Semantic Invalidation of Annotations due to Ontology Evolution

Julius Köpke and Johann Eder

Author's Version of

Julius Köpke and Johann Eder “Semantic Invalidation of Annotations due to Ontology Evolution”.
In On the Move to Meaningful Internet Systems: OTM 2011
Lecture Notes in Computer Science Volume 7045, 2011, pp 763-780

The final publication is available at link.springer.com:
http://link.springer.com/chapter/10.1007%2F978-3-642-25106-1_25

Semantic Invalidation of Annotations due to Ontology Evolution

Julius Köpke, Johann Eder

Department of Informatics-Systems, University of Klagenfurt, Austria
firstname.lastname@aau.at http://isys.uni-klu.ac.at

Abstract. Semantic annotations assign concepts of a reference ontology to artifacts like documents, web-pages, schemas, or web-services. When an ontology changes, these annotations have probably to be maintained as well. We present an approach for automatically checking whether an annotation is invalidated by a change in the reference ontology. The approach is based on annotation paths and on the explicit definition of change-dependencies between ontology artifacts. Ontology change-logs and change dependencies are then exploited to identify those annotation paths which require maintenance.

Keywords: Semantic annotation, ontology evolution, annotation maintenance, semantic invalidation, semantic dependencies

1 Introduction

Semantic annotations were developed to assign semantics to various documents, e.g. XML-Schemas, XML documents, web-pages, or web-services by linking elements of these documents to elements of a reference ontology [20],[13]. When the reference ontology evolves, these annotations might have to be maintained as well. To ease the maintenance effort it is highly desirable to identify those annotations which have to be maintained in contrast to those which are invariant to the changes. So the goal of the research reported here[1] was to develop a technique to automatically identify those annotations which need attention as a consequence of a given set of ontology changes. The method should deliver all annotations where the annotations itself or instance data (might) need to be changed and it should return the smallest possible set of annotations (no false negatives and as little false positives as possible). Furthermore, the analysis of the necessity of maintenance should also deliver strategies for changing the annotation, if possible.

The proposal is based on annotation path expressions [14] as a method for semantic annotations. Annotations consists of paths of concepts and properties of the reference ontology. These annotation paths were developed to grasp the

[1] This work was partially supported by the Austrian Ministry of Science and Research; program GEN-AU, project GATIB

semantics more precisely and to provide a pure declarative representation of the semantics opposed to the more procedural lifting and lowering scripts of traditional XML-Schema annotation [13] methods. These annotation paths can automatically be transformed to ontology concepts. The ontology concepts can then be used to create semantic matchings between annotated elements from different schemas or different versions of the same schema. An example for an annotation is */invoice/hasDeliveryAddress/Address/hasZipCode* which could be used to annotate a *zip-code* element in an XML-Schema for invoices. The example assumes that *invoice* and *address* are concepts of the reference ontology and *hasDeliveryAddress* is an object-property and *hasZipcode* is a datatype-property. Another motivation for the development of annotation paths was to provide a better basis for maintaining annotations when the reference ontology is changed. In [14] we have introduced different types of invalidations of the annotations when the ontology changes:

- **Structural invalidation:** An annotation path references concepts and properties of the reference ontology. All referenced concepts and properties must exist in the ontology and basic structural requirements must be met. For example */invoice/hasDeliveryAddress/Address/hasZipCode* gets invalid if the concept *Address* is removed from the reference ontology.
- **Semantic invalidation:** An annotation path is invalid if its semantic representation (an ontology concept) imposes contradictions to the reference ontology. For example */invoice/hasDeliveryAddress/Address/hasZipCode* gets semantically invalid if the domain of *hasDeliveryAddress* is changed to *order* and *invoice* is defined to be disjoint from *order*.

These types of invalidations can be tracked by structural checks and simple reasoning over the ontology representation of the annotation path. When an annotation got invalid it needs to be repaired. Thus, evolution-strategies [19] are required in order to maintain the annotations. For example, if a referenced concept is removed we may use the super-concept for the annotation instead. If it was renamed we need to rename the element in the annotation paths as well. In this paper we will focus on another kind of invalidation: Changes of the semantics of the annotations which lead to a misinterpretation of annotated data without invalidating the annotation structurally or semantically. We will describe this problem in the following section with an illustrative example. We will first present the notion of semantic changes and discuss if such changes to the semantics of an annotation can, still, be derived from the plain ontology in section 2 and describe and define the explicit dependency definitions in section 3 and 4. In section 5 we show how the explicit definitions can be used to track semantic changes. In section 6 we present a prototype-implementation of the approach. Section 7 gives an overview about the related work. The paper concludes in section 8.

2 Semantic Changes and their Automatic Detection

Semantic changes are consequences of changes in the reference ontology which do not invalidate the annotation semantically or structurally, but might lead to misinterpretations. We illustrate such semantic changes with the following example:

