Ontology Change Detection using a Version Log

Peter Plessers*, Olga De Troyer

Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium
{Peter.Plessers, Olga.DeTroyer}@vub.ac.be

Abstract. In this article, we propose a new ontology evolution approach that combines a top-down and a bottom-up approach. This means that the manual request for changes (top-down) by the ontology engineer is complemented with an automatic change detection mechanism (bottom-up). The approach is based on keeping track of the different versions of ontology concepts throughout their lifetime (called virtual versions). In this way, changes can be defined in terms of these virtual versions.

1 Introduction

With the emergence of the Semantic Web [1], a new dimension has been added to the World Wide Web (WWW). Before, the information and functionality provided on the WWW was primarily tailored towards human interpretation, limiting the possibilities for machine processing. The Semantic Web has been proposed as an answer to these shortcomings by making the semantics of the web content explicit. Two major building blocks are used to realize this vision: ontologies as a formal, explicit specification of a conceptualization [2], and semantic annotations connecting web content and ontologies to enrich the web content with semantic information. Besides containing semantically annotated web pages, the Semantic Web is also a true 'web of ontologies' meaning that ontologies are interconnected, as they are reused and linked to each other.

The subject of this paper concerns ontology evolution. Evolution is an intrinsic part of the Semantic Web: alterations in a particular domain, changes of user requirements or corrections of design flaws, they all may induce changes to the corresponding ontologies and to semantic annotations. Moreover, changes to one ontology may have implications on many depending artifacts (other ontologies, annotations, applications, etc. based on the changed ontology) [3]. The manual handling of this evolution process of ontologies in a distributed, decentralized environment as the Semantic Web is not feasible as it is a too laborious, time intensive and complex process [12]. Therefore, it is vital that an approach is provided guiding the ontology engineer in this complex ontology evolution process.

To be able to understand the modifications applied to an ontology, the changes should be formally represented and captured. This is usually done through an evolution log listing all applied changes. Furthermore, the change representation used

* This research is partially performed in the context of the e-VRT Advanced Media project (funded by the Flemish government) which consists of a joint collaboration between VRT, VUB, UG, and IMEC.

should be sufficient expressive (i.e. able to specify all possible changes to an ontology), and should support different levels of granularity (i.e. fine-grained changes (e.g. the creation of a single class) opposed to coarse-grained changes (e.g. the movement of sibling classes to a different parent)). In current approaches, the evolution log is a direct result of the changes requested by the ontology engineer. In this paper, we argue that such an approach may lead to a limited evolution log, missing valuable information. This makes it harder for (other) users and machines to understand and interpret the ontology modifications. Therefore, we propose an ontology evolution approach combining a top-down and a bottom-up approach. This means that the manual request for changes by ontology engineers (top-down) is complemented with an approach of automatic change detection (bottom-up).

The paper is structured as follows. Section 2 presents an overview of current practices in the domain of ontology evolution. Section 3 gives a general outline of the ontology evolution approach focusing on the different phases of the approach. Section 4 introduces the *version log*, which forms the basis of our approach. In the subsequent sections, the relevant phases of our approach are elaborated in more detail: section 5 discusses the Change Request phase, section 6 presents the Change Implementation phase, the Change Detection mechanism is given in section 7, while section 8 presents the Change Recovery phase. Finally, section 9 discusses the advantages of our approach and provides conclusions.

2 Ontology Evolution

In this section, we give an overview of current practices in the domain of ontology evolution. Stojanovic [11] has defined ontology evolution as the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to depending artifacts. In [10] the authors identified a possible evolution process. The core phases of this process can be summarized as follows:

- *Change representation*: in the context of a change request, the necessary changes have to be identified and represented in a suitable format.
- *Semantics of change*: changes to an ontology can induce inconsistencies in other parts of the ontology or to other depending artifacts. The task of this phase is to solve these inconsistencies by requesting new deduced changes.
- *Change propagation*: the task of this phase is to bring all dependent artifacts in a consistent state by propagating changes to these depending artifacts.
- *Change implementation*: this phase is used to inform an ontology engineer about all the consequences of a change request, to apply all requested and deduced changes and to keep track of all these applied changes in an evolution log.

To represent changes, they introduced in [6] three levels of abstractions of ontology changes for the KAON language. They distinguished: elementary changes (modifications to one single ontology entity), composite changes (modifications to the direct neighborhood of an ontology entity) and complex changes (modifications to an arbitrary set of ontology entities). Also Klein [5] makes a similar taxonomy for the OWL language for which he defines both basic and complex change operations. Basic change operations are changes to one single ontology entity whereas complex change

operations are a mechanism for grouping basic change operations together to form a logical unit. The set of elementary changes and basic change operations (further called *basic changes*) is exhaustive as it is derived from the underlying ontology language; the set of composite changes, complex changes and complex change operations (further called *composite changes*) is infinite as new composite changes can always be defined [5]. The benefit of composite changes is that ontology engineers can formulate their change requests at a higher-level of abstraction, corresponding to their mental model of the change, instead of forcing them to think in terms of individual basic changes.

