consequently, in new version, such text will not be declared as different concerning the text present in old version. Our method works on the level of characters, words, sentences or the whole text, respectively on a combination of these levels. We denote a character, word, sentence or text as text element thereinafter.

The basic principle of the method is to detect whether two text elements are similar (or relatively identical). Detection of the relative identity is based on defined threshold values, which involve the number of detected syntactic changes, eventually their density.

The process of determination of relative conformity is based on an initial comparison of text elements using `diff` algorithm [5]. Application of this algorithm on two versions of texts detects a vector of identical elements, a vector of removed elements from old version and a vector of added elements in new version. Because in some cases, the `diff` is unable to identify a moved element, in such case the moved elements are detected using the vector of added elements and the vector of removed elements. This is accomplished using relative comparisons of element on the lowest level (comparing characters, see Figure 5). For example, in case of moved sentence, we try to find the sentence by detecting identical words between two versions. In case of moved word, we look for words with same characters, etc.

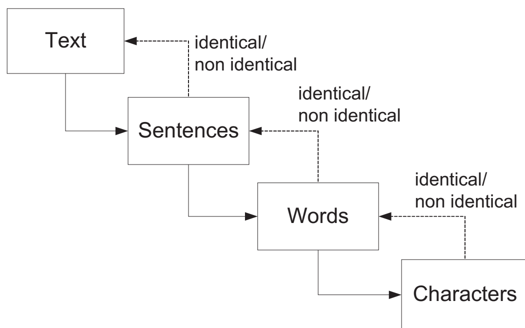

Figure 5: Different levels for text comparison.

In the case of whole text comparison, we decompose text to individual paragraphs first. This step is not necessary, but it has an effect on accuracy and efficiency of moved sentences detection. Text decomposition results in the situation where we search for sentences only in the paragraph. Without the text decomposition we would have to search for sentences in the whole text, what can have great impact on comparison effectiveness. Consequently the analysis continues on the sentence level.

During relative comparisons of sentences we analyze words in the sentence. Individual words between two versions are compared one by one on the character level and the frequency of changes is logged. In the case of different lengths of compared texts, the redundant elements in one word are marked as nonequivalent with corresponding characters in the other word (because they do not exist here) and frequency of changes is increased by this difference.

Following frequency of differences we estimate whether particular text unit contains a semantic change or not. Let us denote this value as *similarity decision* on the level of particular unit. Decision is made following defined threshold values. Threshold values are not generally defined for any unit lengths, but for zones. Considering words, the zone represents the scope of word lengths, for which the given threshold value is valid. For example for words shorter than two characters including, the threshold value should be 100 % of changed characters (i.e., the words should be identical), for words shorter than 5 characters including, the threshold value can be 30 % and so on.

It is desirable to have the choice to identify the threshold values not only as relative quantity, but also as absolute quantity. Suitable complement for determination of threshold values is creation of profiles for individual types of text. Using profiles, we can for example achieve that comparisons on technical texts or law texts would be set as more sensitive than for example in poetry, where the impact of changes on meaning is not so important. The presented way of determination considers with a certain inaccuracy or variance. Here experimental determination of the zones would be beneficial.

After acquiring particular decisions on comparisons, i.e. whether individual elements do or do not contain semantic change, relative conformity is analyzed on higher level with the same approach.

As an example consider comparison of two versions of the text containing one sentence on Figure 6. The difference between the versions is in replacement of the word "Pat" by the word "Jane". Moreover in old version there is a misprint in the word "old". Comparison of replaced words on the level of words will result into unmatched words. However, the words "old" and "olr" are evaluated as similar.

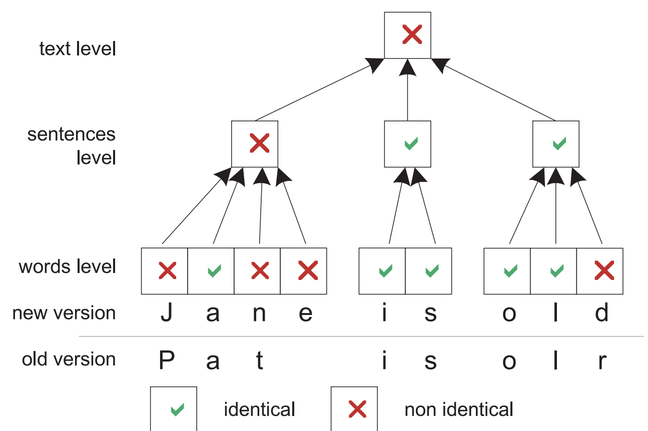

Figure 6: Example of text comparison.

For some languages diacritic marks can have great impact on the results of comparison. Comparison of identical texts with and without diacritic marks would lead to determining texts as different. This problem can be solved using lesser

values for threshold values or by removing diacritic marks from the text during the comparison.

Presented method works on syntactic level. It can be extended by incorporating synonymous dictionary and improve the text analysis both on structural and content levels of ontology changes identification.

