Versioning of distributed ontologies

WonderWeb: Ontology Infrastructure for the Semantic Web

Michel Klein
Vrije Universiteit Amsterdam
email: michel.klein@cs.vu.nl

with contributions from:
Dieter Fensel, Atanas Kiryakov, Natasha F. Noy, Heiner Stuckenschmidt

Identifier	Del 20
Class	Deliverable
Version	1.1
Date	18-12-2002
Status	Draft
Distribution	Public
Lead Partner	VUA

WonderWeb Project

This document forms part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2001-33052.

For further information about WonderWeb, please contact the project co-ordinator:

Ian Horrocks
The Victoria University of Manchester
Department of Computer Science
Kilburn Building
Oxford Road
Manchester M13 9PL
Tel: +44 161 275 6154
Fax: +44 161 275 6236
Email: wonderweb-info@lists.man.ac.uk

Contents

1 Introduction

1.1 Semantic Web requires ontology versioning

The envisaged next generation of the Web (called Semantic Web [8]) 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[1]. 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 [23].

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.

In this deliverable, we present a framework for the versioning of online, distributed, reusable and extendable ontologies. The framework assumes a decentralized and distributed development environment where people develop and maintain ontologies without synchronization with other developers. With framework, we mean that we describe the necessary elements of a versioning system and work them out into several directions, but that we not yet present a complete versioning system.

The deliverable is structured as follows. In the remainder of this introductory chapter, we show the necessity of a versioning framework by illustrating the current practices w.r.t. versioning of online ontologies. This results in a number of general requirements that a versioning framework should fulfil. In the second chapter, we analyze in detail different aspects of changes in ontologies, which will give us the necessary input for the framework. Chapter 3 can be considered as the core of the deliverable, as it describes the basic elements of a framework for the versioning of online ontologies. In the following chapters, we then elaborate on two different aspects of the framework, i.e. identification (Chap. 4) Chapter 5 describes a tool that is developed in the project to support the presented framework. Finally, chapter 6 summarizes the report.

1.2 Current practices and requirements

To come up with concrete requirements for an ontology versioning mechanism on the web, we will now look at the current practices for managing changes in web-ontologies. We will describe a few typical scenarios for ontology change and explore the effects of those scenarios on two examples.

Currently, there is no agreed versioning methodology for ontologies on the web. However, in an decentralized and uncontrolled environment like the web, changes are certainly

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

needed and do occur! When we look at the current practices, we can sketch several scenarios for ontology changes.

1. The ontology is silently changed; the previous version is replaced by the new version without any (formal) notification.
2. The ontology is visibly changed, but only the new version is accessible; the previous version is replaced by the new version.
3. The ontology is visibly changed, and both the new version and the previous version are accessible.
4. The ontology is visibly changed, both the new version and the previous version are accessible, and there is an explicit specification of the level of compatibility between concepts of the new version and the previous version.

1.2.1 Simple example

To come to concrete requirements for a versioning methodology, we will now look at the effects of these scenarios on the compatibility when an ontology changes. As an example we use an ontology of the education system in the Netherlands and web pages that are annotated with this ontology.

In the distant past, there was only one type of higher education, which was called an "University". A small part of an ontology that describes this looks as follows:[2]

class-def Education
class-def AcadamicEducation
  subclass-of Education
class-def Higher-Education-Institute
class-def University
  subclass-of Higher-Education-Institute
  slot-constraint *type-of-education*
    has-value AcadamicEducation

Many years later, a new type of higher-education was introduced, which provides professional education. This type was called "HBO". The above ontology has to be extended with the following three definitions. This addition is a monotonic extension to the ontology and the new version can be considered as being backward compatible with the first version.

class-def ProfessionalEducation
  subclass-of Education
disjoint AcadamicEducation ProfessionalEducation
class-def HBO
  subclass-of Higher-Education-Institute
  slot-constraint *type-of-education*
    has-value ProfessionalEducation

[2] We use the "presentation syntax" of OIL [16] to represent the ontology; the interpretation is more or less straightforward. We could also have used DAML+OIL, but this would have required much more space.

Let us suppose that there are a lot of web pages about education in the Netherlands around, which are annotated using the first version of the ontology. We will now try to interpret this information using the second version of the ontology (prospective use). We describe the effects of the change for each of the scenarios that are listed above.

Ex. 1, ad 1 When we have completely no clue that the ontology is changed, we encounter — in this backward compatible case — no problems at all. The terms that are used in the data source are the same as those in the ontology, and they also have the same meaning. A “University” on a web page is correctly interpreted.

Ex. 1, ad 2 When we know that the ontology is changed, but we don’t know anything about the previous version of the ontology, nothing is sure anymore! Definitions could be changed and we can not derive that the term “University” on a web page (using ontology version 1) is the same as our definition.

Ex. 1, ad 3 In case the previous ontology can be accessed, we can compare and relate the ontologies, and see whether the changes interferes with the semantics of existing terms. In this case, we could have derived that the concept of ”University” is not changed, and that the data sources can still correctly be interpreted.

Ex. 1, ad 4 If the compatibility between the concepts is explicitly specified, it would be clear that the the new version is backward compatible with the previous version, because the new version only adds a concepts. We could then safely conclude that the interpretation of previous data is still valid.

Notice that in the last three scenarios it is important to know which version of the ontology is used to annotate the data sources. This should me made explicit in some way.

1.2.2 More complicated example

In the year 2000, the Dutch government decided that both professional and academic institutes for higher-education are allowed to call themselves “University”. This implies a new change to our ontology, resulting in a third version. This version is in general incompatible with the previous version. In the new version, the definition of “University” is changed to:

class-def University
  subclass-of Higher-Education-Institute
  slot-constraint *type-of-education*
    has-value (ProfessionalEducation **or** AcadamicEducation)

Let us again look at the consequences of this change with current versioning practices:

Ex. 2, ad 1 When we do not know that the ontology is changed, we use an other interpretation of a “University” than intended. Because in this case, a University-v1 is a subclass of University-v2, the interpretation of data is not incorrect, but also not complete. We cannot interpret that every “University” in the data sources actually provide academic education.

Ex. 2, ad 2 Same problem as with the change in the previous example.

Ex. 2, ad 3 If we have both version of the ontologies, we could compare them and see that only the definition of "University" is changed. We can see that both versions are subclasses of "Higher-Education-Institute", and interpret all instances of the old "University" as instance of "Higher-Education-Institute". This is again correct but incomplete. It ignores some knowledge that is available. Notice that, in this case, smart agents (agents capable of performing OIL classification) can derive that both Universities and HBOs are subclasses of new universities. Figure 1 shows[3] the classes in our example before and after classification.

Ex. 2, ad 4 In the case in which the relations between the concepts in the two ontologies are explicitly specified, it would tell us that "University" in the new ontology subsumes both "HBO" and "University" in the previous version.

It is also worth to notice that, in the last two scenarios, it is necessary to be able to distinguish between the different version of a concept. As the definition of "University" is changed, we need a separate identifier for each version of the definition. Otherwise, it is not possible to relate the previous and new definition to each other. In Figure 1, this is temporarily solved by appending "-v2" to the concept name.

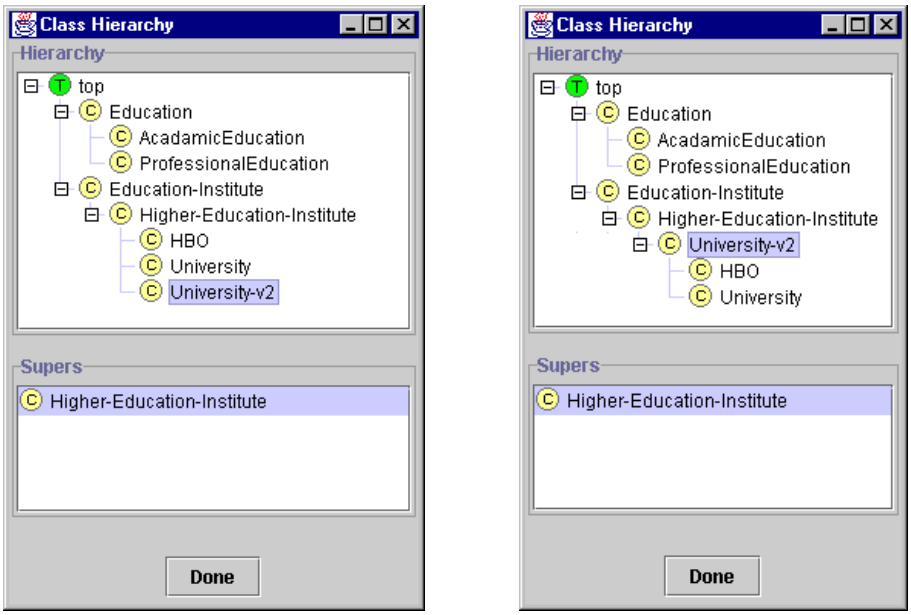

Figure 1: The hierarchy of the example ontology before and after classification with FaCT.