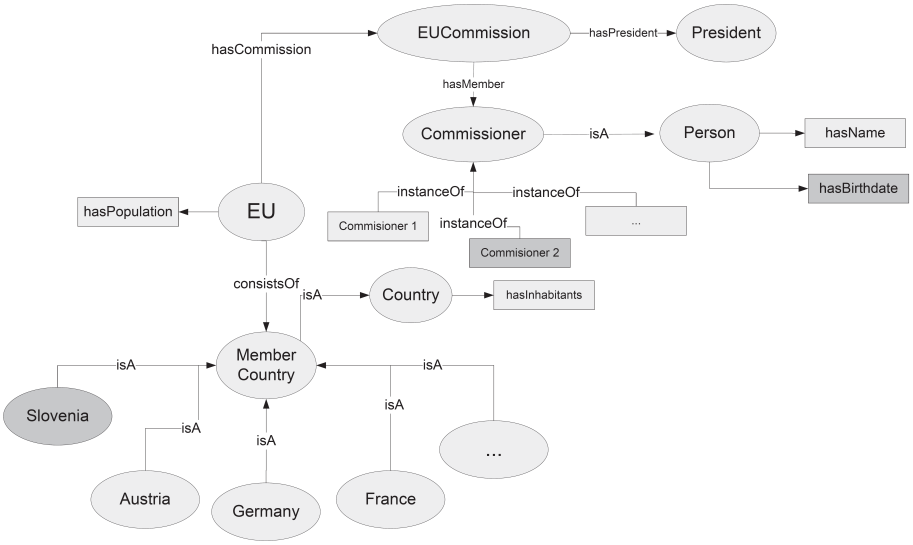

Fig. 1. Example Ontology

In figure 1 an example ontology[2] that represents parts of the European Union is depicted. We now assume that we have an annotation of an XML-Schema-element with the annotation path */EU/hasPopulation*. Some changes were made to the ontology: *Slovenia* was added as an additional *MemberCountry* and an additional *Commissioner*-instance was added and persons now have the property *hasBirthdate*. The differences between the old and the new ontology version are marked in dark-grey. Now obviously the semantics of an annotation */EU/hasPopulation* of some element in an XML Schema are changed because the parts of the European union were changed. This change does not influence the validity of the annotation itself - but the semantics of the annotation has changed. Documents that were annotated with the old ontology version will have a lower population number than documents of the current version. This imposes problems because it leads to the misinterpretation of the data. For example a

[2] The member countries are modeled in form of concepts and specific commissioners as instances in order to show problems that may occur when concepts or instances evolve.

human reader might come to the conclusion that the EU has a higher birthrate after the change. The goal of this paper is the automatic generation of warnings for such changes.
Since an ontology is used to express the semantics of a domain it should be possible to derive the changes of the semantics of annotations automatically. To avoid that each annotation has to be checked, if any ontology element has changed, we need to reduce the set of ontology elements which might invalidate a specific annotation. Ontology views [16] are methods to reduce the size of an ontology. Ontology module extraction [7] techniques can be applied for the same purpose. These methods generate a sub-ontology that only contains relevant elements for a given starting point (set of concepts). The starting point in our case is the semantic representation of an annotation path expression. When such sub-ontologies are created for the old and the new ontology version we can check, if there were changes between the sub-ontology versions. Since the sub-ontologies only contain elements that are relevant for the annotation in question we should be able to significantly reduce the number of false-positives.
Typical methods for the generation of sub-ontologies begin with a concept in question and then add more and more concepts that are related. The relation is expressed in form of sub/superclass relations or object-properties. We will illustrate the general idea of the generation of a sub-ontology with an example:

- The starting point is *EU/hasPopulation*
- This leads the inclusion of the concept *EU*
- The *consistsOf* property on *EU* requires the addition of *MemberCountry*
- *MemberCountry* requires the addition of its super-concept: *Country* with the property *hasInhabitants*
- *MemberCountry* requires the addition of *Slovenia*, *Austria*, *Germany*, ...

When we now compare the view of the old and the new ontology version we can figure out that *Slovenia* was added. We would throw a warning that the semantics of *EU/hasPopulation* was possibly changed. In this example we have assumed that we include all datatype-properties of a concept in the view and all concepts that are in the range of the object properties of the included concepts. In addition, all super- and sub-concepts as well as individuals of the included concepts are added. Unfortunately, such an algorithm would thus, include much more concepts:

- The *hasCommission* property of *EU* requires the addition of *EUCommission*
- The *hasPresident* property of *EUCommision* requires the addition of *President*.
- The *hasMember* property of *EUCommision* requires the addition of *Commissioner*.
- The *Commissioner* concept requires the addition of its instances.
- ....

At the end the entire ontology would be included in the view. Thus, all changes that happened between the old and the new ontology version are assumed to

be relevant for *EU/hasPopulation*. This is certainly not true. In order to avoid this behavior the set of properties that are followed to build the view needs to be much smaller. Thus, strategies are required to choose the proper object-properties that should be followed. But where is the difference between *consistsOf* and *hasCommission*? From where can we know that if we want to build the view for *EU/hasPopulation* we need to follow the *consistsOf* property and that if we want to create the view for *EU/numberOfCommissioners* we need to follow the *hasCommision* property?
Thus, strategies are required in order to keep the view small and meaningful.

3 Requirements for Explicit Dependency-Definitions

As shown in the last section there is typically no knowledge about what may be invalidated semantically by changes since this is not an invalidation of the logical theory (which could be calculated) but a change of the semantics of the annotations. The reason for this is that the ontology does not fully specify the real-world domain. Therefore, a straight forward solution is the addition of the missing knowledge to the ontology. This means sentences like "*The population of the EU is changed when the MemberCountries change*" should be added to the ontology. Obviously this a very wide definition because we have not stated anything about the types of relevant changes. Is it changed when the name of a country changes or only if a specific attribute changes? In general which operations may invalidate our value? The examples of the population of the EU can be described as the aggregation of the population of the member countries. Thus, we need a way to describe such functions. These observations lead to the following requirements for change-dependency definitions:

1. The change-dependent concept or property must be described including the context. For example *hasPopulation* of *EU* and *hasPopulation* of *City* might depend on a different set of ontology elements.
2. The definition of the change-dependency should allow fine grained definitions of dependencies. For example it should be possible to define that *EU/hasPopulation* is dependent on the *population* of the *MemberCountries*. It would not be sufficient to state that it is dependent on *population* in general.
3. It should be possible to define that one artifact is dependent on a set of other artifacts.
4. Multiple dependencies should be possible for one change-dependent concept or property.

