Ontology versioning and change detection on the Web

Michel Klein[1], Dieter Fensel[1], Atanas Kiryakov[2], and Damyan Ognyanov[2]

[1] Vrije Universiteit Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam, the Netherlands
{michel.klein|dieter}@cs.vu.nl
[2] OntoText Lab., Sirma AI Ltd.
38A Hristo Botev blvd., Sofia 1000, Bulgaria
{naso|damyan}@sirma.bg

Abstract. To effectively use ontologies on the Web, it is essential that changes in ontologies are managed well. This paper analyzes the topic of ontology versioning in the context of the Web by looking at the characteristics of the version relation between ontologies and at the identification of online ontologies. Then, it describes the design of a web-based system that helps users to manage changes in ontologies. The system helps to keep different versions of web-based ontologies interoperable, by maintaining not only the transformations between ontologies, but also the conceptual relation between concepts in different versions. The system allows ontology engineers to compare versions of ontology and to specify these conceptual relations. For the visualization of differences, it uses an adaptable rule-based mechanism that finds and classifies changes in RDF-based ontologies.

1 The Web needs change management for ontologies

The envisaged next generation of the Web (called Semantic Web [6]) will consist of data defined and linked in such a way that it can be used for more effective discovery, automation, integration, and reuse across various applications[3]. In this vision, ontologies have an important role in defining and relating concepts that are used to describe data on the web. However, the distributed and dynamic character of the web will cause that many versions and variants of ontologies will arise. Ontologies are often developed by several persons and continue to evolve over time. Moreover, domain changes, adaptations to different tasks, or changes in the conceptualization might cause modifications of the ontology. This will likely cause incompatibilities in the applications and ontologies that refer to them and will give wrong interpretations to data or make data inaccessible [14].

To form a real Semantic *Web*, it is necessary that the knowledge that is represented in the different versions of ontologies is interoperable. It is therefore important to create links between ontology versions that specify how the knowledge in the different versions of the ontologies is related. These links can be used to re-interpret data and knowledge under different versions of ontologies.

[3] http://www.w3.org/2001/sw/Activity

In this paper, we present various elements of a methodology for ontology versioning. We describe a method to specify relations between versions of ontologies and we also propose an identification scheme for ontologies. We then present a web-based system that supports the user in specifying the relations between ontology versions. The system, called OntoView, can also be used store ontologies and to provide a transparent interface to different versions. The goal of this system is not to provide a central registry for ontologies, but to allow ontology engineers to store their versions and variants of ontologies and relate them to other (possibly remote) ontologies. The resulting mapping relations between versions can also be exported and used outside the system.

The rest of the paper is organized as follows. In the next section, we analyze the characteristics of the relation between different versions of ontologies. Section 3 contains a discussion of ontology identification and proposes a identification scheme for ontologies. In section 4, we give an overview of the versioning support system and describe its the main functions. Section 5 describes the main feature of the system: comparing ontologies. In that section, we explain the mechanism we used to find changes in RDF-based ontologies and present some of the rules that we used to encode change types. We discuss some open issues in section 6, and we conclude the paper in section 7.

2 Characteristics of a version relation

There are three important aspects to discuss when considering an version relation between ontologies. First, this is **the difference between version relations and conceptual relations inside an ontology**.

Ontologies usually consist of a set of class (or concept) definitions, property definitions and axioms about them. The classes, properties and axioms are related to each other and together form a model of a part of the world. A change constitutes a new version of the ontology and also a *version relation* between the definitions of concepts and properties in the original version of the ontology and those in the new version.[4]

The relations between concepts inside an ontology, e.g. between class A and class B, are thus fundamentally different from the version relations between two versions of a concept, e.g. between class $A_{1.0}$ and class $A_{2.0}$. In the first case, the relation is a purely conceptual relation in the domain of interest; in the second case, however, the relation describes meta-information about the change of the concept.

Nevertheless, two versions of a concept still have *some* conceptual relation. In other words, although the update relation itself is not a conceptual relation, the participating versions of a concept (e.g. $A_{1.0}$ and $A_{2.0}$) do have a particular conceptual (logical) relation to each other.

Altogether, we distinguish the following properties of an version relation:

– **transformation** or **actual change**: a specification of what has actually changed in an ontological definition, specified by a set of change operations (cf. [1]), e.g., change of a restriction on a property, addition of a class, removal of a property, etc.;
– **conceptual relation**: the relation between constructs in the two versions of the ontology, e.g., specified by equivalence relations, subsumption relations, or logical rules;

[4] Except for removals and additions of classes and properties, of course.

– descriptive meta-data like **date**, **author**, and **intention** of the update: this describes the when, who and why of the change;
– **scope**: a description of the context in which the update is valid. In its simplest form, this might consist of the date when the change is valid in the real world, conform to *valid date* in temporal databases [18] (in this terminology, the "date" in the descriptive meta-data is called *transaction date*). More extensive descriptions of the scope, in various degrees of formality, are also possible.