1.3 Observations

Based on the examples above, we can make a few observations. First, changing an ontology without any notification *may* result in a correct interpretation of the data. This is

[3]Modelled with the OILed tool, `http://oiled.man.ac.uk/`.

the case when the modification in the ontology does not affect the existing definitions, i.e., when the change is a monotonic extension. [20] show that the addition of concepts or relations are such extensions. When used on a data source, ontologies that are extended in this way yield the same perspective as when the original ontology is used. The interpretation is also valid when the revised concepts subsumes the original concepts. However, although the interpretation is correct in this case, it is only partial: not all the knowledge is exploited.

Because many changes in ontologies consist just of additions of concepts, it is understandable that the first scenario of ontology change is sometimes used. It is, however, not difficult to think of an change that — in this scenario — will result in an *invalid* interpretation, e.g., every change that restricts the extension of a class. This scenario should therefore be prevented.

Second, we see that it can be beneficial to have access to older versions of the ontology. This allows to compare the definitions and judge the validness of definitions used with other versions the data. In case of a cleanly modelled ontology, [4] it is even possible to have some automate support for this, e.g. by using the FaCT classifier[5].

As a third observation, we see that explicit knowledge about the compatibility between concepts of different versions may yield in a partial but correct interpretation of the data. This knowledge can either be manually specified, or can partly be derived, as described in the previous paragraph. For the specification of the relation between concepts of different versions, we need a identification mechanism for individual concepts of an ontology version. If we cannot refer to a specific version of a concept, this specification is not possible.

[4] That is, the definitions of concepts should state whether they are necessary or necessary and sufficient; in Description Logic parlance: primitive or defined. This way of modelling is more naturally in OIL than in DAML+OIL, as the second requires that a defined concept is modelled as an equivalence to a couple of restrictions. It is therefore questionable whether in practice DAML+OIL ontologies can benefit much from the classification support. See also the discussion on this topic on the RDF-Logic mailing-list: `http://lists.w3.org/Archives/Public/www-rdf-logic/2001Mar/0000.html`.

[5] `http://www.cs.man.ac.uk/~horrocks/FaCT/`

2 Analysis of ontology change

Developing methods for ontology change management requires that we know what ontology change actually is. In this chapter we will describe different causes of ontology change and look at their effects. This means that we look at the evolution of meta-data on the web, not at the evolution of the data that is described by the meta-data.

2.1 Causes of ontology change

To examine the causes of changes, we will have to look at the nature of ontologies. According to [19], an ontology is a *specification of a shared conceptualization of a domain*. Hence, changes in ontologies are caused by either:

1. changes in the domain;
2. changes in the shared conceptualization;
3. changes in the specification.

The first type of change is often occurring. This problem is very well known from the area of database schema versioning. In [33], seven different situations are sketched in which changes in a domain (domain evolution) require changes to a database model. An example of this type of change is the merge of two university departments: this is a change in the real world, which requires that an ontology that describes this domain is modified, too.

Changes in the shared conceptualization are also frequently happening. It is important to realize that a *shared* conceptualization of a domain is not a static specification that is produced once in the history, but has to be reached over time. In [15] ontologies are described as dynamic networks of meaning, in which consensus is achieved in a social process of exchanging information and meaning. This view attributes a dual role to ontologies in information exchange: they provide consensus that is both a *pre-requisite* for information exchange and a *result* of this exchange process.

An conceptualization can also change because of the usage perspective. Different tasks may imply different views on the domain and consequently a different conceptualization. When an ontology is adapted for a new task or a new domain, the modifications represent changes to the conceptualization. For example, consider an ontology about traffic connections in Amsterdam, with concepts like roads, cycle-tracks, canals, bridges and so on. When the ontology is adapted from a bicycle perspective to a water transport perspective, the conceptualization of a bridge changes from a remedy for crossing a canal to a time consuming obstacle[6].

Finally, a specification change is a kind of translation, i.e., a change in the way in which a conceptualization is formally recorded. Although ontology translation is an important and non-trivial issue in many practical applications, it is less interesting *from a*

[6] Actually, for many people this meaning is also an element of the bicycle perspective.

change management perspective, for two reasons. First, an important goal of a translation is to retain the semantics, i.e., specification variants should be equivalent[7] and they thus only cause syntactic interoperability problems. Second, a translation is often created to use the ontology in an other context (i.e., an other application or system), which heavily reduces the importance of interoperability questions. Therefore, we will leave specification changes alone and concentrate on support for changes in the semantics of an ontology, caused by either domain changes or conceptualization changes.

A different type of reason for ontology change is alignment. In many practical situations — for example B2B communication [29] — interoperability between applications requires the mapping of different ontologies to each other. This means that bridges are defined between the ontologies or that one of the ontologies is adapted in such a way that it is interoperable with the other one(s). In this case, the different "versions" of ontologies are independently developed and do *not* have a derivation relation. We can see the variants as separate ontologies that describe the domain from a specific *viewpoint* or *perspective*, where the adaptation specifies the change from one to the other.

2.2 Consequences of changes

A consequence of ontology change is incompatibility. This means that the original ontology can not be replaced by the changed version without causing side effects. The side effects depend on the use of the ontology and the relations that it has. We can see different types of usage of an ontology.

- In the first place, there is the data that conforms to the ontology. In a semantic web, this can be web pages of which the content is annotated with terms from an ontology. When an ontology is changed, this data may get an different interpretation or may use unknown terms.

- Second, there are other ontologies that use the changed ontology. This may be ontologies that are built from the source ontology, or that import the ontology. Changes to the source ontology may affect the resulting ontology.

- Third, applications that use the ontology may also be hampered by changes to the ontology. In the ideal case, the conceptual knowledge that is necessary for an application should be merely specified in the ontology; however, in practice applications also use an internal model. This internal model may become incompatible with the ontology.

The interpretation of compatibility is different for each of those types of usage. In the first case, compatibility means the ability to interpret all the data correctly through the changed ontology. This is much like the interpretation of compatibility in database schema versioning. Compatibility here means "preservation of instance data".

In the second case, the effects of the changes on the logical model that the ontology forms are often important. Other ontologies that import an ontology might depend on the

[7] Although in practice a translation often implies a change in semantics, possibly caused by differences in the representation languages. See for a exploration of ontology language differences and mismatches [14] and [22].

conclusions that can be drawn from the it. A change in the ontology should not make previous conclusions invalid. In this case, compatibility means "consequence preservation".

Applications that use the ontology might depend on the logical model, but also on the characteristics of the ontology itself. For example, a web site that use an ontology for navigation can depend on the fact that there are only four top-level classes, or that the hierarchy is only three levels deep. A change that does not invalidate queries to instance data or the logical model might invalidate queries to the ontology itself. This interpretation of compatibility is "preservation of answers to ontology queries".

Incompatibility between ontologies and data sources can occur in two different directions: in *prospective use* and in *retrospective use*. These terms, which are borrowed from database schema versioning literature [30], are illustrated in Figure 2. "Prospective use" is using a newer version of an ontology with a data source that conforms to an older version, and "retrospective use" is using an older version of an ontology with a data source that conforms to a more recent ontology.

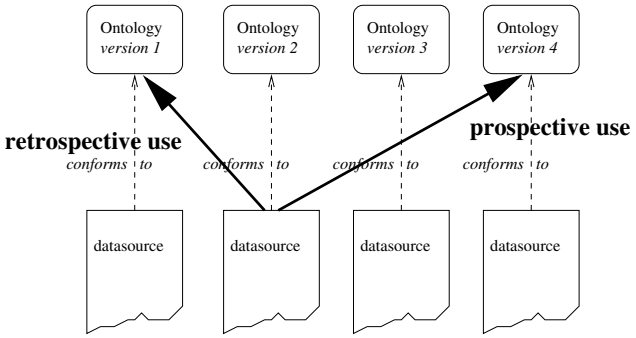

Figure 2: Examples of prospective and retrospective use of ontologies.

When the change to an ontology yields in a revision that can be used prospectively, it is called a *backward compatible* change. This is the case when the modification in the ontology does not affect the existing definitions, i.e., when the change is a monotonic extension. In [20] is shown that the addition of concepts or relations are such extensions. When used on a data source, ontologies that are extended in this way yield the same perspective as when the original ontology is used. Changes are called *upward compatible* when data sources that obey the new ontology can be used retrospectively. For example, this is true for a deletion of an independent class.

We thus see that compatibility can have different interpretations (or dimensions) and can also be considered in different directions.

2.3 Packaging of changes

A third, somewhat different, aspect of a change 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 a versioning system for ontologies.

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.

2.4 Differences between ontologies and database schemas

A final part of our analysis is a comparison of ontologies with data base schemas. For ontology evolution, we can learn from research on data base schema evolution. Schema-evolution research includes analysis of causes of change, effects of different operations on the data and frameworks for handling different versions coherently. And in theory, many issues in ontology evolution are exactly the same as the issues in schema evolution. In practice, however, there are significant differences between ontologies and database schemas from the point of view of evolution and versioning. The content and usage of ontologies are often more complex than that of database schemas. Ontologies turn some of the theoretical problems and opportunities of database-schema versioning into practical ones.

