Logical Invalidations of Semantic Annotations

Julius Köpke and Johann Eder

Author's Version of

Julius Köpke and Johann Eder “Logical Invalidations of Semantic Annotations”.
In Advanced Information Systems Engineering
Lecture Notes in Computer Science Volume 7328, 2012, pp 144-159

The final publication is available at link.springer.com:
http://link.springer.com/chapter/10.1007%2F978-3-642-31095-9_10

Logical Invalidations of Semantic Annotations

Julius Köpke, Johann Eder

Department of Informatics-Systems, Alpen-Adria Unversität Klagenfurt, Austria
firstname.lastname@aau.at http://www.aau.at/isys

Abstract. Semantic annotations describe the semantics of artifacts like documents, web-pages, schemas, or web-services with concepts of a reference ontology. Application interoperability, semantic query processing, semantic web services, etc. rely on a such a description of the semantics. Semantic annotations need to be created and maintained. We present a technique to detect logical errors in semantic annotations and provide information for their repair. In semantically rich ontology formalisms such as OWL-DL the identification of the cause of logical errors can be a complex task. We analyze how the underlying annotation method influences the types of invalidations and propose efficient algorithms to detect, localize and explain different types of logical invalidations in annotations.

Keywords: Semantic annotation, ontology evolution, annotation maintenance, logical invalidation

1 Introduction

Semantic annotations are used to attach semantics to various kinds of artifacts like documents, web-pages, schemas, or web-services. Semantic annotations are used in information systems engineering in various ways: To enable semantic interoperability of information systems, to generate transformations of documents between heterogenous information systems, to support correct schema integration, selection and composition of web services, etc. - see e.g. [10, 15, 16]. In this paper we focus on annotations on the schema level rather than on the instance level (e.g. RDF [9]). Schema level annotations are better suited for high volume data because all documents that are instances of an annotated schema can be interpreted with the reference ontology. XML-Schemas can be annotated according to the W3C Recommendation Semantic Annotations for WSDL and XML Schema (SAWSDL) [6] which provides two different methods: declarative annotations with model references that attach concepts of a reference ontology to elements or types of a schema, as well as references to lifting and lowering mappings (scripts) that actually transform XML-instance data to individuals of the reference ontology. In [7] we proposed an annotation technique which is fully declarative but allows to express semantics more precisely than mere concept references.

The aim of this work is to support the process of annotating schemas and of maintaining annotations that got invalid after ontology evolution [3]. We do not

assume that invalid annotations can be corrected automatically as this typically requires real world domain knowledge. Annotations need to be validated during the creation or maintenance process on three levels:

1. Structure: The referenced concepts or properties have to exist in the reference ontology and satisfy basic structural constraints [7].
2. Logics: The representation of a structurally valid annotation as a concept must not contradict with the reference ontology.
3. Semantics: The annotations correctly describe the real-world domain. We define semantic invalidations as changes of the ontology that have consequences on the interpretation of instance data of annotated schemas. This is different from logical invalidation. We have presented an approach for the detection of semantic invalidations that is based on the explicit definition of change-dependencies in [8].

The contribution of this paper is an in depth analysis of logical invalidations, resulting in algorithms and methods to (a) discover whether an annotation path expressions is logically invalid, (b) which part of an annotation path expression is invalid, and (c) the cause of this invalidation. This information should enable annotators to correct the annotation.
In the following we discuss semantic annotations of XML-schemas. However, the annotation method and the algorithms for validation of the annotations are not restricted to XML Schemas, but can as well be used for all other types of artifacts like web services, relational schemas, etc.

2 Annotation Method

We will briefly introduce the declarative annotation method we proposed in [7]. The goal of the annotation method is to describe the annotated element in much more detail than the direct annotation with existing concepts of the reference ontology while being declarative in contrast to rather procedural lifting/lowering mappings. The proposed annotation method has two representations: Path expressions over concepts and properties of an ontology that are directly used to annotate XML-Schema elements or types in form of SAWSDL [6] model references. These annotation paths are translated to complex OWL formulas representing the annotation in the ontology. We first define the structure of an annotation path expression and then show how a path is translated to an OWL formula. For details we refer to [7].