A well-designed ontology change specification mechanism should take all these characteristics into account.

Another issue to discuss about ontology updates is the **possible discrepancy between changes in the specification and changes the conceptualization**. We have seen that an ontology is a *specification* of a *conceptualization*. The actual specification of concepts and properties is thus a *specific representation* of the conceptualization: the same concepts could also have been specified differently. Hence, a change in the specification does not necessarily coincide with a change in the conceptualization [14], and changes in the specification of an ontology are not per definition ontological changes.

For example, there are changes in the definition of a concept which are not meant to change the concept itself: attaching a slot "fuel-type" to a class "Car". Both class-definitions still refer to the same ontological concept, but in the second version it is described more extensively. Theoretically, the other way around is also possible: a concept could change without a change in its specification. However, this usually means that the concept is badly modelled.

It is important to distinguish changes in ontologies that affect the conceptualization from changes that don't. In [19] the following terms are used to make this distinction:

– **conceptual change**: a change in the way a domain is interpreted (conceptualized), which results in different ontological concepts or different relations between those concepts;
– **explication change**: a change in the way the conceptualization is specified, without changing the conceptualization itself.

It is important to notice that it is not possible to determine automatically whether a change is a conceptual change or a explication change. This requires insight in the conceptualization, and is basically a decision of the ontology engineer. However, heuristics can be applied to suggest the effects of changes. We will discuss that later on.

A third, somewhat different, aspect of an update is the **packaging of changes**, i.e., the way in which updates are applied to an ontology. This is an important practical issue for the development of an ontology change management system.

We can distinguish two different dimensions with respect to the packaging of the change specification. One dimension is the *granularity* of the specification: this can be either the level of a single "definition" or the level of a "file" as a whole.

The second dimension is the *method* of specification. There are several methods thinkable:

– a "transformation specification": an update specified by a list of change operations (e.g., add A, change B, delete C);

– a “replacement”: an update specified by replacing the old version of a concept or an ontology with a new version; this is an implicit change specification;
– a “mapping”: an update specified as a mapping between the original ontology and another one. Although this is not a update in the regular sense, an explicit mapping to another ontology can be considered as an update to the viewpoint of that ontology.

This gives several possible change specifications. For example, a change can be specified individually, as a mapping between one specific definition in one ontology and another definition in another ontology, but it can also be done at a file level, by defining the transformation of the ontology.

Notice that the packaging methods are not equivalent, i.e., they do not give the same information about the update relation. It is clear that the mapping provides a conceptual relation between versions of concepts, something that is not specified in a transformation.

3 Ontology identification on the web

Identification of versions of ontologies is very important. Ontologies describe a consensual view on a part of the world and function as reference for that specific conceptualization. Therefore, they should have a unique and stable identification. A human, agent or system that conforms to a specific ontology, should be able to refer to it unambiguously. We will now discuss the major issues of ontology identification on the Web, and outline an identification mechanism.

3.1 Identity of ontologies

The first question that has to be answered when we want to identify versions of an ontology on the web is: what is the identity of an ontology? This is not as trivial as it seems. For example, one could ask whether an update of a natural language description changes the identity of an ontology. If one regards a specific *specification* of a conceptualization as an essential characteristic of an ontology, then every modification to that specification forms a new version of the ontology. In that case, the descriptions specify different concepts, which are *per definition* not equal.

Looking at this from another perspective, one might regard an ontology primarily as a conceptualization, which is represented as complete as possible in a specification. In this case one could argue that an update to a natural language description of a concept is not a conceptual change, but just a more precise description of the same conceptualization. This would be an example of an explication change: the specification is changed, but the concept that is described remains the same.

In this philosophical debate, we take the following (practical) position. We assume that an ontology is represented in a file on the web. Every change that results in a different character representation of the ontology constitutes a revision. In case the logical definitions are not changed, it is the responsibility of the author of the revision to decide whether this revision is conceptual change and thus forms an new conceptualization with its own identity, or just an change in the representation of the same conceptualization.

If we relate this to the distinction between conceptual changes and explication changes, this means that whenever there has been a conceptual change in an ontology, it gets a new identifier. In case of explication changes, the ontology keeps the same identifier if and only if these changes were non-logical changes (thus, changes in the natural language description). This is summarized in table 1. Again, note that it is up to the ontology engineer to decide whether a change is a conceptual change or not.

	logical	non-logical
conceptual	new	new
explication	new	unchanged

Table 1. Change types and their effect on the identity of an ontology.

3.2 Identification on the web

The second question is: how does this relate to web resources and their identity? To answer this question, we have look at identification mechanisms on the web (i.e. URIs, URNs and URLs) and see how we can use them for the identification of the "entities" in our domain (i.e., the entities in the domain of ontology versions, e.g. a conceptualization, a revision, a specification).