We start our comparison with differences between database schemas and ontologies in general. We then discuss different usage paradigms for database schemas and ontologies. The last group of differences addresses knowledge-representation issues. We discuss here only the differences that have direct implications on developing a framework for ontology evolution and versioning.

2.4.1 Ontologies are data, too

The main goal for schema-evolution support in databases is to preserve the integrity of the *data* itself: How does the new schema affect the view of the old data? Will queries based on the old schema work with the new data? Can old data be viewed using the new schema?

The same issues are certainly valid for instance data in ontologies. We can view ontologies as "schemas for knowledge bases." Having defined classes and slots in the ontology, we populate the knowledge base with instance data.

However, there is a major second thrust in ontology evolution: Ontologies themselves are data to an extent to which database schemas have never been. Ontologies themselves (and not the instance data) are used as controlled vocabularies, to drive search, to provide navigation through large collections of documents, to provide organization and configuration structure of Web sites. And in many cases, an ontology will not have any instance data at all. A result of a database query is usually a collection of instance data or references to text documents, whereas a result of an ontology query can include elements of the ontology itself (e.g., all subclasses of a particular class). Therefore, when considering ontology evolution, we must consider not only the effect of ontology changes on the way applications access instance data, but also the effect of these changes on queries for the ontology contents itself.

There is an extra layer of abstraction where database schemas themselves do act as data — meta data repositories [25]. Meta data repositories provide the information about various databases and applications in an organization. Ontologies are different from meta data repositories: Meta data repositories are designed to store schema and application data, whereas ontologies describe a domain of discourse for any domain. Concepts and relations in an ontology usually have formally-defined semantics that machines can interpret. In addition, meta data repositories are different from schemas themselves, providing an extra layer of description, whereas with ontologies no such extra layer exists. Therefore, while we can learn from the research in the schema-evolution issues for meta data repositories, they will not be directly applicable to ontology evolution.

2.4.2 Ontologies themselves incorporate semantics

Database schemas and catalogs often only provide a little explicit semantics for their data. Either the detailed semantics has never been specified, or the semantics were specified explicitly at database-design time in the conceptual schema, but this specification
is lost in the translation to a physical database schema and is not available anymore. Therefore, with databases, we need specific protocols for resolving conflicting restrictions when the schema changes. These protocols are usually part of a schema-evolution framework [1]. Ontologies, however, are logical systems that themselves incorporate semantics. Formal semantics of knowledge-representation systems allow us to interpret ontology definitions as a set of logical axioms. We can often leave it to the ontology itself to resolve inconsistencies and do not need to do anything about them in the evolution framework. For example, if a change in an ontology results in incompatible restrictions on a slot, it simply means that we have a class that will not have any instances (is "unsatisfiable"). If an ontology language based on Description Logics (DL) is used to represent the ontology (e.g., OIL [16]) and DAML+OIL [21]), then we can use DL reasoners to re-classify changed concepts based on their new definitions.

2.4.3 Ontologies are more often reused

A database schema defines the structure of a specific database and other databases and schemas do not usually directly reuse or extend existing schemas. The schema is part of an integrated system and is rarely used apart from it. There are exceptions to this rule, which include schemas that support packaged commercial produces for applications such as accounting and personnel records. The situation with ontologies is exactly the opposite: Ontologies often reuse and extend other ontologies, and they are not bound to a specific system. Therefore, a change in one ontology affects all the other ontologies that reuse it, and, consequently, the data and applications that are based on these ontologies. Even seemingly monotonic changes, such as additions of new concepts to an ontology, can have adverse effects on the other ontologies that reuse it. If we add a concept that already exists in the reusing ontology, no logical conflicts arise, but the reusing ontology contains two representations of the same concept. We will need to specify an equivalence statement to reflect this fact.

2.4.4 Ontologies are de-centralized by nature

Traditionally, database schema development and update is a centralized process: Developers of the original schema (or employees of the same organization) usually make the changes and maintain the schema. The development and maintenance of integrated databases [2] and federated database systems [31] is already much more de-centralized, but at the very least, database-schema developers usually know which databases use their schema. By nature, ontology development (and, therefore, evolution) is an even more de-centralized and collaborative process. As a result, there is no centralized control over who uses a particular ontology. It is much more difficult (if not impossible) to enforce or synchronize updates: If we do not know who the users of our ontology are, we cannot inform them about the updates and cannot assume that they will find out themselves. Lack of centralized and synchronized control also makes it difficult (and often impossible) to trace the sequence of operations that transformed one version of an ontology into another. The envisioned huge scale of the Semantic Web and even more de-centralization in ontology development and maintenance greatly exacerbate the problem: In today's Web, we can neither know who uses an ontology that we maintain or how many users there are, nor prevent or require others to use a particular ontology. It is interesting to note that in recent years, the database field is moving in the direction of de-centralization as well: there are standard XML schemas that are reused through different applications, particularly in e-commerce.

2.4.5 Ontology data models are richer

An inherent part of any schema-evolution methodology is a detailed consideration of the effects of each change operation on the data: What happens if a new superclass is added to a class in an object-oriented database, if an instance variable is removed from a class, if a domain of an instance variable is changed, and so on. The number of representation primitives in many ontologies is much larger than in a typical database schema.

For example, many ontology languages and systems allow the specification of cardinality constraints, inverse properties, transitive properties, disjoint classes, and so on. Some languages (e.g., DAML+OIL) add primitives to define new classes as unions or intersections of other classes, as an enumeration of its members, as a set of objects satisfying a particular restriction. Therefore, any detailed treatment of ontology changes must include a much more extensive set of possible operations.

2.4.6 Classes and instances can be the same

Databases make a clear distinction between the schema and the instance data. In many rich knowledge-representation systems it is hard to distinguish where an ontology ends and instances begin. The use of meta-classes — classes which have other classes as their instances [13] — in many systems (e.g., Protégé [18]), Ontolingua, RDFS [10] blurs or erases completely the distinction between classes and instances. In set-theoretic terms, meta-classes are sets whose elements are themselves sets. This means that "being an instance" and "being a class" is actually just a *role* for a concept. For example, the "Lonely Planet for Amsterdam" is a specific instance of the class "Travel guides" in a bookstore; at the same time, however, it is a class of which the individual copies of the book are instances. Therefore, analysis of schema-change operations, which considers only effects on instance data, is not directly applicable to ontologies.

2.5 Requirements

To prevent that the changes to ontologies will make them useless because of compatibility problems, we need a methodology that allows humans and computers to cope with ontology versioning. We define ontology versioning as *the ability to manage ontology changes and their effects by creating and maintaining different variants of the ontology*. To achieve this ability, a methodology should provide methods to distinguish and recognize versions, procedures for updates and changes in ontologies, and an interpretation mechanism for effects of change.

Focused on ontologies, we can say that a *versioning methodology* provides mechanism to disambiguate the interpretation of concepts for users of the ontology variants, and that it makes the effects of changes on different tasks explicit. The extend of the changes determines the compatibility between the versions. This implies that the semantic impact of changes should be examined.

The general goal can be detailed in a number of more specific requirements. We impose the following requirements on a versioning framework:

- for every use of a concept or a relation, a versioning framework should provide an unambiguous reference to the intended definition; (**identification**)
- a versioning framework should make the relation of one version of a concept or relation to other versions of that construct explicit; (**change specification**)
- a versioning framework should — as far as possible — automatically translate and relate the versions and data sources, to enable transparent access to the data sources via different ontology versions; (**transparent evolution**)

- since there are several dimensions to compatibility (e.g., preservation of instance data, preservation of answers to ontology queries, consequence preservation, etc.), a versioning framework should distinguish between the effect of changes on different tasks; (**task awareness**)

- a versioning framework should provide mechanism to determine the compatibility between different versions for which there is no trace of the changes that led from one version to another. (**support for untraced changes**)

Identification is the first — and most obvious — requirement for a versioning methodology. If it is not clear what the intended definition of a concept is, both the interpretation of information is unknown and change specification is not possible.

Change specification is a second requirement. Knowledge about the relation between concepts of different versions may help to achieve a correct interpretation of the data when a changed ontology is used. This relation can either be manually specified, or can partly be derived automatically. In case of a an ontology language that is based on a Description Logic (e.g. DAML+OIL), it is possible to state whether the definitions of concepts are necessary or necessary and sufficient (in Description Logic parlance: primitive or defined). This allows an automatic re-classification of changed concept definitions, which gives the subsumption relation between the two versions of the concept.