According to the first requirement the dependent artifact needs to be specified precisely. This can be realized with the annotation path syntax. Therefore, the path *EU/hasPopulation* defines that the *hasPopulation* property on the concept *EU* is the subject of a dependency definition. The second requirement supposes that not only the subject should be described via path expressions but also the object of a change-definition should be described in terms of path expressions. Unfortunately, the plain annotation path syntax does not fulfill the third requirement. It, therefore, needs to be enhanced with expressions to address sets.

Dependencies on Sets and Aggregations: Some ontology artifact may be change-dependent on a set of other artifacts. In our running example the population of the *EU* depends on the set of *MemberCountries*. More precisely it is not dependent on the set of *MemberCountries* in general but on the sum of the *hasInhabitants* property of each *MemberCountry*. In general, there are different kinds of sets in an ontology: subclasses, sub-properties and instances. Therefore, all those must be expressible. The sum function is only one aggregation function. Typical other aggregation functions are are min, max, count, and avg. In addition to aggregation functions another kind of function over sets is of interest: The value function. It can be used to state than one artifact is directly dependent on the values of a set of other artifacts. The subclasses and instances operator can be used in a path wherever a concept is allowed and the sub-properties operator can be used wherever a property-step is allowed. We will illustrate the ideas with examples:

1. *EU/hasPopulation* is dependent on */EU/consistsOf/sum(subclasses(MemberCountry))/hasInhabitants*.
2. *EU/numberOfCommissioners* is dependent on */EU/hasCommission/ EUCommission/hasMember/count(instances(Commissioner))*.
3. *MemberCountry/hasInhabitants* is dependent on *MemberCountry/subproperties(hasInhabitants)*.
4. */city* is dependent on *value(subclasses(city))*

The first example calculates the sum of all *hasInhabitants* properties of all subclasses of $member - countries$, while the second one just counts the number of *commissioner* instances. The first example defines an abstract sum because the ontology cannot contain any information about the number of inhabitants on class level. It only defines that the value becomes invalid if the ontology structure changes in a way that the function would operate on a changed-set of ontology artifacts. In contrast the second example can return a defined number because it is a simple count operation. In addition, it defines the change-dependency over instances. In this case the ontology may contain instance data. In the third example it is assumed that the *hasInhabitants* property has sub-properties and that a change of the sub-properties will also invalidate *EU/hasPopulation*. Examples for sub-properties could be *hasMalePopulation* and *hasFemalePopulation*.
The last example shows the value function. It defines that elements that are annotated with */city* are change-dependent on all the subclasses of *city*. Therefore, a rename of a subclass of *city* requires a rename of the specific city-element in XML-documents as well.

4 Definition of Change-Dependencies

In order to introduce the proposed change-dependency definitions we will first define our ontology model. We use an abstract ontology model which can be compared to RDFS[2] shown graphically in figure 2. It contains the relevant aspects

of typical ontology languages. Basically an ontology consists of concepts, properties and individuals. Concepts and properties are hierarchically structured. An individual is an instance of a set of concepts. A property has a domain that defines the set of classes that have this property. Properties are divided into object-properties and datatype-properties. Object-properties form relationships between classes and, therefore, have a range that defines the set of classes which are targets of the property. Datatype-properties have a definition of the data type. Properties are modeled on the class level while an instantiation of a property is done on instance-level using property assertions. Concepts may restrict the usage of properties. A restriction has a type and a value. The type can be a typical OWL [1] cardinality- or value- restriction. The type indicates the type of restriction (min, max, value) the value indicates the value that is attached to this restriction. Individuals can have *propertyAssertions* that indicate that an individual instantiates some property. In case of an object-property the target of a *propertyAssertion* is another individual. In case of datatype-properties it is some data-value. This abstract ontology model covers the important concepts of most ontology languages. By using this abstract model we can apply the work on different ontology formalisms that can be transformed to our representation. This does not require to transform the whole ontologies to our ontology model. It is sufficient to formulate the changes that occur to the ontology in terms of our ontology-model. Depending on the used ontology formalism reasoning may induce additional changes that we can simply also add to our change-log by comparing the materialized ontology version before and after the change.