In [4] the usefulness of a composite change detection approach was already indicated. They introduced a detection mechanism based on rules and heuristics to detect composite changes between two ontology versions (V_{old} and V_{new}). While their approach is applicable in specific cases, in general, the approach has serious limitations:

- The approach requires that V_{old} is still available, because detection rules rely on both V_{old} and V_{new}. Unfortunately, when an ontology is modified, the original version is often no longer available.
- Multiple changes to V_{old} may interfere possibly invalidating defined change detection rules. Take for example the composite change 'moveSiblings' (representing the movement of all siblings to a different parent). A detection rule can be formulated checking if all siblings of a parent *A* in V_{old} have a new parent *B* in V_{new}. Assume that after the move of the siblings, one of the siblings was removed. This would mean that the rule, as formulated, no longer applies. Nevertheless, the 'moveSiblings' change did occur.

The authors of [4] try to overcome these problems by introducing heuristics to change the precise criteria of the rules to approximations. While heuristics may provide the ontology engineer with some flexibility in the rule definitions, it is clear that it doesn't offer a bullet-proof solution as it makes the detection process imprecise and unpredictable.

We argue that, when the ontology engineer solely specifies changes manually, the log of changes may be missing valuable information. This is because of the following reasons:

- It is not always trivial for ontology engineers to select the intended composite change they want to apply due to the complexity involved. Instead they rely on basic changes to achieve step by step the desired result, evaluating the progress after each step. As a consequence, the intended composite change will not be listed in the evolution log.
- A same ontology modification can be achieved in different ways, using composite changes that may differ in level of granularity (and therefore also have different semantics) (e.g. 'moveClasses' and 'moveSiblings'). The ontology engineer will only select one change, meaning that the others will not be listed in the evolution log.
- Meta-changes (information about changes) are valuable to understand occurred ontology modifications as they define the implication of a change. They are, unfortunately, not useful for ontology engineers, as they don't specify 'what' has to change. Therefore, they don't get listed in the log of changes.
- The number of possible composite changes is infinite. Nevertheless, ontology engineers only use a finite number of these composite changes. If a fixed set of com-

posite changes is defined, this means that users are restricted to this set to understand the occurred modifications, although other composite changes may be more appropriate for them.

3 Overview of the Approach

In this section, we will give an overview of the different phases of our ontology evolution approach. Some of the phases resemble phases from the evolution process proposed in [10], but the incorporation of a change detection mechanism has influenced these phases. The five phases of our approach are: (1) Change Request, (2) Change Implementation, (3) Change Detection, (4) Change Recovery and (5) Change Propagation. An overview of the phases is shown in Figure 1.

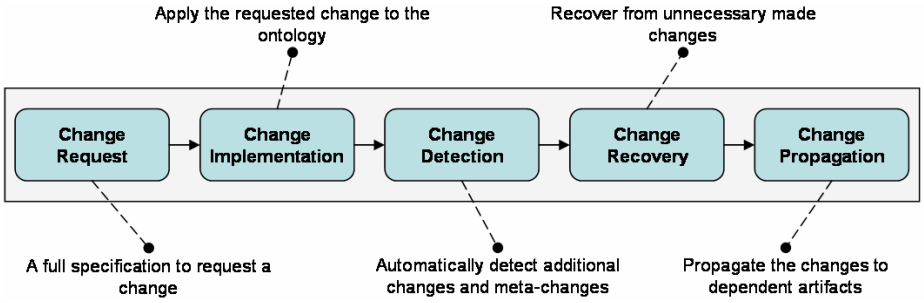

Fig. 1. Five phases of the ontology evolution approach.

The purpose of the different phases is summarized as follows:

1. *Change Request:* In this phase, it is specified which changes need to be applied to the ontology. The phase is divided into two steps. In the first step, the ontology engineer specifies the request for change in terms of basic and composite changes. In the second step, it is checked whether the ontology remains consistent if the requested change would be applied. If this is not the case, new changes (called deduced changes) are added to the change request to solve the inconsistencies. Note that this is an iterative process: new deduced changes may result in additional deduced changes. The result of this phase is a *complete change request specification* composed of requested and deduced changes that transform an ontology from one consistent version into another consistent version. In our example (see Figure 2), two change requests are specified: (1) 'removeSubtype(*B*, *A*)' with deduced change 'removePropertyInstantiation(*p*, *I*, *"abc")*', and (2) 'addSubtype(*B*, *C*)'. This phase is further elaborated in Section 5.
2. *Change Implementation:* This phase takes as input the complete change request specification of the previous phase, and executes the specified changes on the ontology (see Figure 2b+c). We keep track of all changes applied through an *evolution log*, i.e. a log that stores all changes applied. A detailed overview of this phase is presented in Section 6.