The third requirement is transparent evolution. This means that the methodology should provide mechanisms that specify how data and ontology concepts should be interpreted when the changes are applied. Those mechanisms will use the change specifications to automatically translate and relate the different versions of concepts. The mechanisms should provide a correct interpretation to as much data as possible, without deriving incorrect information. Notice that this does not require that there all data can correctly be interpreted through all versions of the ontology (i.e. that all versions are compatible). It means that the interpretation rules have to make clear what can be translated (and how it will be translated) and what can not be interpreted anymore. This will thus yield in a correct but partial interpretation.

Task awareness is a fourth requirement. In order to determine which changes to an ontology are backward-compatible, we need to determine what compatibility means. In databases, backwards-compatibility usually means the ability to access all of the old data through the new schema. In other words, no instance data is lost as a result of the change. For ontologies, query results can include not only instance data but also elements of the ontology itself. Therefore, we cannot express compatibility only in terms of preservation of instance data. Consider a situation in which a new class is added to an ontology as a subclass of an existing class. This change has no effect on instance data and will not change or invalidate answers to existing queries that return only instance data. However, if queries are about the ontology itself (e.g., a list of subclasses of a specific class), the answers to existing queries change. This issue becomes even more complicated with ontology languages that support automatic classification (e.g., DAML+OIL): When a class is added to an ontology, a reasoner can re-classify existing concepts and instances, possibly invalidating existing data or applications. When we characterize the effects of change operations we need to take the different tasks into account.

A final requirements is about the compatibility between ontologies for which we do not have a trace of changes. In this case, all we have are two versions of an ontology and no knowledge of the steps that led from one version to another. We will need to find the differences between the two versions in an automated or semi-automated way. The problem of finding the differences between (versions of) ontologies is in fact very close to the problem of ontology merging. In both cases, we have two overlapping ontologies and we need to determine a mapping between their elements. When we are merging ontologies, we concentrate on similarities, whereas in evolution we need to highlight the differences, which can be a complementary process. In addition, in the case of ontology evolution we need to make much more "liberal" assumptions in determining which concepts are the same. For example, if we are merging two ontologies that came from independent sources, we cannot assume that two classes named `University` in the two sources refer to the same concept: One could refer to the organizational structure of a university and the other to a university campus. However, if two different versions of the same ontology both contain a class named `University`, it is much more likely that these classes do indeed refer to the same concept.

3 Version framework

The versioning framework provides mechanisms and guidelines for the versioning of online, distributed, reusable and extendable ontologies. We start this chapter with a discussion of the different objectives of "versioning" (section 3.1). This list describes the broad area of tasks that a framework should eventually be able to support. We then formulate a number of basic assumption that underly the framework (in section 3.2). In section 3.3 we describe the actual approach for ontology versioning. This approach consists of template for the specification of version relations, guidelines and mechanisms to collect and derive this information, and an ontology of change types and effects. A tool that could help to fill in the template for version relations is discussed in chapter 5.

3.1 Objectives of versioning support

The general goal of a versioning methodology is to provide interoperability. In this case, interoperability can be defined as the ability to use different versions of ontologies and possible data sets together in a meaningful way. Meaningful means that the way in which ontologies or data are uses is not conflicting with the intended meaning of the ontology.

However, this general goal can be achieved in several ways and has several specific interpretations, which may differ for the different tasks for which ontologies are used (see section 2.2). Below we describe a number of different tasks in which a versioning system my help users that work with evolving ontology. It is not at all meant as a definition of components of a versioning support system, but it just sketches the wide area of tasks where some support for change management might be useful.

3.1.1 Data accessibility

The most often mentioned objective of versioning is the ability to access instance data via a different version of the ontology than the one that was used to describe it. In this case the interpretation of interoperability is data accessibility. Data accessibility can be achieved in two ways:

1. Restricting the modification to changes do not affect the interpretation of the data, i.e. *backward compatible* changes. In this case the change in the ontology does not change the perspective on the data [20].

2. Translating data structures from one to another version. For example, if two classes are merged into one, this usually has as a result that instances of the original classes cannot be accessed anymore. However, if the querying agent knows about the merge, it can translate the instances of the original classes to the new class. In [27] a taxonomy of change operations is shown where this translation might be possible.

Full data accessibility is not always achievable, of course. Some changes might make a part of the data inaccessible, for example, a deletion of a class.

3.1.2 Consistent reasoning

Another interpretation of interoperability is that the reasoning over the ontologies is not affected. This forms a second objective of versioning. In general, this would mean that answers to logical queries are not changed by modifications in the ontology. Of course, this is almost never true in the general case: only non-logical changes will have this effect. For each logical change a query can be invented that will have a different answer.

However, it might be very useful to know that a change in an ontology does not affect the answers to a *specific set* of queries. In many cases, applications will not ask arbitrary queries to an ontology, but only a number of predefined ones. For example, an application might be interested in the instances of a class, or in the properties that can be applied to a class.

If it can be predicted whether or not the answers to such a set of queries will change, it is possible to decide if the versions of ontologies can be exchanged for a specific task.

A set of queries which should give an consistent answer can be defined explicitly, by listing those queries, or implicitly, by specifying the type of reasoning that is assumed. For example, for ontologies that have an underlying description logic, consistent reasoning means that the derived subsumption hierarchy is not affected by the changes in the ontology.

3.1.3 Synchronization

Something else that is often considered to be part of versioning support is the ability to make copied (and possibly changed) versions of ontologies up to date with a remotely changed ontology. This might be desirable when an ontology is copied and the original is consecutively changed but it is still necessary to work with the original one, too. This requires that all consecutive changes in the original ontology are carried out in the local copy. This process is called *synchronization* in [28]. The term used in software development is *patching*.

This goal is often part of another objective: collaborative development of ontologies. Besides synchronization, collaborative working requires mechanisms for access control (e.g. locking) and conflict resolution.

3.1.4 Data translation

A related goal is the translation of data sources that conform to one version of an ontology in such a way that they conform to another version. This requires that consecutive changes between the two versions of the ontology are executed in the data sets. This is useful in situations where there is control over the data sources and it is important to keep data sets up to date with the ontology, e.g. in a company where the information on the intranet has to be annotated with a specific ontology.

3.1.5 Management of development

Something else that a versioning framework could support is a step by step verification and authorization of changes to an ontology. This task is also useful in collaborative

working scenario. It is related to *synchronization*, but there are some differences. First, it allows a step-by-step acceptation or rejection of the changes, so the user can control the process. A practical difference is that the changes are often performed to the unchanged ontology, instead of to a locally changed ontology. As a consequence, less conflicts will occur in this scenario.

This task requires that the changes in an ontology are presented to the user in a such way that he or she can understand what the change is. This means that it is often not enough to show a list of additions and deletions of terms, but as "aggregations" of a number of atomic changes that are performed for the same reason. For example, a change that occurs quite often is the introduction of an additional distinction: a class gets two new subclasses and its former subclasses become subclasses of the new ones. It makes much more sense if this is presented to the user as one operation instead of 5 or more atomic addition and move operations.

3.1.6 Editing support

Another type of change management support is support during the editing of changes in ontologies. One aspect is the propagation of changes, i.e. the handling of effects of changes in other parts of the ontology. This can be as simple as highlighting the consequences, or as advanced as automatically performing necessary additional changes. For example, concepts and properties could be updated or deleted automatically if they refer to other deleted concepts. An implementation of this type of support is described in [32]. They use different "strategies" that describe the what actions should be performed if concepts are deleted.

3.2 Starting points

Based on the analysis in the previous chapter, we will now formulate a number of starting points for the ontology versioning framework. These starting points function as underlying assumptions for the framework.

3.2.1 Change operations do not determine the conceptual consequence

A first starting point is the fact that the effect of change operations (i.e. the atomic actions, e.g. changing the domain of a slot) on the conceptual relations between concepts in an ontology cannot be derived automatically, but is a human decision.

The reason for this is the following. 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 [23], 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 de-

scribed 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 [34] 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 not possible to determine automatically whether a change is a conceptual change or an 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 on the conceptual relations in the ontology. We will discuss this later on.

3.2.2 Version relations are more than just conceptual relations

A second starting point 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.[8] This is depicted in Figure 3.

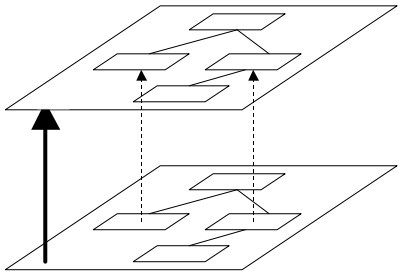

Figure 3: A version relation between two ontologies (thick arrow) implies version relations between classes in two versions of an ontology (dashed arrows). These version relations are orthogonal to the conceptual relations inside an ontology (lines between boxes).

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

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

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. The meta data of changes will be described in a next section.

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.

3.2.3 Consequences are task dependent

A third axiom is the task dependence of the effects of change. In section 3.1, we already discussed that there are different interpretations of interoperability. As a consequence, it is not possible to describe the effects of change operations on the compatibility in general terms (e.g., "compatible" or "not-compatible"), but only by referring to a specific task.