Things on the web are called "resources" in the W3C[5]-terminology. According to the definition of Uniform Resource Identifiers (URI's) (defined in [5]), "a resource can be anything that has identity". In [7] is stated: "a 'resource' is a conceptual entity (a little like a Platonic ideal)". Both definitions comprise our idea of an ontology. Hence, an ontology can be regarded as a resource. An URI, which "is a compact string of characters for identifying an abstract or physical resource" [5] can be used to identify the resources. Notice that URI's provide a general identification mechanisms, as opposed to Uniform Resource Locators (URL's), which are bound to the *location* of a resource.

Usually, the XML Namespace mechanism [8] is used for the identification of web-based ontologies. This means that an ontology is identified by a URI. In practice, people tend to use a URL for this. In other words, they couple the identity of an ontology with the location of the ontology file on the web. The important step in our proposed method is to separate the identity of ontologies completely from the identity of files on the web that specify the ontology. In other words, the class of ontology resources should be distinguished from the class of file resources. As we have seen above, a revision — which is normally specified in a new file — *may* constitute a new ontology, but this is no automatism. Every revision is a new file resource and gets a new file identifier, but does not automatically get a new ontology identifier. If a change does not constitute a

[5] The standardization body for the World Wide Web

conceptual change, the new version gets a new location, but does not get a new identifier. For example, the location of an ontology can change from "../example/1.0/rev0" to "../example/1.0/rev1", while the identifier is still "../example/1.0".

3.3 Baseline of an identification method

When we take into account all these considerations, we propose an identification method that is based on the following points:

– a distinction between three classes of resources:
  1. files;
  2. ontologies;
  3. lines of backward compatible ontologies.
– a change in a file results in a new file identifier;
– the use of a URL for the file identification;
– a change in the conceptualization or in the logical definition results in a new ontology identifier, but a non-logical explication change doesn't;
– a separate URI for ontology identification with a two level numbering scheme:
  - minor numbers for backward compatible modifications (an ontology-URI ending with a minor number identifies a specific ontology);
  - major numbers for incompatible changes (an ontology-URI ending with a major number identifies a line of backward compatible ontologies);
– individual concepts or relations, whose identifier only differs in minor number, are assumed to be equivalent;
– ontologies are referred to by an ontology URI with the according major revision number and the *minimal extra commitment*, i.e., the lowest necessary minor revision number.

The ideas behind these points are the following. As already pointed out in the beginning of this section, the distinction between ontology identity and file identity has the advantage that file changes and location changes (e.g., copy of an ontology) can be isolated from ontological changes. By using a separate URI, it is possible to encode all the information in it that is necessary for our usage, and it also prevents confusion with URL's that specify a location.

The distinction between individual ontologies on the one hand and lines of backward compatible ontologies on the other hand, provides a simple way to indicate a very general type of compatibility, likewise the "BACKWARD-COMPATIBLE-WITH" field in SHOE [13]. The distinction we make is also in line with the idea of "levels of generality", which is discussed in [7]. Applications can conclude directly — without formal analysis or deduction steps — that a version can be validly used on data sources with the same major number and a equal or lower minor number. To achieve a maximal backward compatibility, we also propose that not the minor number of the newest revision is specified in a data source, but the minimal addition to the base version that is used by this data source. For example, suppose an ontology with concepts A, B and C. Version 1.1 added a concept D and version 1.2 added concept E. Then a data source data only relies on concepts A, C and D, would specify its commitment only to version 1.1, although there is already a version 1.2 available. We adopted this idea from software-program library versioning, as described in [10].

An interesting point for discussion is whether it would be possible to specify the *real* ontological commitment, instead of only the necessary extra commitment. In our example, this would mean that the data sources specifies that it relies on exactly A, C and D. This would require a different type of identification.

The point that states that individual concepts with a identifier that only differs in minor number are considered to be equivalent, is necessary to actually enable the backward compatibility. By default, all resources on the web with a different identifier are considered to different. This statement allows the creation of a stand-alone ontology revision, which has concepts that are equal to a previous version.

4 OntoView: support for ontology versioning

Up to now, we discussed two theoretical aspects of ontology versioning: the characteristics of a version relation and the identification of ontologies. Based on this, we will now describe a system that provides support for the versioning of online ontologies. The main function of the system is to help a user to manage changes in ontologies and keep ontology versions as much interoperable as possible. It does that by comparing versions of ontologies and highlighting the differences. It then allows the users to specify the conceptual relation between the different versions of concepts. This function is described more extensively in the next section.

It also able to store ontologies and to provide a transparent interface to arbitrary versions of ontologies. To achieve this, the system maintains an internal specification of the relation between the different variants of ontologies, with the aspects that were defined in section 2: it keeps track of the **meta-data**, the **conceptual relations** between constructs in the ontologies and the **transformations** between them.

OntoView is inspired by the Concurrent Versioning System CVS [4], which is used in software development to allow collaborative development of source code. The first implementation is also based on CVS and its web-interface CVSWeb[6]. However, during the ongoing development of the system, we are gradually shifting to a complete new implementation that will be build on a solid storage system for ontologies, e.g., Sesame[7].