3. *Change Detection:* In this phase, it is checked whether other (composite) changes (besides the one specified in the change request) or meta-changes occur as a consequence of the ontology modification. This is done by comparing *change definitions* to the modifications kept in *a version log* (see section 4). The change is added to the *evolution log*, when a combination of modifications of the version log meets the definition of that particular change. E.g. for Figure 2c, a composite change 'changeSubclassRelation' can be detected. We discuss this phase in detail in Section 7.
4. *Change Recovery:* In this phase, the deduced changes from the Change Request phase are checked and possibly need to be revised. We clarify this with the example. When we specified during change request to remove the subclass relation between *B* and *A*, a deduced change was added to remove the property instantiations of *p* for *I* to maintain consistency (Figure 2b). When we later on created a new subclass relation between *B* and *C*, we detected that together both changes form a composite change (e.g. changeSubclassRelation) and that the remove of the property instantiation *p* of instance *i* was unnecessary. Therefore this deduced change needs to be revised (see Figure 2d). This results in a new iteration of the evolution process. A detailed description of this phase is given in Section 8.
5. *Change Propagation:* In this phase, depending artifacts are brought into a consistent state by propagating changes listed in the evolution log to these depending artifacts. Due to space limitations as well as because the focus of this paper is on the change detection aspect, this phase is not further elaborated in this paper. A detailed approach concerning change propagation is described in [7, 8].

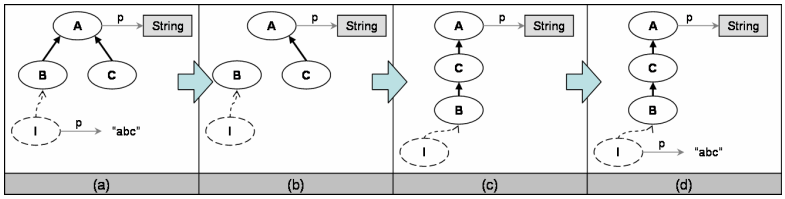

Fig. 2. An example illustrating an evolution process.

For this paper, we assume that an ontology is defined in terms of classes, properties and individuals. The definition of classes, properties and individuals is specified by instantiating either built-in or user-defined properties. We therefore define an ontology as a five-tuple $O = (C, P, I, PI, D)$ where:

C	is the set of Classes
P	is the set of Properties
I	is the set of Individuals
PI	is the set of Property Instantiations
D	is the set of all data values

A property instantiation $pi \in PI$ is a three-tuple $pi = (p, s, t)$ where:

$p \in P$	is a property
$s \in C \vee s \in P \vee s \in I$	is the source of the property instantiation
$t \in C \vee s \in P \vee s \in I \vee t \in D$	is the target of the property instantiation

4 Version log

Before discussing the different phases of our approach, we first present one of the key elements in our ontology evolution approach i.e. the *version log*. This log keeps track of the different versions, called *virtual versions*, an ontology concept passes during its lifetime: starting from the creation of the concept, over its modifications until its retirement. Note the difference between the version log and the evolution log: the former lists the different versions of the ontology concepts, the latter lists the interpretations of these versions in terms of changes. We use the version log to keep ontologies consistent; to serve as the basis for the definition of changes; and as source for change detection. We will first explain the structure and the concepts used in this log. The concepts used in this log are defined by means of an ontology, called the *version ontology*, and discussed in detail in subsection 4.1. In subsection 4.2, we introduce the *change definition language*. The changes, used by ontology engineers to specify their change requests, are defined in terms of this language. Change definitions are treated in subsection 4.3.

4.1 Version ontology

The version log captures, for each concept of the ontology, its different versions. Each version represents the definition of the concept at a moment in time. For each class, property and individual that is created in the ontology, we create an associated instance in the version log. This instance is an instance of the *EvolutionConcept* class. Such an EvolutionConcept instance keeps, besides a reference to the concept in the ontology (for which it was created – called the referred concept), a list of past and current versions of the referred concept.

Whenever a change request for a concept in the ontology is executed, a new *Version instance* is added to the associated EvolutionConcept instance, representing the new version of the referred concept. Such a Version instance has (1) a transaction time property i.e. the moment in time the modification was applied to the ontology *(hasTransactionTime)*, (2) a *causes* (and inverse *causedBy*) property to express which version causes which other versions, reflecting the relation between requested changes and deduced changes in the change request, (3) a state property *(hasState)* to reflect the state of the version (pending or confirmed) and (4) optionally (if it exists) the ID of the referred concept *(hasID)*. Figure 3 gives an overview.