This is notably different from databases, where "backward compatible" usually means that you can interpret old data correctly via the new version of the schema. This is also the perspective that is described in [20], and is a very important perspective in the context of the Semantic Web. However, for ontologies there are also other perspectives, resulting from the different tasks for which ontologies are used. For example, if you use the ontology for classification, a change that is backward compatible from the "data interpretation" perspective, might result in different answers to the classification task.

We will give two examples of changes that are considered as backward compatible from a data interpretation perspective, but that influence other usage.

1. Suppose a small ontology, a class Conference and a class City, connected with a property located_in, and a small data set IJCAI03 located_in Acapulco (see Figure 4). We can now derive that Acapulco is an instance of City. Now, let us change the ontology in such a way that City gets a superclass Location and the domain of located_in is changed to Location. This is a backward compatible change from the data interpretation point of view. However, the most specific answer we can get now on the question "what kind of thing is Acapulco" is Location. This is not wrong of course, but different.

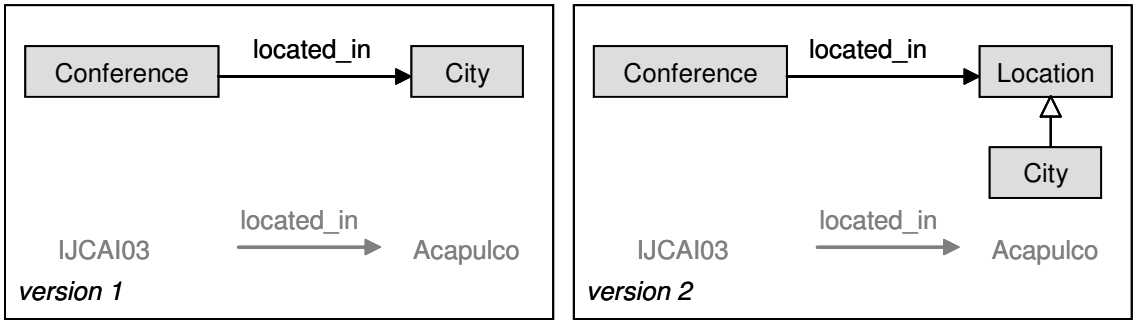

Figure 4: Two versions of a very simple ontology and a piece of associated data.

2. Suppose we add a slot restriction to a class. If it is a for-all restriction, this is a backward compatible change. However, if we think about it from a subsumption reasoning point of view, it makes the class definition more specific and might change the resulting hierarchy.

It is thus important to relate the effect of a change to a specific task. A specific change is than associated with different consequences. In example 1 above, we could specify for the class City that there is no effect on data accessibility, but that the effect on a query for the instances is that some of the previous instances cannot be found anymore.

3.3 Versioning approach

The versioning approach assumes a decentralized and distributed development environment where people develop and maintain ontologies without synchronization with other developers. The approach is not yet a complete ontology versioning solution, but it provides a number of import building blocks of it. The approach consist of four main elements:

- a "template" for the specification of the relation between related ontologies, e.g. different versions;
- an ontology of change operations;
- rules that determine the effect of change operations on different tasks;
- a tool that detects changes and helps to specify the relation between versions according to the template.

The template functions as a placeholder for the information that is required to support the different objectives of a versioning system that are listed in section 3.1. It also describes the effect of a change on different tasks for each individual concept or property (definition). Agents or application can then decide about the effect on a specific task in which specific definitions are used.

The tool, which is described in more detail in chapter 5, is used to derive the information that is required to fill the template of version relations. For this purpose, the tool uses several sources:

- the detected differences between two versions of an ontology,
- the human specification of the conceptual implication of a change (see section 3.2.1), and
- the rules that determine the effect of change operations on different tasks.

With respect to the third point, at the moment we developed rules to determine the effect of changes on data interpretation and on the direction of movement of a concept in a subsumption hierarchy. The last thing can be useful when applications depend on the specific hierarchy. We are currently using this to predict the validity of interfaces to ontologies that contain references to external concepts.

3.3.1 Template for change specification

We distinguish the following properties of a version relation between two versions of a concept:

- **evolution relation**: the relation that specifies which ontological definition (e.g. class) in one version is changed into which ontological definition in another version. This is the dashed arrow in Figure 3.
- **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. This is specified via the "change ontology" (see section 3.3.2).
- **conceptual relation**: the relation between constructs in the two versions of the ontology, e.g., specified by equivalence relations, subsumption relations, or logical rules;
- 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 [30] (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.

Meta data representation in RDFS 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. This meta-schema is published at `http://ontoview.org/schema/meta/1/1`.

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.

For example, let us consider a change from a concept $\text{Pet}_1.0$ to $\text{Pet}_2.0$. Based on the input of the ontology engineer, we know that first version is conceptually subsumed by the second version. This is denoted by the following statement:

```
<rdf:Description rdf:about="http://www.cs.vu.nl/~mcaklein/1/0#Pet">
  <rdfs:subClassOf rdf:resource="http://www.cs.vu.nl/~mcaklein/2/0#Pet"/>
</rdf:Description>
```

This statement can be used by agents or systems that are not aware of any versioning approach. The asserted statement, however, is re-ified into the following set of statements:

```
<rdf:Statement>
  <rdf:subject rdf:resource="http://www.cs.vu.nl/~mcaklein/1/0#Pet"/>
  <rdf:predicate rdf:resource="...subClassOf"/>
  <rdf:object rdf:resource="http://www.cs.vu.nl/~mcaklein/2/0#Pet"/>
</rdf:Statement>
```

This set of re-ified statements in now extended with a number of other properties that contain the meta-data as described in the previous section.

```
  ...
  <rdf:type rdf:resource="...#ConceptualChange/>
  <ov:from rdf:resource="http://www.cs.vu.nl/~mcaklein/1/0#Pet"/>
  <ov:to rdf:resource="http://www.cs.vu.nl/~mcaklein/2/0#Pet"/>
  <ov:effect>
    <ov:Classification rdf:resource="...#Specialized"/>
    <ov:DataInterpretation rdf:resource="...#BackwardCompatible"/>
  </ov:effect>
  <ov:transformation>
     ... see next section ...
  </ov:transformation>
  <ov:author>Michel Klein</ov:author>
  <ov:reason rdf:resource="...#Mistake"/>
</rdf:Statement>
```

Agents and systems that are aware of the versioning approach are now able to use this meta-data to decide about the compatibility between different versions.

3.3.2 Ontology of changes

In this section, we present an ontology that can be used to describe the changes that occur in ontologies. We will use this general taxonomy of changes to specify the effect of changes on specific tasks in the next section.

The set of possible change operations for ontologies is larger than the traditional sets of database schema-change operations [1]. There are two causes of the differences between these two sets. The first cause is the richer knowledge model for ontologies: We must add operations that deal with changes in slot restrictions, with slot attachment, and so on. The second cause is the use of composite operations which few researchers in the schema-evolution community have addressed. Consider for example a change in the domain of a slot from a class to its superclass. If we treat this operation as a sequence of two operations, we would have to delete all the values of the slot for all instances of the class after the first operation. However, after the second operation, all instances can have the slot again. The composite effect of the two operations does not violate the integrity of the instance data, whereas one of the operations does.

Therefore, the change ontology includes these composite operations since their compound effect on schema evolution (1) is predictable and (2) can belong to a completely different class of operations than each of the simple operations that constitute it.

At the highest level, the ontology makes a distinction between atomic changes and composite changes. Some of the composite changes are named, others are just unnamed aggregations of atomic changes. The named composite changes make it possible to specify an effect that is different than the combination of the effects of composite changes. The ontology assumes the OWL-light knowledge model.

The figures 5, 6, 7 and 8 show a class hierarchy of atomic change operations on resources (ie. classes, properties *and* individuals), classes, properties and individuals respectively. Note that classes that are marked with a small A are abstract classes; these classes should not be used to specify a actual transformation, but they are introduced for structuring purposes. An RDFS version of the change ontology can be found at `http://ontoview.org/changes/1/3/`.

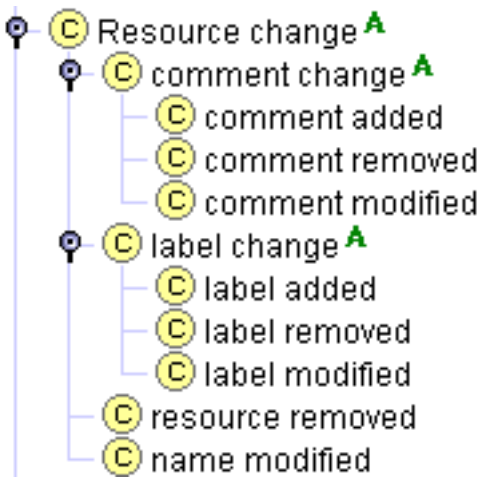

Figure 5: Possible changes of classes, properties, as well as instances.