6. ONTODIFF SOFTWARE TOOL

In order to validate feasibility of proposed approach, especially defined heuristics, we developed software tool OntoDiff [12]. The main objective of OntoDiff was to prototype the method for ontology change identification in order to validate functionality and behavior of heuristics on various patterns of ontologies. Our user interface for displaying changes between two individuals was inspired by the interface that Microsoft Word uses to present changes similarly as in [8].

OntoDiff allows storing versions of the ontology and identified changes into the repository (see architecture of the system on Figure 7). It contains module for version management, module for management of changes between versions, allows using automated or manual identification of changes on the structural level. The tool visually marks identified changes. OntoDiff is implemented as web application on platform J2EE using Struts framework[1] and iBatis[2]. For manipulation with ontologies we used Jena framework[3].

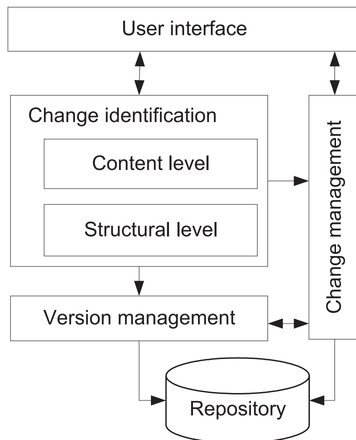

Figure 7: OntoDiff architecture.

Figure 8 illustrates an example of user interface of the OntoDiff. OntoDiff enables semi-automatic ontology change identification where the system suggests mappings according defined heuristics and the user can control the whole process by manual setting of added, moved or removed elements.

For testing purposes we used three ontologies, each of them containing 10 to 15 versions with different types of changes and with various scopes. We used threshold values obtained by series of experiments for various texts. The values are set for each level (words, sentences, whole text) separately. Table 1 gives values of thresholds for zones used on the level of words.

[1] Apache Software Foundation: `http://struts.apache.org`
[2] Data Mapper Framework: `http://www.ibatis.com`
[3] Jena – Semantic Web Framework for Java: `http://jena.sourceforge.net`

Zone	Threshold [%]
2 characters	51
5 characters	33
10 characters	21
$>$ 10 characters	20

Table 1: Example of threshold values definition.

The system was able to identity smaller and middle large changes in the ontology with high reliability. These small or middle large changes have not involved any cardinal movements and renames of whole parts, or brands in class structure of ontology.

In the cases of ontology version with a lot of conceptual changes, successfulness of identification of mappings between classes was reduced and the OntoDiff was practically able to identify only classes and slots with identical names. In these cases, a user should help the system to identify changes manually. For the OntoDiff it is enough to find "footings", i.e., it is necessary to fulfill conditions for heuristics. After fulfilling these conditions, the system was able to identify consequent changes, again.

7. CONCLUSIONS

This paper deals with change detection in versions of ontologies. The base of our solution resides in using heuristics for searching elements that can be marked as equivalent (identical or similar) in ontology versions. To make it possible to detect changes on the level of individuals, change detection on the ontology schema level should be performed. Main contribution of our work presented here is proposal of methods for comparison of ontology versions on content level, along with detecting of equivalence of anonymous individuals. In eventuality, all of the identified changes can be used for validity checkup of slot values of individuals, i.e., the content which is presented to the user.

Benefit of proposed approach can be seen for example for development of software agents processing information within the context of semantic web technologies. The agents usually support only one version of an ontology. If the agents would know the mapping between versions, they will be able to work with another version of the ontology, they were originally not designed for.

Developed software tool OntoDiff offers good basis for next development of change identifications on content level of the ontology. It is aimed for using within corporate memory maintenance in larger project [1]. It is possible to use it for building the solution for distribution of changes between two ontologies or for online identification, which can be used by other systems. All of that can be used employing popular technology of web services.

Presumably the most interesting extension of the system is its extension in the way that it will be more oriented to end consumer of the content. Such system can store versions of the ontology in defined time slots. Storing can be manual, using files with the ontology versions, or automated, when relevant data are downloaded from the Web. After that the system analyzes structural and content level changes. Following the identified changes, it can effectively present changes as tracked information from particular domain and mark detected changes to the end user. In conjunction with adaptive web-based systems this information can be served