Definition 1. *Annotation Path:* An annotation path p is a sequence of steps. Each step is a triple s=(uri, $type$, res), where $s.uri$ is some URI of an element of the reference ontology O; the type $s.type$ defines the type of ontology element that is addressed by $s.uri$. It can be cs for a concept-step, op for an object-property step or dp for a datatype-property step.
Only concept-steps may have a set of restrictions $s.res$. Each restriction $r \in s.res$ can either be an individual or a restricting path expression. Such a path

expression adds a restriction to the corresponding step s. If $s.res$ contains multiple restrictions they all apply to the corresponding step s (logical and). Each annotation path $p \in P$ has a type $\in \{$*ConceptAnnotation, DataTypePropertyAnnotation* $\}$. The type is defined by the last step.

A structurally valid annotation path expression must comply with the following restrictions:

- All steps refer to existing URIs of the ontology.
- The first step must refer to a concept.
- The last step must refer to a concept or to a datatype property.
- A step that refers to an object property can only occur between two concept steps.
- A step that refers to a concept must be followed by an object- or datatype-property step or nothing.
- A step that refers to a datatype property can only exist as the last step.
- Only steps that refer to concepts may have additional restrictions.

Structurally valid annotation path expressions can automatically be transformed to OWL [11] concepts that extend the reference ontology. The following example shows an annotation path and its corresponding concept in the reference ontology.
The annotation path $p =$ *Order/billTo/Buyer[Mr_Smith]/hasCountry/Country* is used to annotate a *country* element for some XML-Schema for *order* documents. It describes a subconcept of a *country* with an inverse relation *hasCountry* to some *Buyer* that has an inverse *billTo* relation to some *Order*. The buyer has a restriction to state that Buyer is a specific buyer with the name/URI *Mr. Smith*. The corresponding class definition *p.c* is shown in listing 1.1.

```
1 Class: Order/billTo/Buyer[Mr_Smith]/hasCountry/Country
2 EquivalentClasses(
3       ConceptAnnotation and Country and inv
4         (hasCountry) some
5            (Buyer and {Mr_Smith} and inv (billTo) some (Order)
6         ))
```

Listing 1.1. Representation of an annotation path in OWL

Structurally valid annotation path expression can be transformed to OWL concepts with the following mappings:

- *Concept-steps* are directly mapped to OWL-concepts.
- Restrictions on *concept-steps* are mapped to enumerated classes or restrictions over the corresponding concept.
- *Object-property-steps* are mapped to inverse OWL *some values from* restrictions between concepts on that specific property.
- *Datatype-property-steps* are mapped to OWL some values from restrictions of the last concept step on that specific datatype-property.

The annotation method allows different types of annotations, that we now define in order to describe the possible invalidations for each type in the following sections. *Simple concept annotations* consists of only one concept. *Simple datatype annotations* consists of only one concept and one datatype property. *3-step concept annotations* consists of a concept, an object-property and another concept. *General annotations* consist of more than 3 steps.

3 Logical Invalidation of Annotation Paths

An annotation path is logically invalid, if the corresponding annotation concept is not satisfiable in the reference ontology. Thus, the detection of logically invalid annotations is a classical reasoning task. The root concept of OWL is *Thing*. Any subconcept of this concept is satisfiable in the ontology. A satisfiable concept can contain individuals. Concepts that are not subclasses of *Thing* are not satisfiable and are subclasses of the concept $Nothing$, which is the complement of $Thing$.