To capture a version of a concept, we have to file all its property instantiations that together form its definition. To do this independently of the ontology language used, we have defined a number of classes and properties to capture the most common ontology language constructs (e.g. complement, union and intersection of classes; symmetric and transitive characteristics of properties; etc.). We have defined the classes *IndividualVersion*, *PropertyVersion* and *ClassVersion* to capture the version of respectively individuals, properties and classes. Because space is limited, we cannot describe the complete version ontology, therefore we only discuss some parts. The interested reader is referred to the full specification of the version ontology[1].

[1] See http://wise.vub.ac.be/ontologies/versionontology.owl

For an IndividualVersion we specify that the referred individual is an instance of a certain EvolutionConcept instance *(instanceOf)* and capture the user-defined PropertyInstantiations that form the definition of the individual *(hasPI)*. For a PropertyVersion we can specify for the referred property (among other properties) the domain and range *(hasDomain, hasRange)*. Also cardinality and value constraints may be specified *(hasConstraint)*. For a ClassVersion we can specify for the referred class (among other properties) a subtype relation *(subTypeOf)*, possible cardinality and value constraints *(hasConstraint)*, and an enumeration of individuals *(enumerates)* that together form the definition of the class.

Note that for the cardinality and value constraints mentioned above, we make a distinction between global and local constraints, referring to the scope of the constraint. Global constraints apply to every instantiation of a given property. Local constraints only hold for those instantiations of a given property when used in a particular class.

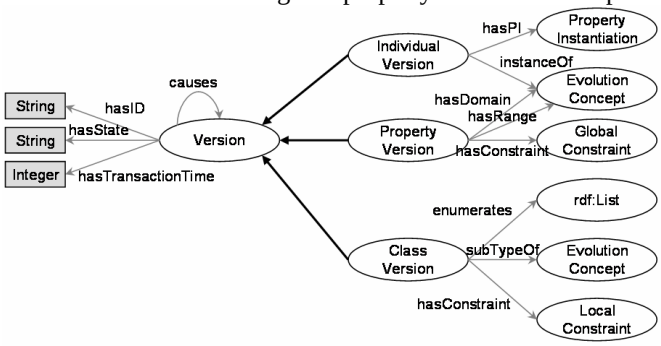

Fig. 3. The Version concepts of the version ontology.

An extract of an example version log is given below. The extract shows an EvolutionConcept representing the different versions a class ‘Student’. The first version represents the initial version of the class. In the second version, we see that the class includes a subtype relation. Note that the version log doesn’t specify ‘what’ has changed; it only lists the successive versions.

```
<EvolutionConcept rdf:ID="fd42cc20">
  <refersTo rdf:resource="…/university#Student"/>
  <hasVersion>
    <ClassVersion rdf:ID="389a99b0">
      <hasTransactionTime>624</hasTransactionTime>
      <hasState>confirmed</hasState>
      <hasID>Student</hasID>
    </ClassVersion>
  </hasVersion>
  <hasVersion>
    <ClassVersion rdf:ID="389a99b1">
      <hasTransactionTime>628</hasTransactionTime>
      <hasState>confirmed</hasState>
      <hasID>Student</hasID>
      <subtypeOf rdf:resource="#fd42cc22" />
```

```
    </ClassVersion>
  </hasVersion>
</EvolutionConcept>
```

4.2 Change Definition Language (CDL)

The version log uses an explicit timeline for the different versions. The 'hasTransactionTime' sequentially orders all versions across all EvolutionConcepts. Note that versions originating from the same change request (i.e. user-specified and deduced changes) will have the same transaction time. The order between such versions is defined by the 'causes' and 'causedBy' properties. As previously mentioned, these properties define which version causes which other versions, reflecting the relation between requested changes and deduced changes. This information is required to be able to undo changes (see Section 8). Figure 4 shows the timeline (T_A and T_B) for two EvolutionConcept instances *A* and *B*. The transaction times of the different versions refer to the timeline *T*. A variable *cv* refers to the current version of a concept. If $n \in \mathbb{N}$ specifies the total amount of versions of one EvolutionConcept, than we can use cv - *a* (where $a \in$ N, a = n) to refer to the $(n - a)^{th}$ version of that concept. We also define a variable cv_p (where $p \in$ P). This variable takes only those versions into account where the instantiation of the given property *p* was changed; cv_p refers to the last one of these versions, cv_p-1 to the previous one, etc.