The ideas underlying the versioning system are not depending on a specific ontology language. However, the implementation of specific parts of the system assume RDF based languages, for example the mechanism to detect changes. In the remainder of this article, we will use DAML+OIL[8] [11,12] and RDF Schema (RDFS) [9] as ontology languages. These two languages are widely considered as basis for future ontology languages for the Web.

Besides the ontology comparison feature — which will be described in detail in the next section — the system has the following functions:

[6] Available from `http://stud.fh-heilbronn.de/˜zeller/cgi/cvsweb.cgi/`

[7] A demo is available at `http://sesame.aidministrator.nl`

[8] Available from `http://www.daml.org/language/`

– **Reading changes and ontologies.** OntoView will accept changes and ontologies via several methods. Currently, ontologies can be read in as a whole, either by providing a URL or by uploading them to the system. The user has to specify whether the provided ontology is new or that it should be considered as an update to an already known ontology. In the first case, the user also has to provide a "location" for the ontology in the hierarchical structure of the OntoView system.
  Then, the user is guided through a short process in which he is asked to supply the meta-data of the version (as far as this can not be derived automatically, such as the date and user), to characterize the types of the changes (see below in section 5), and to decide about the identifier of the ontology.
  In the future, OntoView will also accept changes by reading in transformations, mapping ontologies, and updates to individual definitions. These update methods provides the system with different information than the method described above. For that reason, this also requires an adaptation of the process in which the user gives additional information.
– **Identification.** OntoView uses the namespace mechanism with URIs for ontology identification, separated from the location of the ontology file. Depending on the compatibility effects of the type of change (see table 1), it assigns a new identifier or it keeps the previous one.
  OntoView supports two ways of persistent and unique identification of web-based ontologies. First, it can in itself guarantee the uniqueness and persistency of namespaces that start with "http://ontoview.org/", because the system is located at the domain `ontoview.org`. Second, because the location and identification of ontologies are not necessarily coupled, it can also store ontologies with arbitrary namespaces. In this case, the ontology engineer is responsible for guaranteeing the uniqueness. The ontologies with arbitrary namespaces are not directly retrievable by their namespace, but can be accessed via a search function.
– **Analyzing effects of changes.** Changes in ontologies do not only affect the data and applications that use them, but they can also have unintended, unexpected and unforeseeable consequences in the ontology itself [16].
  OntoView provides some basic support for the analysis of these effects. First, on request it can also highlight the places in the ontology where conceptually changed concepts or properties are used. For example, if a property "hasChild" is changed, it will highlight the definition of the class "Mother", which uses the property "hasChild". In the future, this function should also exploit the transitivity of properties to show the propagation of possible changes through the ontology.
  Further, we expect to extend the system with a reasoner to automatically verify the changes and the specified conceptual relations between versions. For example, we could couple the system with FaCT [3] and exploit the Description Logic semantics of DAML+OIL to check the consistency of the ontology and look for unexpected implied relations.
– **Exporting changes.** The main advantage of storing the conceptual relations between versions of concepts and properties is the ability to use these relations for the re-interpretation of data and other ontologies that use the changed ontology. To facilitate this, OntoView can export differences between ontologies as separate mapping ontologies, which can be used as adapters for data sources or other ontolo-

gies. The mappings are created on basis of conceptual information that is attached to the update relation.

Mapping ontologies are separate ontologies that import definitions from two other (versions of) ontologies and relates these definitions conceptually to each other. They only provide a partial mapping, because not all changes can be specified conceptually, e.g. complicated changes like splits of concepts, or deletions. The definitions are imported by the namespace mechanism. This mechanism allows RDF-based ontologies to refer to definitions in other ontologies, by connecting the URI (identifier) of an other ontology with a symbolic name. The exported mapping ontologies are represented with the standard constructs of the ontology langauge.

The meta-data about the ontology update is specified as a set of properties of the conceptual relations themselves. In RDF Schema and DAML+OIL, this meant that we also have to re-ify the mapping statements. For this purpose, we defined an RDFS "meta-schema" that specifies the classes and properties that are used to attach the meta-information about an update to the mapping statements. Due to space restrictions, we cannot show it here.

This method has two advantages. First, when specified over re-ified statements, the meta-data does not interfere with the actual ontological knowledge, as would be the case when meta-data is specified as characteristics of classes and properties. Second, because the meta-data is data about the *mappings themselves*, agents or systems that understand the meta-data can use this to decide which mappings are applicable in a specific context and which are not.

In the future, it should also be possible to export *transformations* between two versions of an ontology. A transformation is a complete specification of all the change operations. This can be used to re-execute changes and to update ontologies that have some overlap with the versioned ontology in exactly the same way as the original one. However, transformations facilitates data re-interpretations only to a very small extent. A mapping ontology provides better re-interpretation, because it also captures human knowledge about the relations.