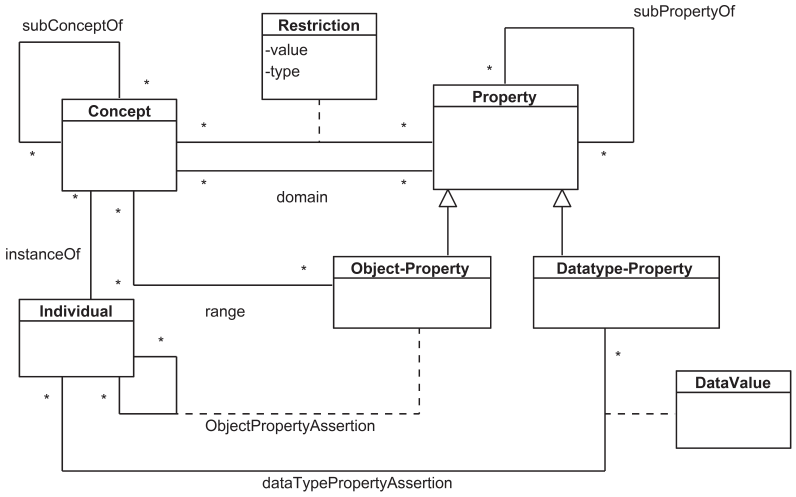

Fig. 2. Ontology Meta-Model

Based on this ontology model we define annotation paths that are used to annotate artifacts of an XML-Schema with a reference ontology. Both annotation path and dependency-definitions are modeled in the meta-model in figure 3. An *AnnotationPath* consists of a sequence of *AnnotationPathSteps*. Each step has a position and a *uri* that points to some property or concept of the reference ontology. According to the referenced ontology artifact an *AnnotationPathStep* is either a *conceptStep* or a *propertyStep* with the subclasses *objectPropertyStep* and *dataTypepropertyStep*. Each step except the last step has a succeeding step. Each step except the first step has a previous step. An annotation path has a defined first and a defined last-step.
A $DependencyDefinition$ has one *hasSubject* relation to an *AnnotationPath* and one or more *hasObject* relations to $dependencyDefinitionPath$. Each *dependencyDefinitionPath* consists of a number of *DependencyPathSteps*. A *DependencyPathStep* is a subclass of an *annotationPathStep* which is extended with an optional *setExpression*. The *setExpression* has a *type (subclasses, subproperties, instances)* and an optional $function$ which has a type that can be *value*, *min*, *max*, *avg*, *count*. Each *DependencyDefinitionPath* has a *hasAnnotationPath* relation to one *AnnotationPath*. This specific *annotationPath* is created by casting all steps to standard *AnnotationPathSteps*. An *annotationPath* can be represented in form of an ontology concept. This concept can be obtained with the method *getConcept*().

In order to meet the requirements that were introduced in the last section some integrity constraints on *AnnotationPath* and $DependencyDefinitionpath$ are required. We refer the interested reader for the complete definition of *annotation Path* to [14].

4.1 Integrity Constraints on *AnnotationPath*

1. The first step must be a *ConceptStep*.
2. An *AnnotationPath* must not contain *DependencyPathSteps*.
3. The last step must be a *ConceptStep* or a *dataTypePropertyStep*.
4. When a *conceptStep* has a previous step then the previous step must be an *ObjectPropertyStep*.
5. The next step of a *ConceptStep* must be an *ObjectPropertyStep* or a *DataTypePropertyStep*.
6. A *DataTypePropertyStep* can only exist as the last-step.
7. A *ConceptStep* must not reference to another *AnnotationPath*.

4.2 Integrity Constraints on *DependencyDefinitionPath*

1. All integrity constraints of standard steps except (2) and (7) also apply on $DependencyDefinitionPath$.
2. Only the last two steps may have a *setExpression* including a $function$.
3. The *setExpression* of type *subclasses* is only allowed for *conceptSteps*.

4. The *setExpression* of type *instances* is only allowed for *conceptSteps*.
5. The *setExpression* of type *subproperties* is only allowed for *propertySteps*.

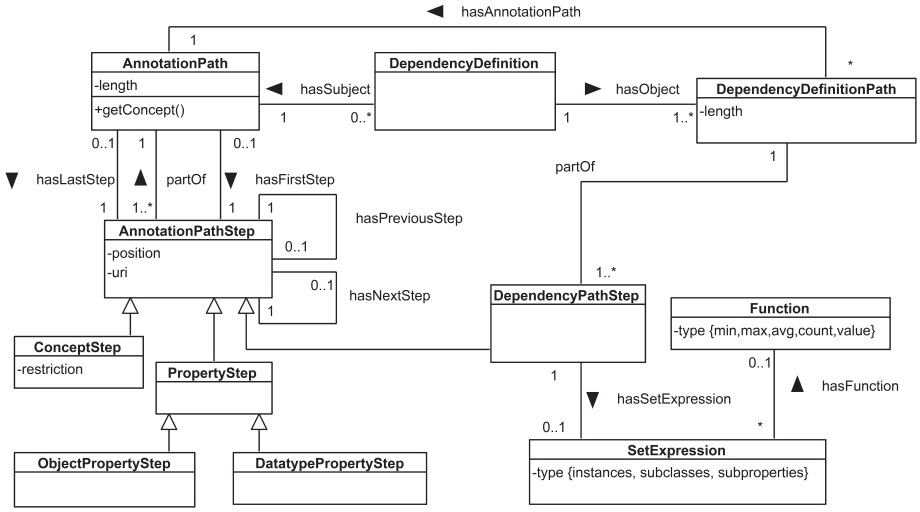

Fig. 3. Meta-Model of the Change-Dependency Definitions

5 Detection of Semantic Changes

In order to detect if the semantics of an annotation path got invalid by a change we first introduce the change-model for our ontology-model. The changes are stored in a change-log and the detection of relevant changes is realized by rules that operate on the statements of the change-log and the old and the new ontology versions.

5.1 Change-Log

The change-log consists of a set of changes C. Each change $c \in C$ is a tuple $c = (op, tid, p1, ...pn)$. Where op defines the kind of change, tid is a unique identifier of a change and also creates an order over the changes. A change with a lower tid was made before a change with a higher tid. The parameters $p1$ .. pn are parameters for the change. A multi-version ontology O has different versions $O_1...O_n$. Each version has a timestamp tid that uniquely identifies the version and also creates an order over the different versions. The set of changes that were made between two ontology versions O_n and O_{n+1} is denoted $Changes(O_n, O_{n+1}, C)$.

$$Changes(O_n, O_{n+1}, C) = \{\forall c \in C | c.tid \geq O_n.tid \wedge c.tid \leq O_{n+1}.tid\}$$

There are different change operations. We will discuss the different operations in the next section. For readability reasons we present the changes in form of predicates. Thus, *c.op* is the name of the predicate. In addition each change has an inverse change c^{-1} that compensates the change-effect of the change c. We assume that the changes in $Changes(O_n, O_{n+1}, C)$ are free of redundancies between O_n and O_{n+1} such that no change is compensated by an inverse change in the change-log.