Definition 2. *Logical Invalidation of an Annotation:* A structurally valid annotation path p is logically invalid if the corresponding annotation concept $p.c$ is unsatisfiable in the reference Ontology O. $O \bigcup p.c \rightarrow p.c \not\sqsubseteq$ Thing

Why can't we use standard tools for validation and repair such as [12] or [14]? First, we want to determine which steps of the annotation path are responsible for the invalidation rather than determining a set of axioms of the (extended) ontology causing the invalidation. Second, repairs can only change annotation paths, and not axioms of the ontology. For debugging annotations we have the following requirements:

- The ontology is assumed to be consistent and therefore, free of contradictions before the annotation concept is added.
- The structure of the annotation concepts is strictly defined by the annotation method.
- Repairs can change annotation path expression but not the ontology.
- In case of annotation maintenance we require that the annotation concept was valid in the previous ontology version.

Therefore, we need to find the error in the steps of the annotations rather than in their OWL representation. This limits the usefulness of standard OWL debugging methods (see section 6). If an ontology evolves, annotation maintenance means to identify those annotation paths which became logically invalid due to the changes in the ontology and to identify those steps in the annotation path which cause the invalidation. An expert then can repair the invalid annotation paths efficiently using the information about the cause of the invalidation.

In OWL logical contradictions boil down to a limited set of contradictions [12]: **Atomic** - An individual belongs to a class and its complement. **Cardinality** - An individual has a max cardinality restriction but is related to more

distinct individuals. **Datatype** - A literal value violates the (global or local) range restriction of a datatype property.
These clashes also apply for unsatisfiable classes. Thus, for example a class is unsatisfiable if it is defined as an intersection with its complement or if it has contradicting cardinality- or datatype-restrictions. Of course such invalidations can be produced by non-local effects. In the next sections we discuss how the different annotation types can be logically invalid.

3.1 Invalidation of Simple Concept Annotations

A simple concept annotation consists of only one concept. Thus, a concept with the name $prefix + conceptUri$ is generated, where prefix is some unique identifier that is not used in the ontology O, with the equivalent class definition ($ConceptAnnotation$ *and* $conceptUri$).

Theorem 1. A simple concept annotation that is structurally valid is also logically valid.

Proof. We require that all concepts of the reference ontology are satisfiable. Thus, there is only one case, where the union of $ConceptAnnotation$ and $conceptURI$ can result in an unsatisfiable concept: The class with the URI $ConceptAnnotation$ is disjoint from the concept with the URI $conceptURI$. This is impossible because the primitive concept $ConceptAnnotation$ does not exist in the reference ontology before the annotations are added. Thus, there cannot be an axiom in the ontology that contradicts with it.

3.2 Invalidation of Simple Datatype Annotations:

A simple datatype annotation of the form $/c/datatypeProperty$ consists of a concept and a restriction over some datatype-property of the form: *(datatypeAnnotation and c and datatypeProperty some rdf:Literal).*

Theorem 2. There exists no invalid simple datatype annotation that does not violate one of the following conditions:

1. ***Invalid-domain:*** The intersection of the domain of the property with the concept is not a subclass of $OWL : Thing$.
   $c \sqcap domain(datatypeProperty) \not\sqsubseteq Thing$
2. ***Invalid-restriction:*** The intersection of the concept and the restriction over the datatype-property is not a subclass of $OWL : Thing$.
   c *and* $datatypeProperty$ *some* $Literal \not\sqsubseteq Thing$

Proof. Obviously, case 2 of an invalidation is equivalent to the satisfiability-check of the whole annotation concept. There is only one additional case for an invalidation where the concept with the URI $datatypeAnnotation$ is disjoint from c, which is impossible in analogy to theorem 1. Thus, every logically invalid simple datatype annotation is captured.

According to theorem 2 every simple datatype annotation that is invalid due to an *invalid-domain* is also invalid due to an *invalid-restriction*. Thus, in order to detect the cause of the error in more detail we need to investigate the reasons for the invalid restriction. This can be realized by additionally checking the first case. In addition the restriction clash is not yet atomic. In OWL there are the following scenarios for invalid restrictions over datatype-properties:

1. The datatype of the restriction does not comply with a datatype that is required by an existing restriction in O.
2. There is a cardinality clash between the existential restriction of the annotation path and an existing restriction in O.