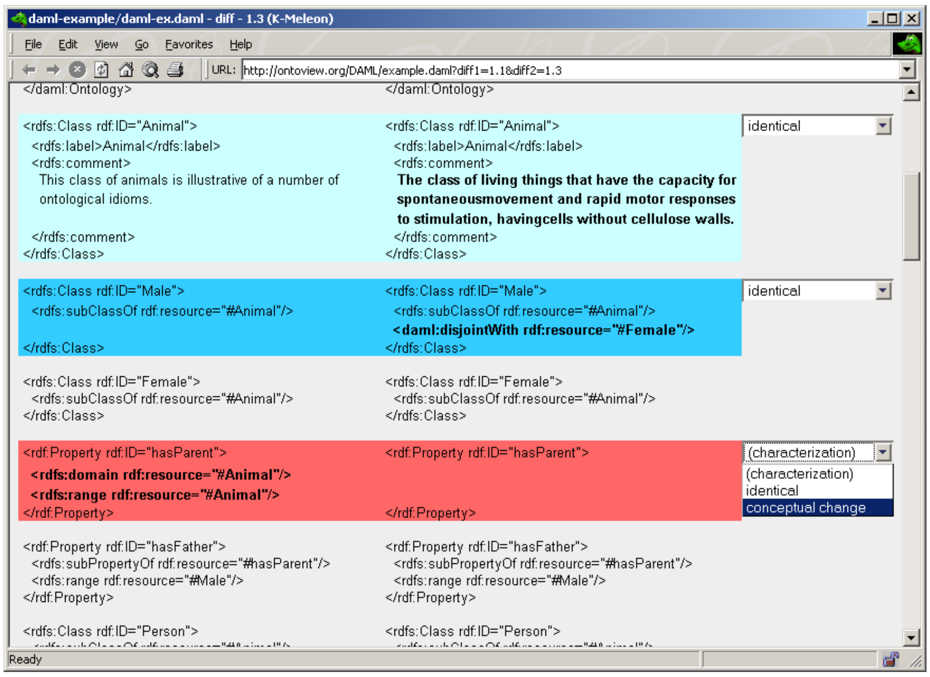
5 Comparing ontologies

One of the central features of OntoView is the ability to compare ontologies at a structural level. The comparison function is inspired by UNIX `diff`, but the implementation is quite different. Standard `diff` compares file version at line-level, highlighting the lines that textually differ in two versions. OntoView, in contrast, compares version of ontologies at a *structural* level, showing which definitions of ontological concepts or properties are changed. An example of such a graphical comparison of two versions of a DAML+OIL ontology is depicted in Figure 1.[9]

5.1 Types of change

The comparison function distinguishes between the following types of change:

[9] This example is based on fictive changes to the DAML+OIL example ontology, available from `http://www.daml.org/2001/03/daml+oil-ex.daml`.



```
</daml:Ontology>

<rdfs:Class rdf:ID="Animal">
  <rdfs:label>Animal</rdfs:label>
  <rdfs:comment>
   This class of animals is illustrative of a number of
   ontological idioms.

  </rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:ID="Male">
  <rdfs:subClassOf rdf:resource="#Animal"/>

</rdfs:Class>

<rdfs:Class rdf:ID="Female">
  <rdfs:subClassOf rdf:resource="#Animal"/>
</rdfs:Class>

<rdf:Property rdf:ID="hasParent">
  <rdfs:domain rdf:resource="#Animal"/>
  <rdfs:range rdf:resource="#Animal"/>
</rdf:Property>

<rdf:Property rdf:ID="hasFather">
  <rdfs:subPropertyOf rdf:resource="#hasParent"/>
  <rdfs:range rdf:resource="#Male"/>
</rdf:Property>

<rdfs:Class rdf:ID="Person">
```

```
</daml:Ontology>

<rdfs:Class rdf:ID="Animal">
  <rdfs:label>Animal</rdfs:label>
  <rdfs:comment>
   The class of living things that have the capacity for
   spontaneousmovement and rapid motor responses
   to stimulation, havingcells without cellulose walls.
  </rdfs:comment>
</rdfs:Class>

<rdfs:Class rdf:ID="Male">
  <rdfs:subClassOf rdf:resource="#Animal"/>
  <daml:disjointWith rdf:resource="#Female"/>
</rdfs:Class>

<rdfs:Class rdf:ID="Female">
  <rdfs:subClassOf rdf:resource="#Animal"/>
</rdfs:Class>

<rdf:Property rdf:ID="hasParent">


</rdf:Property>

<rdf:Property rdf:ID="hasFather">
  <rdfs:subPropertyOf rdf:resource="#hasParent"/>
  <rdfs:range rdf:resource="#Male"/>
</rdf:Property>

<rdfs:Class rdf:ID="Person">
```

Fig. 1. Comparing two ontologies

- Non-logical change, e.g. in a natural language description. In DAML+OIL, this are changes in the rdfs:label of an concept or property, or in a comment inside a definition. An example is the first highlighted change in Figure 1 (class "Animal').
- Logical definition change. This is a change in the definition of a concept or property that affects its formal semantics. Examples of such changes are alterations of subClassOf, domain, or range statements. Additions or deletions of local property restrictions in a class are also logical changes. The second and third change in the figure is (class "Male" and property "hasParent") are examples of such changes. Note that there are also logical changes that do not affect the semantics
- Identifier change. This is the case when a concept or property is given a new identifier, i.e. a renaming.
- Addition of definitions.
- Deletion of definitions.