Global Atomic Changes: Atomic changes are basic changes that add or remove concepts, properties or instances. It must be ensured by the ontology management system that a removal of a concept, property or instance is only possible when the artifact is not, still, referenced. The table shows the basic add-operations and their inverse delete operations.

c	c^{-1}
addConcept(tid,uri)	delConcept(tid,uri)
addOProp(tid,uri)	delOProp(tid,uri)
addDProp(tid,uri)	delDProp(tid,uri)
addInstance(tid,uri)	delInstance(tid,uri)

Hierarchy Changes: In addition to theses basic operations the following operations that maintain the hierarchy of the artifacts are required. They are used to express changes in the concept- or property-hierarchy.

c	c^{-1}
addChildC(tid,childUri,parentUri)	remChildC(tid,childUri,parentUri)
addChildOProp(tid,childUri,parentUri)	remChildOProp(tid,childUri,parentUri)
addChildDProp(tid,childUri,parentUri)	remChildDProp(tid,childUri,parentUri)
addInstToC(tid,instanceUri,conceptUri)	remInstToC(tid,instanceUri,conceptUri)

Update Changes: Update changes are used to modify the domain and range of properties as well as to maintain restrictions over properties on concepts and to modify property assertions on individuals. The rename operations change the URI of a specific concept, property or individual. We assume that these operations are global in the sense that every usage of the URI is changed automatically. In addition we assume that these operations are added to the change-log as atomic operations. Therefore, a rename does only show up as a rename but not as a delete and subsequent insert in the change-log. The inverse operations of update-changes are update operations of the same type but with swapped parameters.

- updateRestriction(tid, conceptUri, propertyUri, oldValue, oldType, newValue, newType)

– updateDomain(tid, propertyUri, {oldConceptUri}, {newConceptUri})
– updateRange(tid, propertyUri, {oldconceptUri}, {newconceptUri})
– updateType(tid, propertyUri, oldDataType, newDataType)
– updatePropertyAssertion(tid, instanceUri, propertyUri, oldValue, newValue)
– renameConcept(tid, oldUri, newUri)
– renameProperty(tid, oldUri, newUri)
– renameIndividual(tid, oldUri, newUri)

Composite Changes: In addition to the basic operations a set of composite operations is of interest. A merge operation merges a a set of concepts, properties or instances to one single concept, property or instance. The inverse operation of a merge is a split.

c	c^{-1}
mergeC(tid,{conceptUri},conceptUri)	splitC(tid,conceptUri,{conceptUri})
mergeP(tid,{propertyUri},propertyUri)	splitP(tid,propertyUri,{propertyUri})
mergeI(tid,{instanceUri},instanceUri)	splitI(tid,instanceUri,{instanceUri})

A composite operation is reflected as a sequence of other change-operations. In order to specify that an explicit change-operation is part of a composite change the atomic operation is annotated with the *tid* of the corresponding composite change with statements of the form:
$ChangeAnnotation(tidOfCompositeChange, tidOfAtomicChange)$

5.2 Implicit Changes

In addition to explicit changes there are implicit changes. These changes can directly be caused by explicit changes or by classification according to the source ontology language. For direct explicit changes we except the following implicit changes to be automatically included in the change-log by the ontology management system.

– If a concept is added or removed as a child of an existing concept and the existing concept or one of its parents has a restriction on a property then a restriction change is made on the added or removed concept implicitly.
– If a property is added or removed as a child of another property then the domain and range of the added or removed property is changed implicitly.
– If an individual is added or removed to/from a concept then it is added/removed to all its super-concepts implicitly.

The implicit changes as shown above and additional changes that can be derived by comparing the materialized ontology versions of the ontology before and after each change are stored in the change-log. The comparison algorithm can benefit from the fact that the change-log contains information about renames, additions and removes. Therefore, the complexity is strongly reduced since each element from the source and target ontology can directly be mapped. In order to trace

which change caused the addition of these implicit changes the implicit changes are annotated with predicates of the form $causedBy(impl\_tid, tid)$. This allows to keep the change-log clean of redundant implicit changes if the change that caused the implicit changes is compensated by an inverse operation.

5.3 Detection of Semantically Invalid Annotation Paths

Now we define semantic invalidation as a change affecting the semantics of an annotion path as follows.
The predicate $subClassOf(subc, superc, O_n)$ states that $subc$ is a subclass of the superclass $superc$ according to the ontology version O_n. As an equivalent class is logically defined as being sub- and superclass at the same time we assume that every class is a subclass of itself. The predicate $subPropertyOf(subp, superp, O_n)$ expresses the sub-property-relationship analogously.

Definition 1 ***Semantic Invalidation of an Annotation-Path:***

Given an ontology version O_n, a succeeding ontology version O_{n+1}, a set of changes $Changes(O_n, O_{n+1})$ abbreviated by C, a set of explicit dependency-definitions DEP, and a set of XML-Schema-annotations A. An annotation path $a \in A$ is semantically invalid if:

$$semInvalid(a, C, DEP, O_n, O_{n+1}) \leftarrow InvalidByDep \neq \{\}$$

$RelevantDependencies$ is the set of definitions where the corresponding subject is an equivalent- or superclass of the annotation path a:
$RelevantDependencies = \{\forall dep \in DEP | subClassOf(a, dep.subject, O_n)\}$
$InvalidByDep$ is the set of change-dependency definitions where one of the objects got invalid because it contains a step that is invalid or if the semantics of the annotation path of the object itself got changed.

$$InvalidByDep = \{\forall dep \in RelevantDependencies | (hasObject(dep, obj) \\ \wedge isInvalid(obj)) \vee (hasAnnotationPath(obj, annotationPathObject) \\ \wedge semInvalid(annotationPathObject, C, DEP, O_n, O_{n+1}))\}$$

Thus, dependency-definitions are transitive. If a depends on b and b depends on c then a is invalid when c is invalid.