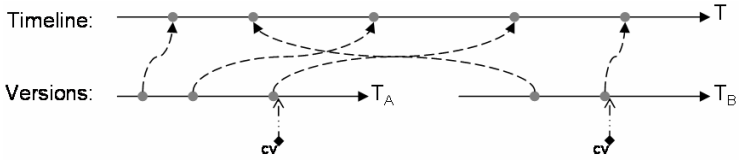

Fig. 4. Timeline introduced by the version log.

This time aspect allows us to check properties of past versions of ontology concepts [9]. This is done by means of *conditions*. These conditions are used to formulate change definitions (see section 4.3). The conditions are resolved using pattern matching. We use the following syntax to define conditions on versions (Note that *V* defines a set of variables):

```
<property>(<source>, <target>, [<version>])
```

where

- <property> is the property we want to retrieve;
- <source> is the source of the queried property or a variable that substitutes the source. <source> ∈ C or <source> ∈ P or <source> ∈ I or <source> ∈ V;
- <target> is the target of the queried property or a variable that substitutes the target. <target> ∈ C or <target> ∈ P or <target> ∈ I or <target> ∈ D or <target> ∈ V;
- <version> is a reference to a version using the cv or cv_p variable or a variable that substitutes the version. Omitting a version reference means we refer to the

current version of the <target> (cv). <version> ::= $cv_{[<property>]}$[- <a>] (where <a> $\in$ $\mathbb{N}$) or <version> $\in$ V.

We illustrate this with an example. Table 1 shows three versions of an individual *i*. In the first version, *i* is an instance of 'Student'. In a second version, *i* becomes an instance of 'Researcher', and in version three *i* "publishes a first article".

Table 1. Different versions of an example individual i.

Versions	Statements
1st version	instanceOf(*i*, 'Student')
2nd version	instanceOf(*i*, 'Researcher')
3rd version	instanceOf(*i*, 'Researcher')
	publishes(*i*, 'article_001')

The following are two conditions:

```
Condition 1: instanceOf(i, 'Researcher', cv - 1)
Condition 2: instanceOf(i, 'Researcher', cv_instanceOf - 1)
```

The first condition allows to check if the individual $i \in$ I was an instance of the concept 'Researcher' during the previous version of *i*. The second condition allows to check if during the previous version of the 'instanceOf' property instantiation, *i* was an instance of 'Researcher'. The first condition returns 'true' (cv - 1 refers to the 2nd version), the second one returns 'false' (because the $cv_{instanceOf}$ - 1 refers to 1st version).

4.3 Change Definitions

The Change Definition Language introduced in the previous subsection is used to specify *change definitions*. A *change* is an interpretation of an ontology modification i.e. the definition of a change formally specifies the modifications that correspond with this change. These change definitions are used in two ways in our approach. Firstly, ontology engineers specify their change requests in terms of change definitions. The definition of the change specifies how the ontology has to change (see section 5). Secondly, these same change definitions allow detecting other changes (not specified during change request). This is possible because we are able to verify whether some change definitions are satisfied by the modifications that occurred (see section 7).

Both [5] and [6] distinguish basic and composite changes where composite changes are defined in terms of basic and other composite changes (i.e. a functional definition). In our approach, we define changes declaratively in terms of changing versions. It is exactly this declarative definition of changes that will allow us to detect changes based on the versions kept in the version log.

We make a distinction between *changes* (i.e. define 'what' has changed) and *meta-changes* (i.e. define the implications of a change). Changes are further classified into *basic* and *composite* changes. Basic changes can be expressed as a modification of exactly one element of the version log by only imposing conditions on the changing

element (e.g. createSubtypeOf, deleteHasDomain, etc.). These basic changes are sufficient to express any desirable change. A Composite change is either a modification of exactly one element but also imposes conditions on other elements, or a modification of more than one element.

As examples, we define the basic change 'addDomain', the composite change 'moveUpDomain' and the meta-change 'restrictProperty'.

The basic change 'addDomain' adds *A* as domain of property *p*. Note that the definition only expresses a condition on the element that changes (hasDomain of a property *p*). The definition specifies that *A* is the domain of *p* in the current version, but wasn't in a previous version.

$$\forall\ p \in \mathrm{P},\ A \in \mathrm{C}: \mathrm{addDomain}(p, A) \leftarrow \neg\mathrm{hasDomain}(p, A, \mathrm{cv}\text{-}1) \wedge \mathrm{hasDomain}(p, A, \mathrm{cv})$$

The composite change 'moveUpDomain' moves the domain of a property *p* up in the hierarchy of classes. So if *p* has as domain the class *A*, the domain will be changed to a superclass of *A*. Note that this change expresses an additional condition on the subtype relation between *A* and *B*.