Theorem 3. An invalidation of a simple datatype annotation due to a conflicting datatype restriction is impossible.

Proof. A contradicting datatype must be disjoint from the datatype in the existential restriction. This is impossible because every datatype is a subtype of *rdfs:literal*, which is used for the existential restriction in the annotation concept. No subtype can be disjoint from its supertype.

Cardinality clashes are possible, when there is a restriction on the class ($c \sqcap datatypeProperty$) of the form: $datatypeProperty\ max\ n\ type$, where *type* is $rdfs : Literal$ or any subtype of it.

3.3 Invalidation of 3-Step Concept Annotations

A 3-step concept annotation is a triple of the form *concept/property/otherconcept*. It is represented as an OWL equivalent class expression *otherconcept and inv (property) some concept*. Such an expression can be invalid due to *domain-invalidation*, *range-invalidation* and *restriction-invalidation*.

Definition 3. *Domain-Invalidation:*
An annotation triple of the form $concept/Property/otherconcept$ is unsatisfiable due to a *domain-invalidation*, iff: $domain(Property) \sqcap concept \not\sqsubseteq Thing$

Definition 4. *Range-Invalidation*
An annotation triple of the form $concept/Property/otherconcept$ is unsatisfiable due to a *range-invalidation*, iff: $range(Property) \sqcap otherconcept \not\sqsubseteq Thing$

Definition 5. *Restriction-Invalidation:*
An annotation triple of the form $concept/Property/otherconcept$ is unsatisfiable due to a *restriction-invalidation*, iff: $otherconcept \sqcap inv\ (Property)\ some\ concept \not\sqsubseteq Thing$

Theorem 4. There exists no invalid *3-step concept annotation* that does not introduce a *domain-invalidation*, *range-invalidation* or *restriction-invalidation*.

Proof. A *restriction-invalidation* is defined as $otherconcept \sqcap inv\ (hasProperty)\ some\ concept \not\sqsubseteq Thing$. This is equivalent to the satisfiability requirement for the whole annotation path because the intersection of *otherconcept* and $ConceptAnnotation$ cannot result in a clash (see proof of theorem 1). Thus, there exists no invalid 3-step annotations that are not captured by the enumerated invalidations.

While the domain or range invalidations are already atomic there can be different causes for invalid restrictions: A restriction can be invalid because the range of the restriction is disjoint from another $allvaluesFrom$ restriction on *concept* or it can be invalid because there is a cardinality restriction on *concept* of the form $property\ max\ n\ otherconcept$. Therefore, the *invalid-restriction* problem can be divided into *invalid-value-restriction* and *invalid-cardinality-restriction*.

OWL2 allows the definition of object properties to be functional, inverse functional, transitive, symmetric, asymmetric, reflexive, and irreflexive. Since the existence of the object property is defined by the existential quantification of the inverse of the property these characteristics can influence the satisfiability of the annotation. For example, given an annotation path $p = /A/hasB/B$, the path is invalid, if $hasB$ is defined as inverse functional and B has an inverse $hasB$ restriction in O to some other class that is disjoint from A.
We can summarize that a *3-step concept annotation* can be invalid because of the restrictions that are formulated over the corresponding annotation concept. Definition 5 is sufficient but the root cause can be found in property characteristics or cardinality or value clashes.

4 Invalidation of General Annotations

In the last section we defined all local invalidations that can occur in annotations that consists of 3 steps. A general annotation consists of a sequence of 3-step concept annotations called triples. The last step can be a 3-step concept annotation or a simple datatype annotation. We will first show that all local invalidation types also apply to general concept annotations and then discuss additional kinds of invalidations that are only possible in general annotations.