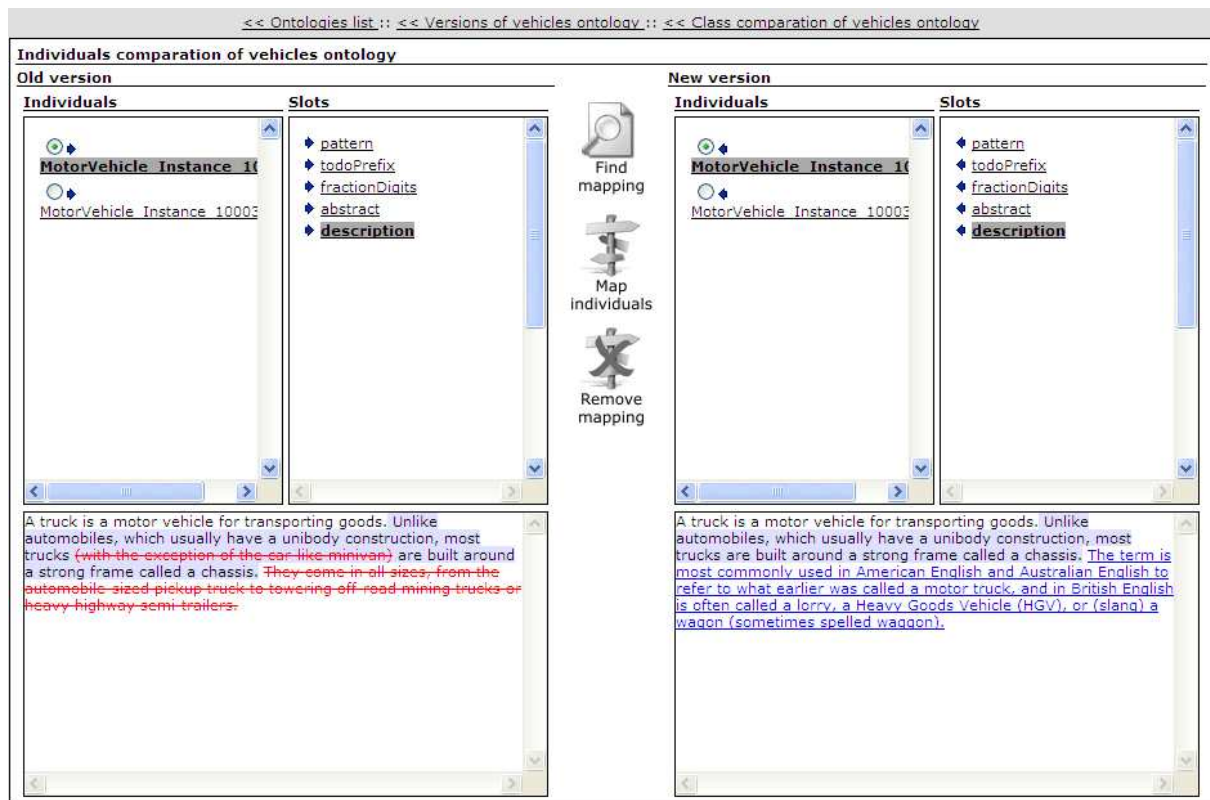

Figure 8: Example of OntoDiff screen.

the end user matching his demands and needs in such a way that only similar individuals of presented ontology are offered to particular user.

8. ACKNOWLEDGMENTS

This work was partially supported by the Science and Technology Assistance Agency under the contract No. APVT-20-007104 and by the Scientific Grant Agency of Slovak Republic, grant No. VG1/3102/06.

9. REFERENCES

[1] M. Ciglan, M. Babik, M. Laclavik, I. Budinska, and L. Hluchy. Corporate memory: A framework for supporting tools for acquisition, organization and maintenance of information and knowledge. In J. Zendulka, editor, *Proc. of 9th Int. Conf. on Information Systems Implementation and Modelling (ISIM 2006)*, pages 185–192. MARQ, Ostrava, 2006.

[2] M. Ehrig, P. Haase, M. Hefke, and N. Stojanovic. Similarity for ontologies - a comprehensive framework. In *Proc. of 13th European Conf. on Inf. Systems*, 2005.

[3] D. Fatterly, M. Manase, M. Najork, and J. Wiener. A large-scale study of the evolution of web pages. *Software—Practice and Experience*, 34(2):213–237, 2004.

[4] P. Haase and Y. Sure. State of the art on ontology evolution, 2004. Available at `www.aifb.uni-karlsruhe.de/WBS/ysu/publications/SEKT-D3.1.1.b.pdf`.

[5] J. Hunt and M. McIlroy. An algorithm for differential file comparison. Bell Telephone Labs CSTR #41, 1976. Available at `www.cs.dartmouth.edu/~doug/diff.ps`.

[6] M. Klein, D. Fensel, Dieter, A. Kiryakov, and D. Ognyanov. Ontology versioning and change detection on the web. In *13th Int. Conf. on Knowledge Engineering and Knowledge Management (EKAW02)*, page 197. Springer, LNCS 2473, 2002.

[7] A. Maedche, L. Stojanovic, R. Studer, and R. Volz. Managing multiple ontologies and ontology evolution in ontologging. In Y. G. et al., editor, *Proc. of the Conf. on Intelligent Information Processing (IIP-2002)*, pages 51–63. Montreal, Canada, 2002.

[8] N. Noy, S. Kunnatur, M. Klein, and M. Musen. Tracking changes during ontology evolution. In *Proc. of Int. Semantic Web Conference (ISWC 2004)*, pages 259–273, 2004.

[9] N. Noy and M. A. Musen. Ontology versioning as an element of an ontology-management framework. *IEEE Intelligent Systems*, 2003.

[10] OCLC – Online Computer Library Center. Web characterization. Available at `http://wcp.oclc.org`.

[11] P. Plessers and O. D. Troyer. Ontology change detection using a version log. In *ISWC 2005*, pages 578–592. Springer, LNCS 3729, 2005.

[12] M. Tury. Identification and management of onology changes. Master's thesis, Slovak University of Technology in Bratislava, 2005.