$$\forall\ p \in \mathrm{P},\ A, B \in \mathrm{C}: \mathrm{moveUpDomain}(p, A, B) \leftarrow \mathrm{hasDomain}(p, A, \mathrm{cv}_{\mathrm{hasDomain}}\text{-}1) \wedge \neg\mathrm{hasDomain}(p, A, \mathrm{cv}_{\mathrm{hasDomain}}) \wedge \neg\mathrm{hasDomain}(p, B, \mathrm{cv}_{\mathrm{hasDomain}}\text{-}1) \wedge \mathrm{hasDomain}(p, B, \mathrm{cv}_{\mathrm{hasDomain}}) \wedge \mathrm{subtypeOf}(A, B, \mathrm{cv})$$

As an example of a meta-change, we define the 'constraintWeakening' meta-change indicating a weakening of constraints as a consequence of the change. Different modifications to an ontology can lead to a weakening of constraints, e.g. the raise of a cardinality constraint or the extension of a value constraint, a change to a subtype relation, etc. This means that multiple definitions exist for the 'constraintWeakening' each reflecting different causes. For this example, we define the 'constraintWeakening' meta-change in the case of a replacement of class *B* as the domain of property *p* by class *A*, where *A* is a superclass of *B*. This is a weakening of constrains as first only instances of class *B* could instantiate property *p*, now also individuals of class *A* can. Note that the definition uses the 'moveUpDomain' change definition.

$$\forall\ p \in \mathrm{P}: \mathrm{constraintWeakening}(p) \leftarrow \exists\ A, B \in \mathrm{C}: \mathrm{moveUpDomain}(p, B, A)$$

5 Change Request

Ontology engineers express their change requests (i.e. 'what' has to change) in terms of the change definitions (as defined in section 4.3). Applying these changes directly to the ontology may cause inconsistencies, meaning that the ontology would no longer conform to the constraints imposed by the ontology language used. To avoid this, we

first process the requested change in the version log. To check if a requested change can be applied, the conditions in the change definitions are tested. The conditions in the change definition that refer to past versions form a pre-condition that needs to be satisfied. If this pre-condition is not satisfied, the requested change cannot be applied and the change request will be rejected. Otherwise, the changes are recorded in the version log by adding new versions to the EvolutionConcepts referring to the ontology concept to be changed so that the new current version satisfies the post-conditions in the change definition. The conditions in the change definition that refer to current versions form a post-condition. Note that these new versions are marked in the version log with the value 'pending' for the property 'hasState' indicating that they are not yet applied to the ontology itself.

Next, we have to check whether the ontology would remain consistent if we would apply these new versions to the ontology itself. The consistency check is based on a consistency model i.e. a model that restricts the version ontology so that it conforms to the constraints imposed by the ontology language used (explained in section 5.1). If it turns out that the requested change would cause inconsistencies, additional changes should be added to the change request to solve these inconsistencies. Note that in our approach, it is currently still the responsibility of the ontology engineer to specify the additional changes. Keep in mind that the deduction of additional changes is an iterative process. Every new deduced change creates a new version in the version log, and the consistency check is reapplied. The 'cause' and 'causedBy' properties are used to express the causal relation between versions to reflect the causal relation that exists between requested and deduced changes.

The result of this phase is a *complete change request* (consisting of requested and deduced changes) that transforms an ontology from a consistent version to another consistent version. The actual implementation of the complete change request to the ontology is done in the next phase (Change Implementation).

5.1 Consistency Model

To check for consistency, we make use of a *consistency model*. Such a consistency model is a formal meta-model that restricts the version ontology so that it is conforms to the constraints imposed by the ontology language used. This means that whenever the latest version stored in the version log conforms to the consistency model, the requested changes can be applied. Note that different consistency models may exist for different ontology languages or different variants of an ontology language (e.g. OWL Lite and OWL DL).

To clarify the consistency model, we give a number of example constraints that represent constraints from OWL.

- In OWL, a subtype relation can only be defined between either two classes (subClassOf) or two between two properties (subPropertyOf):

$$\forall\ a, b: \mathrm{subTypeOf}(a, b) \Rightarrow (a \in C \wedge b \in C) \vee (a \in P \wedge b \in P)$$

- In OWL, a property may have a domain and the domain of a property is a class:

$\forall\ a, b$: hasDomain(a, b) $\Rightarrow a \in P \wedge b \in C$

- In OWL DL and Full, the value of a cardinality constraint is a non-negative integer:

  $\forall\ C, v$: hasCardinality(C, v) $\Rightarrow v \in \mathbb{N} \wedge v \geq 0$

- In OWL Lite however, the previous constraint would not hold as the value of a cardinality constraint is in this case restricted to 0 or 1. The previous constraint is replaced by the following:

  $\forall\ C, v$: hasCardinality(C, v) $\Rightarrow v \in \{0, 1\}$

6 Change Implementation

The objective of this phase is twofold. The first objective of the change implementation phase is to synchronize the ontology with the latest version of EvolutionConcept(s) in the version log, i.e. the requested changes need to be applied to the ontology. The second objective is to add the changes listed in the change request to the *evolution log*. This log keeps track of all applied changes and gives an overview of the complete evolution history of the ontology in terms of changes.