4.1 Invalidation of General Annotation due to Local Invalidations

Definition 6. *Local-Invalidations:* The invalidation types *domain-invalidation* (see def. 3), *range-invalidation* (see def. 4) and *restriction-invalidation* (see def. 5) are local invalidations, that are defined in the context of a triple.

Theorem 5. A locally invalid 3-step annotation cannot get valid, when it occurs as a triple in a general annotation path.

Proof. A general annotation path has the form: $/c_1/p_2/c_3/.../c_{n-2}/p_{n_1}/c_n/$. We now assume that there exits a triple $C_{inv} = c_x/p_y/c_z$, in the path that is invalid,

when it is inspected separately (local invalidation), but the entire annotation concept ... $c_{-2}/p_{-1}/c_x/p_y/c_z/p_1/c_2$ ... is valid. This implies that either c_x or c_z were implicitly changed to classes that are not still causing local invalidations in C_{inv}. When the triple C_{inv} is added to the annotation concept this is realized by an expression of the form:
... c_2 *and* (*inv*) p_1 *some* (c_z *and inv* (p_y) *some* (c_x *and* p_{-1} *some* ...
Thus, z_x is implicitly replaced with an intersection of z_x and (p_1 *some* ...) that we now call z_{x2}. c_z gets implicitly replaced with c_z and (*range* (p_1) that we now call c_{z2}. In order to achieve a satisfiable triple C_{inv} in p, c_{x2} must not be a subclass of c_x or c_{z2} must not be a subclass of c_z. This is a contradiction because they are logically subclasses of c_x and c_z.

Theorem 6. A general concept annotation that contains an invalid triple is itself logically invalid.

Proof. A general concept annotation path consists of triples: $t_1/t_2/t_3/.../t_n$. We will now show via induction that as soon as one of its triples is unsatisfiable, the whole annotation concept is unsatisfiable. Beginning with an annotation p_1 that only consists of t_n. If t_n is itself unsatisfiable, then the whole path cannot be satisfiable because it is represented as a subclass of t_n in $p_1.c$. We now assume that p_1 is satisfiable and we add t_{n-1}, which is supposed to be unsatisfiable. The addition renders the whole annotation path unsatisfiable because the connection between p_n and t_{n-1} is represented in form of an existential restriction. This step can be repeated by adding an unsatisfiable triple to a longer and longer valid path, until t_1 is reached. Therefore, if any triple of a general concept annotation is locally invalid the whole annotation concept must be logically invalid.

As a conclusion all previously discussed local invalidations also apply to general annotations. Additionally there are invalidations that only occur in general annotations: direct-triple-disjointness and arbitrary non local invalidations.

4.2 Direct-Triple-Disjointness

One kind of invalidation that does not exist for 3-step annotations can be caused by the concatenation of two annotation triples. This means the concept that is implicitly created by the first triple is disjoint from the concept which is required by the second triple. An example for such a scenario is shown in Figure 1. The corresponding reference ontology is shown in listing 1.2.

```
1 Order isA Document
2 Invoice isA Document
3 Disjoint(Order, Invoice)
4 Domain(SendsOrder) = Customer
5 Range(SendsOrder) = Order
6 Domain(hasInvoiceNumber) = Invoice
7 Range(hasInvoiceNumber) = InvoiceNumber
```

Listing 1.2. Example ontology for direct-triple-disjointness invalidations

Fig. 1. Example of direct-triple-disjointness

In *Customer/sendsOrder/Document/hasInvoiceNumber/InvoiceNumber* each triple is valid individually, but the combination of the triples leads to an unsatisfiable concept. The reason for this invalidation is that the subclass of *Document* that is produced by the range of *sendsOrder* in the first triple is disjoint from the subclass of *Document* that is produced by the domain of $hasInvoiceNumber$ in the second triple.