Figure 9 shows a number of named composite changes. Note that this can never be a complete list, new composite changes can always be constructed. However, it only useful to name composite changes if you want to use them for some purpose, e.g. the specification of another effect than the atomic changes of which it consists, or to visualize the aggregated change in some way.

The ontology of changes allows a complete (i.e. reversible and re-doable) specification of the transformations. The inheritance relations between different classes in the ontology can be used to specify the effect of changes.

An actual transformation that is described with the change ontology could look as follows. The RDF data describes that the filler of a slotrestriction for ”slotA” is changed from ”ClassA” to ”ClassB”

```
...
<ov:transformation>
  <ov:Slotrestriction_Filler_Modified>
    <ov:slot rdf:resource="#slotA"/>
```

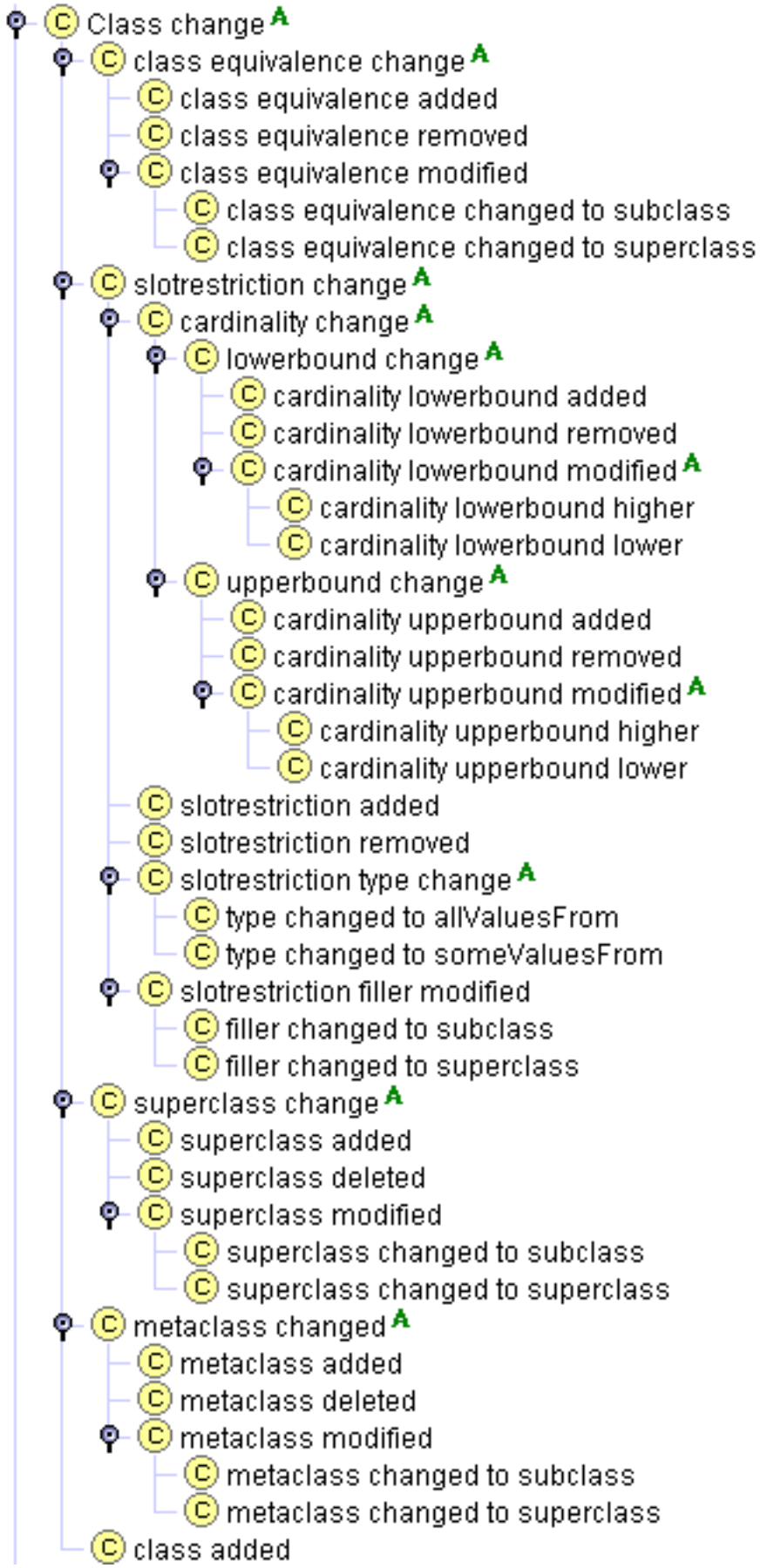

Figure 6: Possible changes in classes.

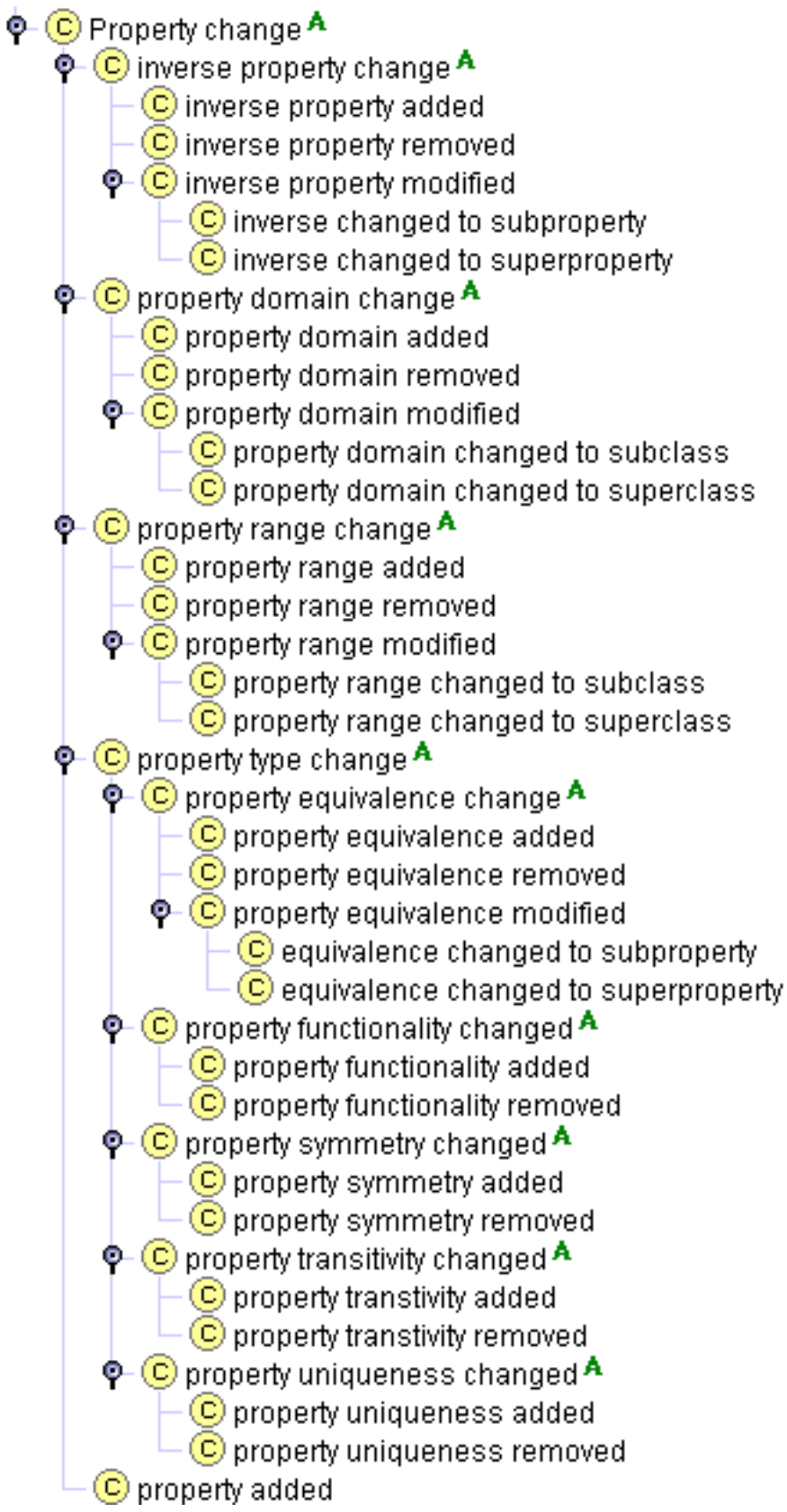

Figure 7: Possible changes in properties.