The process of applying changes to the actual ontology is quite simple. The concepts that need to modify (obtained using the *refersTo* property in the version log) just have to be replaced by the current version specified in the version log. To perform the synchronization, a mapping between the concepts of the version ontology and the elements of the chosen ontology language needs to be provided. Figure 5 shows an example.

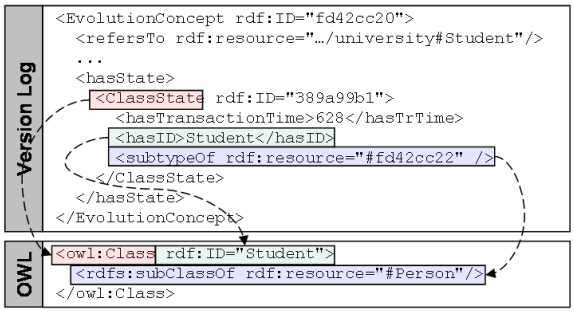

Fig. 5. Example mapping between Version log and an OWL ontology.

7 Change Detection

After the execution of the previous phases, the version log has been modified, the ontology has been synchronized with the version log and the requested changes have been added to the evolution log. The changes specified in the change request are only

one way to see (interpret) the modification made to the ontology. From all the changes defined (by means of change definitions), other definitions may also be satisfied by the ontology modification and therefore these changes have also occurred. To detect these additional changes, we use the following procedure. For each change definition specified, we check if its definition is satisfied. As change definitions are given in terms of conditions on the version log, a change definition is satisfied if all conditions of its definition are satisfied. The changes, whose definitions are met, are subsequently added to the evolution log. Note that this change detection process isn't limited to the ontology owner, but can be performed for all maintainers of artifacts depending on the ontology. This makes it possible for maintainers of depending artifacts to specify their own set of change definitions, independently of the set of change definitions of the ontology owner or other maintainers.

Notice that this change detection process is particularly flexible i.e. the detection process is not dependent on the steps taken to achieve a particular change neither on the order of these steps. Figure 6 illustrates this with two situations where both the 'moveUpDomain' change will be detected. In the first situation (1), the domain of property *p* is changed from class *B* to class *A* being a superclass of *B*. This change confirms to the definition of 'moveUpDomain' (see Section 4.3). In the second situation (2), the domain of property *p* is changed from class *B* to class *A*. This change doesn't conform to the definition of 'moveUpDomain' as the subtype condition is not met. However, when in a next step, the subclass relation between *B* and *A* is added, this modification will result in the detection of the 'moveUpDomain' change as all conditions are now met.

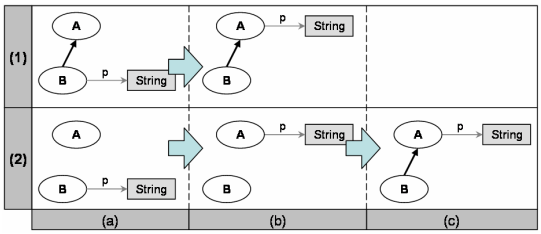

Fig. 6. Two examples illustrating the change detection process.

8 Change Recovery

We start with an example. Assume for Figure 6(2) an instance *i* of class *B* with a property instantiation of *p*. When the domain of property *p* is changed to class *A*, the ontology would become inconsistent because the property instantiation of *p* for *i* is no longer valid. To overcome this inconsistency, the change request will be extended with an additional change to remove the property instantiation from *i*. When in the next step, the subtype relation is added between *B* and *A*, we will detect the 'moveUpDomain' change (see previous section). It becomes now clear that the removing of the property instantiation of *p* from *i* was not necessary. This example illustrates the necessity to be able to recover from deduced changes when detecting changes.

When a composite change is detected, the change recovery process is as follows:

1. As presented in section 4.3, the definition of a change is specified in terms of conditions. For each property, possible pre-conditions and post-condition are specified in terms of past and current versions. In this first step, from the versions of the version log that satisfy the definition of the detected change (in this example 'moveUpDomain'), we select those versions that satisfy the conditions in the change definition that form the post-condition (Figure 6(2b + c)).
2. In a second step, all versions that are caused by one of the selected versions are undone. In our example, this means undoing the remove of property instantiation *p* for *i*. This is possible by following the 'causes' property from the selected versions in the version log. In this way, we undo all deduced changes. Undoing changes is trivial as we can easily return to the previous version (found in the version log).
3. The version log is checked for inconsistencies. If it appears to be consistent, the changes will be applied to the ontology. If not, additional changes need to be formulated by the ontology engineer to solve any inconsistency. This step is repeated until consistency is reached. Note that this step is similar to the process described in section 5. In our example, the ontology is consistent as the property instantiation *p* of individual *i* became valid by adding the subtype.