A $DependencyDefinitionPath$ is invalid if at least one of its steps is invalid:
$isInvalid(obj) \leftarrow \exists step \in obj.steps \wedge InvalidStep(step)$
When a step is invalid is described in the next subsections.

Rules for the Invalidation of Steps: For the sake of simplicity we will define the invalidation of steps in form of rules omitting quantifiers. In addition all rules operate on the change-set defined by $Changes(O_n, O_{n+1}, C)$. Rules for the detection of additions operate on O_{n+1} while rules for the detection of removals

operate on O_{n-1}. The rules 1-3, 6, 8, 10 create possible invalidations, while the others create invalidations. Possible invalidations are invalidations where an invalidation may have taken place but additional review by the user is required.

1. A *PropertyStep* gets possibly invalid, if the domain of the property or of a super-property has changed.
   $PropertyStep(?step) \wedge subPropertyOf(?step.uri, ?superProperty,$
   $O_{n+1}) \wedge updateDomain(\_, ?superProperty, \_, \_)$
   $\Rightarrow InvalidStep(?step,' DomainOfPropertyChanged')$

2. A property-step is possibly invalid, if the range of the property or a super-property has changed.
   $ObjectTypePropertyStep(?step) \wedge subPropertyOf(?step.uri, ?superProperty,$
   $O_{n+1}) \wedge updateRange(\_, ?superProperty, \_, \_)$
   $\Rightarrow InvalidStep(?step,' RangeOfPropertyChanged')$
   The same holds for the change of the data type of a datatype-property analogously.

3. A concept-step gets possibly invalid, if a restriction on the property of the next step has changed.
   $ConceptStep(?step) \wedge isSubConceptOf(?step.uri, ?superuri, O_{n+1}) \wedge$
   $hasNextStep(?step, ?next) \wedge subPropertyOf(?next.uri, ?supernexturi, O_{n+1})$
   $\wedge updateRestriction(\_, ?superuri, ?supernexturi, \_, \_, \_, \_)$
   $\Rightarrow InvalidStep(?step,' RestrictionOnNexStepChanged')$

4. A set expression over subclasses without a function becomes invalid, if a subclass is added.
   $ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
   $!hasFunction(?exp, \_) \wedge equals(?exp.type,' subclasses') \wedge$
   $subConceptOf(?suburi, ?step.uri, O_{n+1}) \wedge addChildC(\_, ?newc, ?suburi)$
   $\Rightarrow Invalid(?step,' SubclassAdded')$

5. A set expression over subclasses without a function becomes invalid, if a subclass is removed.
   $ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
   $!hasFunction(?exp, \_) \wedge equals(?exp.type,' subclasses') \wedge$
   $subConceptOf(?suburi, ?step.uri, O_n) \wedge remChildC(\_, ?newc, ?suburi)$
   $\Rightarrow Invalid(?step,' SubclassRemoved')$

6. A set expression over subclasses without a function becomes possibly invalid, if a restriction on the property of the next step is changed in one of the subconcepts.
   $ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
   $!hasFunction(?exp, \_) \wedge equals(?exp.type,' subclasses')$
   $\wedge subConceptOf(?suburi, ?step.uri, O_{n+1}) \wedge hasNextStep(?step, ?next)$
   $\wedge subPropertyOf(?nextpropuri, ?next.uri, O_{n+1}) \wedge$

$updateRestriction(\_, ?suburi, ?nextpropuri, \_, \_, \_, \_)$
$\Rightarrow Invalid(?step.'RestrictionOnSubclassChanged')$

7. A set expression over instances without a function becomes invalid, if instances are added or removed to/from the specified concept or one of its subconcepts. We will only depict the rule for the addition here.
$ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
$!hasFunction(?exp, \_) \wedge equals(?exp.type,' instances') \wedge$
$subConceptOf(?conceptUri, ?step.uri, O_{n+1})$
$\wedge addInstToC(\_, \_, ?conceptUri) \Rightarrow Invalid(?step,' InstancedAdded')$

8. A set expression over instances becomes possibly invalid, if the succeeding-step is a property-step and property assertions on instances for that property are modified.
$ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
$!hasFunction(?exp, \_) \wedge equals(?exp.type,' instances') \wedge$
$subConceptOf(?conceptUri, ?step.uri, O_{n+1}) \wedge hasNextStep(?step, ?next) \wedge$
$PropertyStep(?next) \wedge instanceOf(?insturi, ?conceptUri) \wedge$
$updatePropertyAssertion(\_, ?insturi, ?next.uri, \_, \_)$
$\Rightarrow Invalid(?step,' PropertyAssertionChanged')$

9. A set expression over sub-properties becomes invalid, if a sub-property is added or removed. We will only depict the rule for the addition here.
$PropertyStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
$!hasFunction(?exp, \_) \wedge equals(?exp.type,' subproperties') \wedge$
$subPropertyOf(?suburi, ?step.uri, O_{n+1})$
$\wedge(addChildOProp(\_, ?newc, ?suburi) \vee (addChildDProp(\_, ?newc, ?suburi))$
$\Rightarrow Invalid(?step,' SubpropertyAdded')$

10. A set expression over sub-properties becomes possibly invalid, if the domain or range of a sub-property is changed. *uDomainOrRange* is the superclass of *updateDomain* and *updateRange*
$PropertyStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
$!hasFunction(?exp, \_) \wedge equals(?exp.type,' subproperties')$
$\wedge subPropertyOf(?suburi, ?step.uri, O_{n+1}) \wedge$
$uDomainOrRange(\_, ?suburi, \_, \_)$
$\Rightarrow Invalid(?step,' DomainOrRangeOfSubpropertyChanged')$

Functions on Set Expressions: The rules in the last section excluded the existence of functions over *setExpressions*. Therefore, any change that has consequences for the *setExpression* is considered to invalidate the step. When a function is given then the problematic change-operations depend on the used function and, therefore, additional rules are required. The sum-function is vulnerable to *add* and *delete* operations but is resistent to local *merge* or *split*

operations. All other aggregation functions are vulnerable to *add*, *del*, *split*, and *merge*. The value function is vulnerable to renames of sub-concepts or sub-properties as well as to *delete*, *split* and *merge* operations. Therefore, specific rules for the different kinds of functions are required. Since *merge* and *split* are complex change-operations the rules need to operate on the annotation of the changes ($ChangeAnnotation(...)$). Due to space limitations we will only provide rules for sum-functions with added sub-concepts and value-functions with renames.