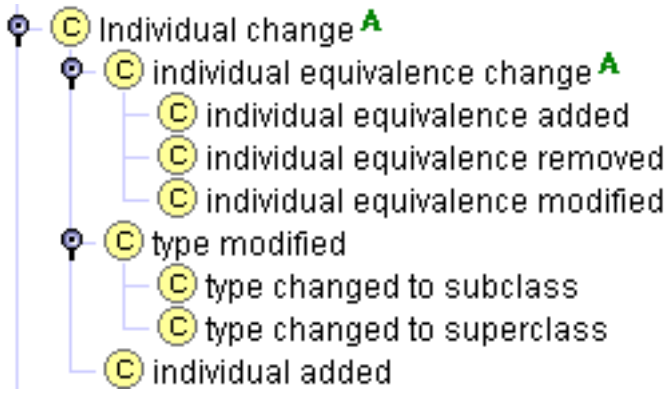

Figure 8: Possible changes in classes.

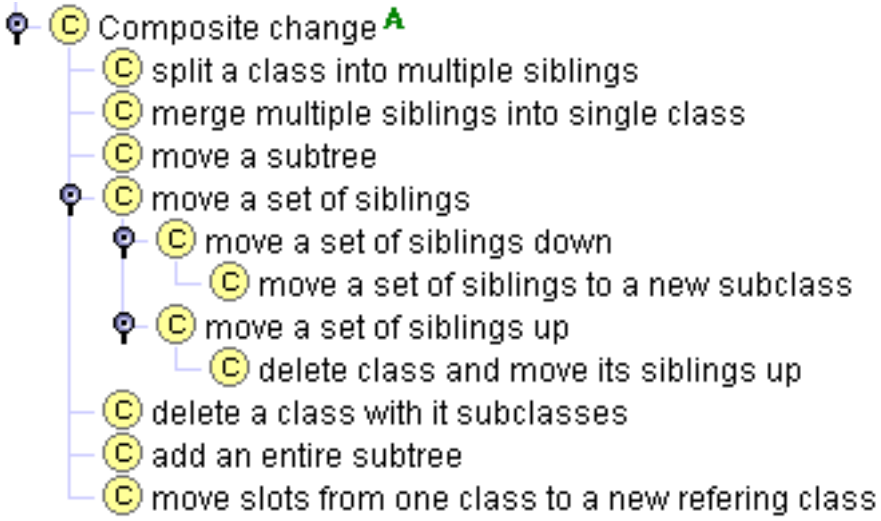

Figure 9: Some named composite changes.

```
    <ov:oldValue rdf:resource="#ClassA"/>
    <ov:newValue rdf:resource="#ClassB"/>
  </ov:Slotrestriction_Filler_Modified>
</ov:transformation>
...
```

Composite changes are represented as sets of atomic changes. The following piece of RDF data gives an idea how to specify that the subclasses of a class are split over two new subclasses.

```
...
<ov:transformation>
  <ov:Subclasses_Moved_To_New_Subclass>
    ...
    class added
    ...
    <ov:atomic_operation>
      <ov:Superclass_Changed>
        <ov:class rdf:resource=""/>
        <ov:oldValue rdf:resource="#ClassA"/>
        <ov:newValue rdf:resource="#ClassB"/>
      </ov:Superclass_Changed>
    </ov:atomic_operation>
    ...
    other superclass relation changed
    ...
  </ov:Subclasses_Moved_To_New_Subclass>
</ov:transformation>
...
```

3.3.3 Rules that determine the consequences of change operations

The possible consequences of these changes can now be described for a number of different tasks in which ontologies are used. In table 1, we consider the effects with respect to instance-data-preservation, i.e. we consider whether instance data can still be accessed through the changed ontology. We classify the operations effects as:

- Information-preserving changes: no instance data is lost (“+” in the table);
- Translatable changes: no instance data is lost if a part of the data is translated into a new form (“$\diamond$” in the table).
- Information-loss changes: it cannot be guaranteed that no instance data is lost (“-” in the table).

In this analysis we also assumed that there is no automatic classification.

We can also specify what the effect of individual change operations is on the classification of concepts in the hierarchy. This could be useful in situations were the recalculation

Operation	**Effect**
Create class C	+
Delete a class C	-
Add a subclass-superclass link between a subclass SubC and a superclass SuperC	+
Remove a subclass-superclass link between a subclass SubC and a superclass SuperC	-
Change a superclass of a class C to a class higher in the hierarchy	-
Change a superclass of a class C to a class lower in the hierarchy	+
Merge classes	◇
Split a class	◇
Change a class C from defined to primitive	+
Change a class C from primitive to defined	+
Re-classify an instance I as a class	+
Re-classify a class C as an instance	-
Declare classes C_1 and C_2 as disjoint	-
Create slot S	+
Delete a slot S	-
Define a slot S as transitive	-
Define a slot S as symmetric	-
”Widen” a restriction for a slot S (e.g., increase the number of allowed values, decrease the number of required values, add a class to the range or replace an existing class in the range with its superclass, etc.)	+
”Narrow” a restriction for a slot S (e.g., decrease the number of allowed values, increase the number of required values, remove a class from the range or replace it with a subclass, etc.)	-
Attach a slot S to a class C	+
Remove a slot S from a class C	-
Move a slot S from a subclass SubC to a superclass SuperC	+
Move a slot S from a superclass SuperC to a subclass SubC	-
Move a slot S from a class C_1 to a referenced class C_2	◇
Encapsulate a set of slots into a new class	◇
Change a slot-restriction from $\forall$ to $\exists$	+
Change a slot-restriction from $\exists$ to $\forall$	-

Table 1: Ontology-change operations and their effects on the interpretation of instance data.

of the hierarchy is too expensive or not possible. With this descriptions, it is still possible to tell something about the validity of subsumption relations without computing the new hierarchy. (This will be described in detail in an upcoming paper). Table 2 shows these effects: a “+” means that the concept might become more general, a “-” means that the concept might go down in the hierarchy. A “?” means that the effect on the movement in the hierarchy is unpredictable.

Operation	**Effect**
Create class C	?
Delete a class C	?
Add a subclass-superclass link between a subclass SubC and a superclass SuperC	-
Remove a subclass-superclass link between a subclass SubC and a superclass SuperC	+
Change a superclass of a class C to a class higher in the hierarchy	+
Change a superclass of a class C to a class lower in the hierarchy	-
Merge classes	-
Split a class	+
Change a class C from defined to primitive	+
Change a class C from primitive to defined	-
Re-classify an instance I as a class	?
Re-classify a class C as an instance	?
Declare classes C_1 and C_2 as disjoint	-
Create slot S	?
Delete a slot S	?
Define a slot S as transitive	-
Define a slot S as symmetric	-
”Widen” a restriction for a slot S (e.g., increase the number of allowed values, decrease the number of required values, add a class to the range or replace an existing class in the range with its superclass, etc.)	+
”Narrow” a restriction for a slot S (e.g., decrease the number of allowed values, increase the number of required values, remove a class from the range or replace it with a subclass, etc.)	-
Attach a slot S to a class C	-
Remove a slot S from a class C	+
Move a slot S from a subclass SubC to a superclass SuperC	-
Move a slot S from a superclass SuperC to a subclass SubC	+
Move a slot S from a class C_1 to a referenced class C_2	?
Encapsulate a set of slots into a new class	?
Change a slot-restriction from $\forall$ to $\exists$	+
Change a slot-restriction from $\exists$ to $\forall$	-

Table 2: Ontology-change operations and their effects on the classification of concepts in the hierarchy.

4 Identification of ontologies

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.

4.1 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 3. 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 3: Change types and their effect on the identity of an ontology.

4.2 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[9]-terminology. According to the definition of Uniform Resource Identifiers (URI's) (defined in [7]), "a resource can be anything that has identity". In [6] 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" [7] 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 [9] 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 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".

4.3 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;

[9] The standardization body for the World Wide Web

- 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 [20]. The distinction we make is also in line with the idea of "levels of generality", which is discussed in [6]. 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 [12].

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.

5 Tool support

5.1 Functions

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 3.3.1: 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 [5], 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[10]. 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[11].

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 chapter, we will use DAML+OIL[12] [16, 17] and RDF Schema (RDFS) [11] 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:

- **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 5.2), and to decide about the identifier of the ontology.

[10] Available from `http://stud.fh-heilbronn.de/~zeller/cgi/cvsweb.cgi/`
[11] A demo is available at `http://sesame.aidministrator.nl`
[12] Available from `http://www.daml.org/language/`

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 3), 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 [26].

  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 [4] 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 ontologies. 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.

5.2 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 10.[13]

5.2.1 Types of change

The comparison function distinguishes between the following types of change:

- 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 10 (class "Animal').

[13] 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`.