Each type of change is highlighted in a different color, and the actually changed lines are printed in boldface.

Most of these changes can be detected completely automatically, except for the identifier change, because this change is not distinguishable from a subsequent deletion and addition of a simple definition. In this case, the system uses the location of the definition in the file as a heuristic to determine whether it is an identifier change or not.

It is a deliberate choice not to show all changes, but only the ones which we think that are of interest to the ontology modeler. This choice is explained in the next paragraphs, together with the mechanism that we use to detect and classify changes. Experimental validation should show whether this list of change types is sufficient.

5.2 Detecting changes

There are two main problems with the detection of changes in ontologies. The first problem is the abstraction level at which changes should be detected. Abstraction is necessary to distinguish between changes in the representation that affect the meaning, and those that don't influence the meaning. It is often possible to represent the same ontological definition in different ways. For example, in RDF Schema, there are several ways to define a class:

```
<rdfs:Class rdf:ID="ExampleClass"/>
```

or:

```
<rdf:Description rdf:ID="ExampleClass">
  <rdf:type rdf:resource="...org/2000/01/rdf-schema#Class"/>
</rdf:Description>
```

Both are valid ways to define a class and have exactly the same meaning. Such a change in the representation would not change the ontology. Thus, detecting changes in the *representation* alone is not sufficient.

However abstracting too far can also be a problem: considering the *logical meaning* only is not enough. In [2] is shown that different sets of ontological definitions can yield the same set of logical axioms. Although the logical meaning is not changed in such cases, the ontology definitely is. Finding the right level of abstraction is thus important.

Second, even when we found the correct level of abstraction for change detection, the conceptual implication of such a change is not yet clear. Because of the difference between conceptual changes and explication changes (as described in section 2), it is not possible to derive the conceptual consequence of a change completely on basis of the visible change only (i.e., the changes in the definitions of concepts and properties). Heuristics can be used to suggest conceptual consequences, but the intention of the engineer determines the actual conceptual relation between versions of concepts.

In the next two sections, we explain the algorithm that we used to compare ontologies at the correct abstraction level, and how users can specify the conceptual implication of changes.

5.3 Rules for changes

The algorithm uses the fact that the RDF data model [15] underlies a number of popular ontology languages, including RDF Schema and DAML+OIL. The RDF data model basically consists of triples of the form `<subject, predicate, object>`, which can be linked by using the object of one triple as the subject of another. There are several syntaxes available for RDF statement, but they all boil down to the same data model. An set of related RDF statements can be represented as a graph with nodes and edges. For example, consider the following DAML+OIL definition of a class “Person”.

```
<daml:Class rdf:ID="Person">
  <rdfs:subClassOf rdf:resource="#Animal"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasParent"/>
      <daml:toClass rdf:resource="#Person"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

When interpreted as a DAML+OIL definition, it states that a “Person” is a kind of ”Animal” and that the instances of its hasParent relation should be of type “Person”. However, for our algorithm, we are first of all interested in the RDF interpretation of it. That is, we only look at the triples that are specified, ignoring the DAML+OIL meaning of the statements. Interpreted as RDF, the above definition results in the following set of triples:

subject	**predicate**	**object**
Person	rdf:type	daml:Class
Person	rdfs:subClassOf	Animal
Person	rdfs:subClassOf	*anon-resource*
anon-resource	rdf:type	daml:Restriction
anon-resource	daml:onProperty	hasParent
anon-resource	daml:toClass	Person

This triple set is depicted as a graph in Figure 2. In this figure, the nodes are resources that function as subject or object of statements, whereas the arrows represent properties.

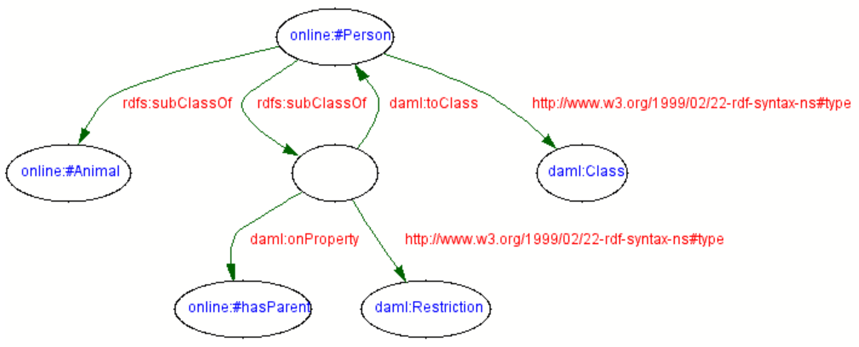


Fig. 2. An RDF graph of a DAML class definition.

The algorithm that we developed to detect changes is the following. We first split the document at the first level of the XML document. This groups the statements by their intended “definition”. The definitions are then parsed into RDF triples, which results in a set of small graphs. Each of these graphs represent a specific definition of a concept or a property, and each graph can be identified with the identifier of the concept or the property that it represents.

Then, we locate for each graph in the new version the corresponding graph in the previous version of the ontology. Those sets of graphs are then checked according to a number of rules. Those rules specify the "required" changes in the triples set (i.e., the graph) for a specific type of change, as described in section 5.1.

The rules have the following format:

```
IF exist:old
     <A, Y, Z>*
   exist:new
     <X, Y, Z>*
   not-exist:new
     <X, Y, Z>*
THEN change-type A
```

They specify a set of triples that should exists in one specific version, and a set that should not exists in another version (or the other way around) to signal a specific type of change. With this rule mechanism, we were able to specify almost types of change (except the identifier change).

For example, a rule to specify a change in the property type looks as follows:

```
IF exist:old
     <X, rdf:type, rdf:#Property>
     <X, rdf:type, daml:#UniqueProperty>
   exist:new
     <X, rdf:type, rdf:#Property>
   not-exist:new
     <X, rdf:type, daml:#UniqueProperty>
THEN logicalChange.propertytype X
```

The rules are specific for a particular RDF-based ontology language (in this case DAML+OIL), because they encode the interpretation of the semantics of the language for which they are intended. For another language other rules would have been necessary to specify other differences in interpretation. The semantics of the language are thus encoded in the rules. For example, the last example not looks at changes in values of predicates (as the first does), but at a change in the type of property. This is a change that is related to the specific semantics of DAML+OIL.

Also, notice that the mechanism relies on the "materialization" of all `rdf:type` statements that are encoded in the ontology. In other words, the closure of the RDF triples according to the used ontology language has to be computed. For example, the rules in example rule above depend on the existence of a statement `<X,rdf:type,rdf:#Property>`. However, this statement can only be derived using the semantics of the `rdfs:subPropertyOf` statement, which — informally spoken[10] — says that if a property is an instance of type X, then it is also an instance of the supertypes of X. The application of the rules thus has to be preceded by the materialization of

[10] The precise semantics of RDF Schema are still under discussion.

the superclass- and superproperty hierarchies in the ontology. For this materialization, the entailment and closure rules in the RDF Model Theory[11] can be used.

5.4 Specifying the conceptual implication of changes

The comparison function also allows the user to *characterize* the conceptual implication of the changes. For the first three types of changes that were listed in section 5.1, the user is given the option to label them either as "identical" (i.e., the change is an explication change), or as "conceptual change", using the drop-down list next to the definition (Figure 1). In the latter case, the user can specify the conceptual relation between the two version of the concept. For example, the change in the definition of "hasParent" could by characterized with the relation $\texttt{hasParent}_{1.1}$ `subPropertyOf` $\texttt{hasParent}_{1.3}$.

More complicated changes, such as deletions, splits of concepts, replacements etcetera, require additional characterizations that specify how the new change should be interpreted. We will developed this in the future.

6 Discussion

There are a few other issues and choices about the design of the system that we want to discuss. First, we purposely do not provide support for finding mappings between arbitrary ontologies. The intention of our system is to provide users with a system to manage versions of ontologies and maintain their relations. Finding the relations is a different task. However, it might be possible to incorporate this function in a future version of the system, e.g. by interfacing it with a ontology mapping tool.

We did not yet specify the way in which the "scope" of the mapping is described. The "scope" will have several dimensions, of which "time" is only one. This is something what still has to be done. Without such a specification, it is difficult to assess the validness of a conceptual relation between concepts in different versions. We can assume that such a relation is at least valid between two successive versions, but we do not know whether such mapping is allowed to "propagate" via other mappings to other ontologies. Research on this is necessary.

A situation in which versioning support is also necessary is the collaborative development of an ontology [17]. We think that OntoView is also useful in this situation, especially because all the conceptual implications of versions have to be characterized individually by users. This integrates the conflict resolution in the update procedure. That is, because users specify the conceptual relation of their changes with the previous version while specifying the update, it is not necessary to resolve conflict between definitions afterwards. Every version of the definition has its own identifier and is conceptually related to the other versions.

A side remark about the use of a versioning system for collaborative ontology development is that this gives an evolutionary way of ontology building. Each person can have its own conceptualization, which is conceptually linked to the conceptualizations

[11] `http://www.w3.org/TR/rdf-mt/`

of others. In this sense, the combination of versions and adaptations in itself forms a *shared* conceptualization of a domain.

Finally, we want to mention that the system is still under construction. In section 4 we extensively depicted the foreseen functionality of OntoView. However, as became clear of some of the descriptions, not everything is already realized. The basis functions are implemented, but a number of more advanced functions are still being developed.