1. A concept-step with a sum-function over sub-concepts gets invalid, if sub-concepts are added or removed and the add and remove operations are not linked to local split or merge operations. A non-local split operation happens when the source concept was a sub-concept of the step and one of the new concepts is not, still, a sub-concept of the step according to the current ontology version. The following rule represents the case of the addition of sub-concepts.

   $ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
   $hasFunction(?exp, ?fu) \wedge equals(?fu.type, 'sum') \wedge$
   $subConceptOf(?suburi, ?step.uri, O_{n+1}) \wedge addChildC(?tid, ?newc, ?suburi)$
   $\wedge !(splitC(?stid, ?source, ?suburi) \wedge ChangeAnnotation(?stid, ?tid)$
   $\wedge subConceptOf(?source, ?step.uri, O_n) \wedge$
   $!(splitC(?stid, ?source, ?otherSplitUri) \wedge notequals(?otherSplitUri, ?newc)$
   $\wedge addChildC(?tid, ?otherSplitUri, ?otherAddUri)$
   $subClassOf(?otherAddUri, ?step.uri, O_{n+1})))$
   $\Rightarrow Invalid(?step, 'SubclassAdded')$

2. A concept-step with a value-function gets invalid if one of the sub-concepts is renamed or deleted. We will show the rule for the renames here.

   $ConceptStep(?step) \wedge hasSetExpression(?step, ?exp) \wedge$
   $hasFunction(?exp, ?fu) \wedge equals(?fu.type, 'value') \wedge$
   $subConceptOf(?oldUri, ?step.uri, O_{n+1}) \wedge$
   $renameConcept(\_, ?oldUri, ?newUri)$
   $\Rightarrow Invalid(?step, 'Value - Changed')$

6 Proof of Concept Implementation

We have implemented this approach for computing semantic invalidations of annotation paths using the Jena API[3] and the pellet[4] reasoner. The input for the algorithm consists of a set of annotation paths, a set of dependency-definition paths, and information about renames, splits and merges. The output is a subset of the input annotation path where the semantics has (possibly) changed.

[3] http://jena.sourceforge.net/
[4] http://clarkparsia.com/pellet/

Additionally, explanations for the semantic invalidations are provided. The system transforms the materialized source and target ontology to instances of our ontology-meta-model as shown in figure 2. In a next step SPARQL[5] queries are used to generate the change-log. Each log-entry is an individual of a change-ontology. The change ontology is a representation of the change-hierarchy defined in section 5.1.
The rules as proposed in section 5.3 are implemented in form of SPARQL queries that operate on instances of the change-log and the instances of the meta-ontology. The required negation is implemented in form of SPARQL filters.
The special property $subClassOf(c1, c2, ?v)$ (and $subpropertyOf(p1, p2, ?v)$ analogously) is realized in form of two distinct properties $subClassOfOld(c1, c2)$ and $subClassofNew(c1, c2)$ that are added to the instances of the source and target ontology version of the ontology meta-model. Most invalidation rules can directly be represented in SPARQL, while some more complex queries need additional post-processing. The prototype demonstrated the feasibility of our approach.

7 Related Work

In [10] the consistent evolution of OWL ontologies is addressed. The authors describe structural, logical and user-defined consistency requirements. While the structural and logical requirements are directly defined by the used ontology formalism the user-defined requirements describe additional requirements from the application domain that cannot be expressed with the underlying ontology language. The authors do not make any suggestion on how these requirements should be expressed. Therefore, our approach can be seen as one specific form of user-defined-consistency requirements. The main difference is that in our case the artifact that becomes inconsistent if a user-defined consistency-definition is violated is not the ontology itself but instance-data in XML-documents. In [4] functional dependencies over Aboxes (individuals) are addressed. The dependencies are formulated in the form antecedent, consequent and an optional deterministic function. The antecedent and consequent are formulated via path expressions which can be compared to our approach. The dependencies are directly transformed to SWRL-rules. Therefore, the functional dependencies directly operate on the individuals (Abox) and additional knowledge can be added to the Abox. In addition, data that does not comply with the rules can be marked as inconsistent. In contrast to our approach, the approach is limited to the instance layer which makes it unusable for our scenario where instance-data from XML-documents is never added to the Abox. Therefore, knowledge about changes needs to be evaluated in order to predict semantic-changes of the semantics of instance-data.
In [18] the validity of data-instances after ontology evolution is evaluated. An algorithm is proposed that takes a number of explicit changes as input and calculates the implicit changes that are induced by the explicit changes. These explicit and implicit changes can then be used to track the validity of data-instances.