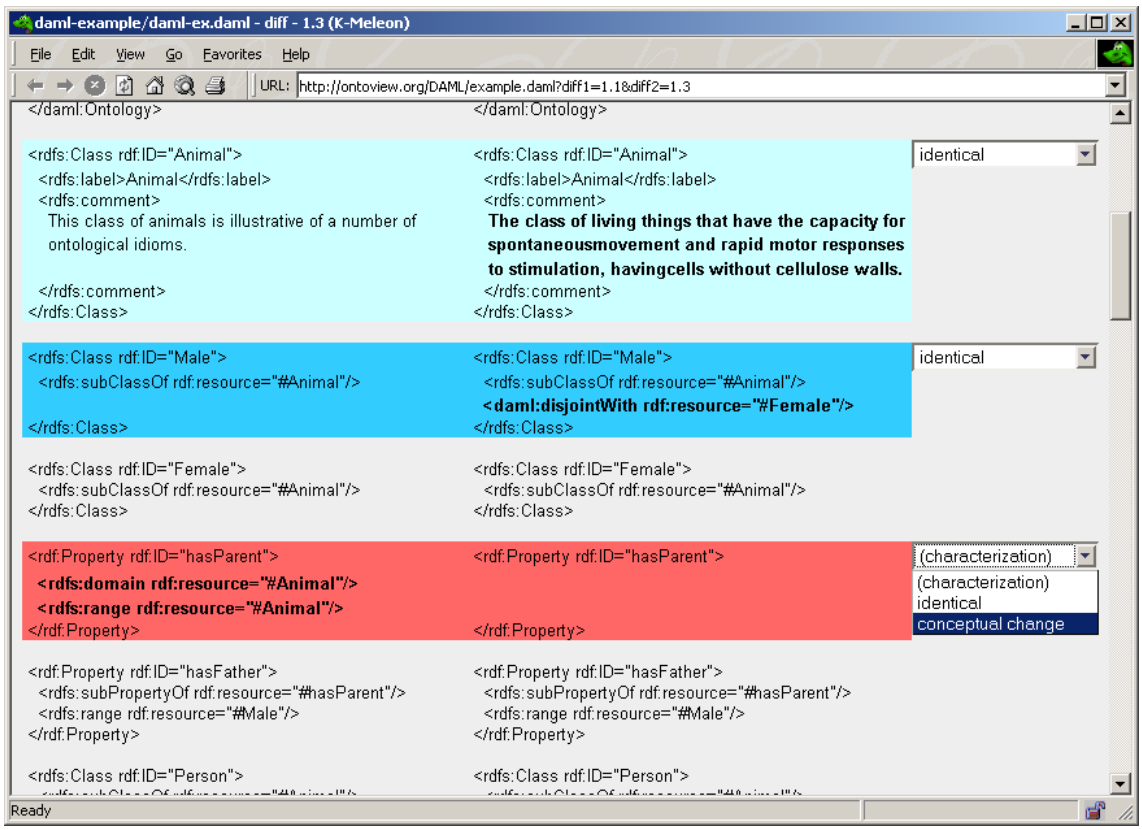

Figure 10: Comparing two ontologies

- 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.

5.2.2 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 [3] 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 3.2.1), 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.

5.2.3 Rules for changes

The algorithm uses the fact that the RDF data model [24] 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".

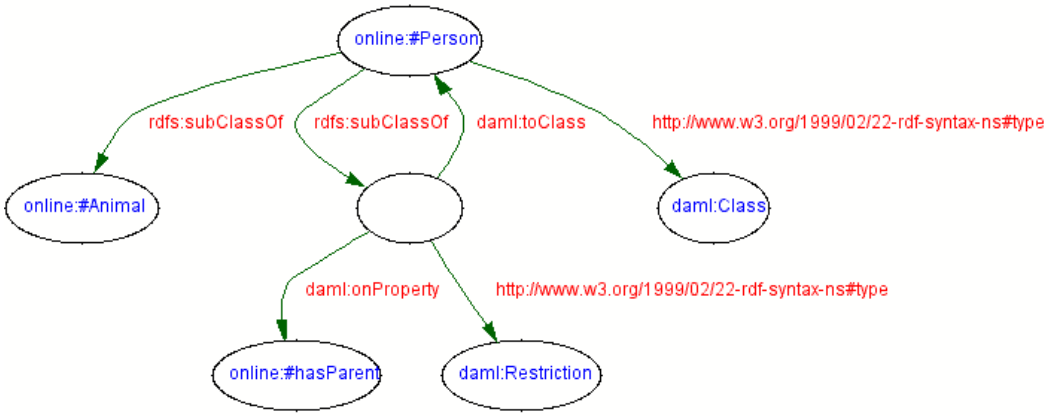

Figure 11: An RDF graph of a DAML class definition.

```
<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 11. In this figure, the nodes are resources that function as subject or object of statements, whereas the arrows represent properties.

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 5.2.1.

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[14] — 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 the superclass- and superproperty hierarchies in the ontology. For this materialization, the entailment and closure rules in the RDF Model Theory[15] can be used.

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

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

5.2.4 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 5.2.1, 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 10). 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 Summary

When ontologies are used in a distributed and dynamic context, versioning support is essential ingredient to maintain interoperability. In this report we have analyzed the versioning relation, described its aspects, and proposed an approach and for the versioning of distributed ontologies. We also described an identification mechanism and finally depicted a computer system that can help users to manage changes in online ontologies.

The analysis of a versioning relation between ontologies revealed several dimensions of it: the descriptive **meta-data**, the **conceptual relations** between constructs in the ontologies, and the **transformations** between the ontologies themselves.

The versioning approach consist of a template for the specification of the relation between ontology versions. The template contains placeholders for all the different dimensions of the version relation. This multi-dimensional 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.

A second element of the approach is an ontology of change operations that allows a functional complete specification of the transformations between ontology versions. We showed this ontology and illustrated how it can be used to represent transformations in RDF.

Rules that specify the effect of individual changes on different tasks form a third element of the approach. We annotated part of the change operations in the ontology with the effect of such changes on two tasks.

A final element of the versioning approach is a system that helps to fill in the template for version relations. 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 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 approach 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 building blocks that we have 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] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies of database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[3] 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.

[4] 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.

[5] 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.

[6] T. Berners-Lee. Generic resources, 1996. Design Issues.

[7] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 2396: Uniform Resource Identifiers (URI): Generic syntax, Aug. 1998. Status: DRAFT STANDARD.

[8] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[9] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. `http://www.w3.org/TR/REC-xml-names/`, Jan. 1999.

[10] D. Brickley, R. Guha, and A. Layman. Resource description framework (RDF) schema specification. W3C proposed recommendation. `http://www.w3c.org/TR/WD-rdf-schema`, Mar. 1999.

[11] D. Brickley and R. V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. Candidate recommendation, World Wide Web Consortium, Mar. 2000.

[12] 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.

[13] V. K. Chaudhri, A. Farquhar, R. Fikes, P. D. Karp, and J. P. Rice. OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 600–607, Menlo Park, July 26–30 1998. AAAI Press.

[14] O. Corcho and A. Gómez-Pérez. A roadmap to ontology specification languages. 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 LNCS 1937 in Lecture Notes in Artificial Intelligence, pages 80–96, Juan-les-Pins, France, Oct. 2–6, 2000. Springer-Verlag.

[15] D. Fensel. Ontologies: Dynamic networks of formally represented meaning. Rejected for SWWS, 2001.

[16] 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.

[17] D. Fensel and M. A. Musen. The semantic web: A new brain for humanity. *IEEE Intelligent Systems*, 16(2), 2001.

[18] R. W. Fergerson, N. F. Noy, and M. A. Musen. The knowledge model of protégé-2000: Combining interoperability and flexibility. 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 Lecture Notes in Artificial Intelligence, Juan-les-Pins, France, Oct. 2–6, 2000. Springer-Verlag.

[19] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2), 1993.

[20] 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.

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

[22] M. Klein. Combining and relating ontologies: an analysis of problems and solutions. In A. Gomez-Perez, M. Gruninger, H. Stuckenschmidt, and M. Uschold, editors, *Workshop on Ontologies and Information Sharing, IJCAI'01*, Seattle, USA, Aug. 4–5, 2001.

[23] 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.

[24] 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/.

[25] D. Marco. *Building and Managing the Meta Data Repository: A Full Lifecycle Guide*. Wiley & Sons, 2000.

[26] 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.

[27] N. F. Noy and M. Klein. Ontology evolution: Not the same as schema evolution. *Knowledge and Information Systems*, 5, 2003. in press.

[28] D. E. Oliver. *Change Management and Synchronization of Local and Shared Versions of a Controlled Vocabulary*. PhD thesis, Stanford University, 2000.

[29] B. Omelayenko and D. Fensel. A two-layered integration approach for product information in B2B E-commerce. In G. P. K. Bauknecht, S.-K. Madria, editor, *Proceedings of the Second International Conference on Electronic Commerce and Web Technologies (EC WEB-2001)*, volume 2115 of *LNAI*, pages 226–239, Sept. 2001.

[30] J. F. Roddick. A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383–393, 1995.

[31] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, Sept. 1990. Also published in/as: Bellcore, TM-STS-016302, Jun.1990.

[32] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. In *13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02)*, Sigüenza, Spain, Oct. 1–4, 2002.

[33] V. Ventrone and S. Heiler. Semantic heterogeneity as a result of domain evolution. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 20(4):16–20, Dec. 1991.

[34] 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.