9 Discussion and Conclusions

The change detection mechanism as presented here has a number of advantages:

1. *Implicitly detection of changes.* It is not always easy for ontology engineers to select the correct, intended composite change that reflects the modifications they have in mind. Therefore, they will opt for basic changes to achieve step by step the desired result. In the end, they might have applied a composite change, not realizing they did. The detection mechanism proposed is able to automatically detect these implicitly executed composite changes.
2. *Allowing different levels of granularity.* Several composite changes, with different levels of granularity (and also different semantics), may result in the same ontology modification. Consider for example the composite changes 'moveClasses' and 'moveSiblings'. 'moveSiblings' provides more semantics than 'moveClasses' (difference in granularity). The same modification can be achieved using either of these changes. When an ontology engineer opts for the 'moveClasses' change to actually execute a 'moveSiblings' change, valuable information is lost because the most accurate change is not registered. The change detection mechanism overcomes this problem, as the more accurate change will be detected.
3. *Meta-changes are automatically detected.* Meta-changes are not useful for ontology engineers to specify change requests as they don't define 'what' has to change. Furthermore, ontology engineers don't want to be burden with the task of manually specifying them. However, these meta-changes are definitely valuable for understanding occurred ontology changes. The change detection mechanism is able to detect such meta-changes.
4. *Different sets of change definitions.* Ontology engineers may use a fix set of composite change definitions to specify their change requests, although an infinite

number of composite changes may exist. Our approach makes it possible for maintainers of depending artifacts to define additional composite change definitions, which are more appropriate for their purpose. The occurrence of these additional changes can be detected using our change detection mechanism.

As a conclusion, we summarize the contributions of this paper. We have presented a new ontology evolution approach that includes an automatic change detection mechanism. The key element of our approach is the version log, which maintains the different versions of the ontology concepts. Change definitions as well as the consistency model are defined in terms of this version log. Because the version log is independent of the ontology language used, also change definitions are defined independently of the used ontology language. Different ontology languages are supported by defining a proper consistency model and specifying a mapping between the version ontology and the ontology language used.

References

1. Berners Lee, T., Hendler, J., Lassila, O.: The semantic web: A new form of web content that is meaningful to computers will unleash a revolution of new possibilities. Scientific American (2001) 5(1)
2. Gruber, T.: A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition 5(2) (1993) 199-220
3. Klein, M., Fensel, D.: Ontology versioning for the Semantic Web. In Proceedings of the First International Semantic Web Working Symposium (SWWS), Stanford University, California, USA (2001) 75-91
4. Klein, M., Fensel, D., Kiryakov, A., Ognyanov, D.: Ontology versioning and change detection on the web. In 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02), Sigüenza, Spain (2002)
5. Klein, M.: Change Management for Distributed Ontologies. PhD Thesis (2004)
6. Maedche, A., Stojanovic, L., Studer, R., Volz, R.: Managing multiple ontologies and ontology evolution in OntoLogging. In Proceedings of the Conference on Intelligent Information Processing (IIP-2002), Montreal, Canada (2002) 51-63
7. Maedche, A., Motik, B., Stojanovic, L., Managing multiple and distributed ontologies on the Semantic Web. TheVLDB Journal - Special Issue on Semantic Web 12 (2003) 286-302
8. Maedche, A., Motik, B., Stojanovic, L., Studer, R., Volz, R.: An Infrastructure for Searching, Reusing and Evolving Distributed Ontologies, In Proceedings of the Twelfth International World Wide Web Conference (WWW 2003), Budapest, Hungary, ACM (2003) 439-448
9. Plessers, P., De Troyer, O., Casteleyn, S.: Event-based Modeling of Evolution for Semantic-driven Systems. In Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05), Publ. Springer-Verlag, Porto, Portugal (2005)
10. Stojanovic, L., Maedche, A., Motik, B., Stojanovic, N.: Userdriven Ontology Evolution Management. In Proceeding of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW, Madrid, Spain (2002)
11. Stojanovic, L.: Methods and Tools for Ontology Evolution. Phd Thesis (2004)
12. Tallis, M., Gil, Y.: Designing scripts to guide users in modifying knowledge-based systems. In Proceedings of the 14th National Conference on Artificial Intelligence (AAAI/IAAI 1999), Orlando, Florida, USA (1999) 242-249