7 Summary and conclusion

When ontologies are used in a distributed and dynamic context, versioning support is essential ingredient to maintain interoperability. In this paper we have analyzed the versioning relation, described its aspects, proposed an identification mechanism and finally depicted a system that helps users to manage changes in online ontologies.

We described how this systems supports helps users to compare ontologies, and what the problems and challenges are. We presented a algorithm to perform a comparison for RDF-based ontologies. This algorithm doesn't operate on the representation of the ontology, but on the data model that is underlying the representation. By grouping the RDF-triples per definition, we still retained the necessary representational knowledge. We also explained how ontology engineers have to specify the conceptual implication of changes. This honors the fact that it is not possible to derive all conceptual implications of changes automatically, because this requires insight in the conceptualization.

The analysis of a versioning relation between ontologies revealed several dimensions of it. In the system that we described, all these dimensions are maintained separately: the descriptive **meta-data**, the **conceptual relations** between constructs in the ontologies, and the **transformations** between the ontologies themselves. This multidimensional specification allows both complete transformations of ontology representations and partial data re-interpretations, which help interoperability. The conceptual differences can be exported and used stand alone, for example to adapt data sources and ontologies.

The important step in the identification method that we proposed is to separate the identity of ontologies completely from the identity of files that contain the specification of the ontology. This allows to distinguish identity changing revisions from explication changes. Moreover, we distinguish backward compatible revisions from incompatible revisions.

The described versioning methodology and the system is not yet finished and have to be developed further. Moreover, validation in a realistic setting is needed. However, we believe that the things that we presented can help to manage changes in ontologies, which will be an essential requirement for the interoperability of evolving ontologies on the web.

References

1. J. Banerjee, W. Kim, H.-J. Kim, and H. F. Korth. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *SIGMOD Record (Proc. Conf. on Management of Data)*, 16(3):311–322, May 1987.

2. S. Bechhofer, C. Goble, and I. Horrocks. DAML+OIL is not enough. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, Stanford University, California, USA, July 30 – Aug. 1, 2001.
3. S. Bechhofer, I. Horrocks, P. F. Patel-Schneider, and S. Tessaris. A proposal for a description logic interface. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, editors, *Proceedings of the International Workshop on Description Logics (DL'99)*, pages 33–36, Linköping, Sweden, July 30 – Aug. 1 1999.
4. B. Berliner. CVS II: Parallelizing software development. In USENIX Association, editor, *Proceedings of the Winter 1990 USENIX Conference*, pages 341–352, Washington, DC, USA, Jan. 22–26, 1990. USENIX.
5. T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic syntax, Aug. 1998. Status: DRAFT STANDARD.
6. T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.
7. T. Berners-Lee. Generic resources, 1996. Design Issues.
8. T. Bray, D. Hollander, and A. Layman. Namespaces in XML. `http://www.w3.org/TR/REC-xml-names/`, Jan. 1999.
9. D. Brickley and R. V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. Candidate recommendation, World Wide Web Consortium, Mar. 2000.
10. D. J. Brown and K. Runge. Library interface versioning in solaris and linux. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, Oct., 10–14 2000.
11. D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell. In R. Dieng and O. Corby, editors, *Knowledge Engineering and Knowledge Management; Methods, Models and Tools, Proceedings of the 12th International Conference EKAW 2000*, number 1937 in LNCS, pages 1–16, Juan-les-Pins, France, Oct. 2–6, 2000. Springer-Verlag.
12. D. Fensel and M. A. Musen. The semantic web: A new brain for humanity. *IEEE Intelligent Systems*, 16(2), 2001.
13. J. Heflin and J. Hendler. Dynamic ontologies on the web. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 443–449. AAAI/MIT Press, Menlo Park, CA, 2000.
14. M. Klein and D. Fensel. Ontology versioning for the Semantic Web. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, pages 75 – 91, Stanford University, California, USA, July 30 – Aug. 1, 2001.
15. O. Lassila and R. R. Swick. Resource Description Framework (RDF): Model and Syntax Specification. Recommendation, World Wide Web Consortium, Feb. 1999. See http://www.w3.org/TR/REC-rdf-syntax/.
16. D. L. McGuinness, R. Fikes, J. Rice, and S. Wilder. An environment for merging and testing large ontologies. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 483–493, San Francisco, 2000. Morgan Kaufmann.
17. H. S. Pinto and J. ao Pavão Martins. Evolving ontologies in distributed and dynamic settings. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, Toulouse, France, Apr. 22–25, 2002.
18. J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.
19. P. R. S. Visser, D. M. Jones, T. J. M. Bench-Capon, and M. J. R. Shave. An analysis of ontological mismatches: Heterogeneity versus interoperability. In *AAAI 1997 Spring Symposium on Ontological Engineering*, Stanford, USA, 1997.