[5] http://www.w3.org/TR/rdf-sparql-query/

The general idea of the approach is that if an artifact gets more restricted existing instances are invalidated. Since the approach only takes into account implicit changes that can be computed based on explicit changes it does not support the explicit definition of change-dependencies.
Our work heavily depends on the existence of an expressive change-log between two versions of an ontology. The automatic detection of changes that happened between two versions of an ontology are covered with approaches like [15], [9] or [11]. If a change-log exists this can also be used as a basis to generate more expressive changes as required by our approach. Approaches that operate on a change-log in order to generate additional changes are [12], [17]. Methods to efficiently store and manage different ontology versions are presented in [8],[5].
While there is only limited work on dependencies in the field of ontologies it is traditionally broadly studied in the database community. Recent and related research in this field is for example [3] and [6]. In [3] a model and system is presented that keeps track of the provenance of data that is copied from different (possibly curated) databases to some curated database. Changes in the source databases may influence the data in the target databases. Therefore, provenance information is required to track those changes. In [6] the problem of provenance in databases is formalized with an approach that is inspired by dependency analysis techniques known from program analysis or slicing techniques. In contrast to our approach both provenance approaches cope with changes of instance data and do not address changes of schema/meta-data.

8 Conclusion

In this paper we have addressed the problem of semantic changes of annotations that occur due to the evolution of their reference ontologies. As ontologies have to keep up with changes in the modeled domain (the real world) such changes occur frequently. Unrecognized semantic changes (may) lead to incorrect results for document transformations, semantic queries, statistics etc. based on semantic annotations. So there is an urgent necessity to maintain semantic annotations when a reference ontology changes. As experience shows, high maintenance costs are a severe obstacle against wide adoption of techniques. We presented a technique to identify those annotations which have to be considered for maintenance due to changes in the reference ontology. This should ease the burden of maintenance considerably.

References

1. OWL web ontology language reference. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/owl-ref/.
2. RDF vocabulary description language 1.0: RDF Schema. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/rdf-schema/.
3. Peter Buneman, Adriane P. Chapman, and James Cheney. Provenance management in curated databases. In *Proc. of SIGMOD'06*, pages 539–550. ACM, 2006.

4. Jean-Paul Calbimonte, Fabio Porto, and C. Maria Keet. Functional dependencies in owl abox. In Angelo Brayner, editor, *Proc. of SBBD'09*, pages 16–30. SBC, 2009.
5. Chuming Chen and Manton M. Matthews. A new approach to managing the evolution of owl ontologies. In Hamid R. Arabnia and Andy Marsh, editors, *Proc. of SWWS'08*, pages 57–63. CSREA Press, 2008.
6. James Cheney, Amal Ahmed, and Umut Acar. Provenance as dependency analysis. In Marcelo Arenas and Michael Schwartzbach, editors, *Database Programming Languages*, volume 4797 of *LNCS*, pages 138–152. Springer, 2007.
7. Mathieu d'Aquin, Anne Schlicht, Heiner Stuckenschmidt, and Marta Sabou. Ontology modularization for knowledge selection: Experiments and evaluations. In Roland Wagner, Norman Revell, and Günther Pernul, editors, *Proc. of DEXA'07*, volume 4653 of *LNCS*, pages 874–883. Springer, 2007.
8. Johann Eder and Christian Koncilia. Modelling changes in ontologies. In Robert Meersman, Zahir Tari, and Angelo Corsaro, editors, *Proc. of OTM'04*, volume 3292 of *LNCS*, pages 662–673. Springer, 2004.
9. Johann Eder and Karl Wiggisser. Change detection in ontologies using dag comparison. In John Krogstie, Andreas Opdahl, and Guttorm Sindre, editors, *Proc. of CAiSE'07*, volume 4495 of *LNCS*, pages 21–35. Springer, 2007.
10. Peter Haase and Ljiljana Stojanovic. Consistent evolution of owl ontologies. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *The Semantic Web: Research and Applications*, volume 3532 of *LNCS*, pages 91–133. Springer, 2005.
11. Michael Hartung, Anika Gross, and Erhard Rahm. Rule-based generation of diff evolution mappings between ontology versions. *CoRR*, abs/1010.0122, 2010.
12. Asad Masood Khattak, Khalid Latif, Manhyung Han, Sungyoung Lee, Young-Koo Lee, and Hyoung-Il Kim. Change tracer: Tracking changes in web ontologies. In *Proc. of ICTAI'09*, pages 449–456. IEEE Computer Society, 2009.
13. Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *IEEE I.C.*, 6:60–67, 2007.
14. Julius Köpke and Johann Eder. Semantic annotation of xml-schema for document transformations. In Robert Meersman, Tharam Dillon, and Pilar Herrero, editors, *Proc. of OTM'10 Workshops*, volume 6428 of *LNCS*, pages 219–228. Springer, 2010.
15. Natalya F. Noy and Mark A. Musen. Promptdiff: a fixed-point algorithm for comparing ontology versions. In *Proc. of AAAI'02*, pages 744–750. AAAI, 2002.
16. Natalya Fridman Noy and Mark A. Musen. Specifying ontology views by traversal. In Sheila McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Proc. of ISWC'04*, volume 3298 of *LNCS*, pages 713–725, 2004.
17. Peter Plessers and Olga De Troyer. Ontology change detection using a version log. In Yolanda Gil, Enrico Motta, V. Benjamins, and Mark Musen, editors, *Proc. of ISWC'05*, volume 3729 of *LNCS*, pages 578–592, 2005.
18. Li Qin and Vijayalakshmi Atluri. Evaluating the validity of data instances against ontology evolution over the semantic web. *Inf. Softw. Technol.*, 51:83–97, January 2009.
19. Ljiljana Stojanovic, Alexander Maedche, Boris Motik, and Nenad Stojanovic. User-driven ontology evolution management. In Asunción Gómez-Pérez and V. Benjamins, editors, *Proc. of EKAW'02*, volume 2473 of *LNCS*, pages 285–300. Springer, 2002.
20. Victoria Uren, Philipp Cimiano, José Iria, and et. Al. Semantic annotation for knowledge management: Requirements and a survey of the state of the art. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(1):14 – 28, 2006.