Theorem 7. *Direct-Triple-Disjointness:* An annotation path $p = /t_1/.../t_n/$ is invalid, if there exist two logically valid neighbored triples $t_n = c_n/p_n/c_m$ and $t_m = c_m/p_m/c_{m+1}$, where $range(p_n) \sqcap restriction(c_n, p_n) \sqcap domain(p_m) \sqcap c_m \not\sqsubseteq Thing$.

Proof. The intersection class $range(p_n) \sqcap restriction(c_n, p_n) \sqcap domain(p_m) \sqcap c_m$ describes the implicit concept between two annotation triples, that is responsible for the concatenation of the triples. If this intersection concept is unsatisfiable any class with an existential restriction for this concept becomes unsatisfiable.

4.3 Non-Local Invalidations

Local- and direct-triple-disjointness invalidations can be located precisely. That means the step in the path that causes the invalidation can be annotated with the type of the clash and the reason for the invalidation. This can be valuable information for a user who has to repair the annotation path.
In case of general annotation paths which consist of two or more triples additional invalidations can occur which are not necessarily induced by neighboring triples. We will now first present an example in listing 1.3 and then define the problem in general.
The example contains the annotation concept *MyAnnotation* that represents the path *BusinessCustomer/sends/Order/has/Itemlist/contains/PrivateProduct /hasPrice/Price*. The annotation concept is free of local- or direct-triple- disjointness invalidations. Nevertheless, it is logically invalid because according to the ontology, business customers may only send business orders and business orders may only contain business products. Business products are disjoint from private products. This renders the whole annotation path unsatisfiable. Now our goal is to find the steps in the path that are responsible for the invalidation. In this case the steps that are responsible for the clash are:
BusinessCustomer/sends/Order/has/Itemlist/contains/PrivateProduct.

```
1  Class(BusinessCustomer)
2  Class(Order), Class(BusinessOrder), Class(PrivateOrder)
3  Class(Itemlist)
4  Class(PrivateProduct), Class(BusinessProduct)
5  Class(Price)
6  Class(ClassMyAnnotation)
7  ObjectProperty(sends)
8  ObjectProperty(contains)
9  ObjectProperty(has)
10 ObjectProperty(hasPrice)
11 EquivalentClass(BusinessCustomer,
12     BusinessCustomer and sends only BusinessOrder)
13 EquivalentClass(BusinessOrder, BusinessOrder and has only
14     (Itemlist and contains only BusinessProduct))
15 EquivalentClass(MyAnnotation, Price and inv (hasPrice) some
16     (PrivateProduct and inv (contains) some
17         (Itemlist and inv (has) some
18             (Order and inv
19                 (sends) some BusinessCustomer))))
20 disjoint(BusinessOrder,PrivateOrder)
21 disjoint(BusinessProduct,PrivateProduct)
```

Listing 1.3. A non local invalidation

To define such invalidations we first introduce a normalized representation form of an annotation concept.

Definition 7. *Normalized Annotation Concept*
An annotation path $p = /c_1/p_2/c_3/...c_{n-2}/p_{n-1}/c_n/$ is represented as an annotation concept $p.c = c_n$ *and inv* (p_{n-1}) *some* $(c_{n-2}$ *...and inv* $(p_2$ *some* $c_1)$ *...)*. This annotation concept uses nested anonymous concepts. In contrast a normalized annotation concept of $p.c$ uses named concepts of the form:
$p.c = Ac_0 = c_n$ *and inv* (p_{n-1}) *some* Ac_1
$Ac_1 = c_{n-2}$ *and inv* (p_{n-3}) *some* Ac_2
$Ac_2 = c_{n-4}$ *and inv* (p_{n-5}) *some* Ac_3
...
$Ac_j = c_3$ *and inv* (p_2) *some* c_1

Definition 8. *Chain of Restrictions of an Annotation Path.*
A chain of restrictions of a normalized annotation concept $p.c$ of an annotation path p is any set of succeeding named concepts Ac_x .. Ac_{x+n} of $p.c$, where $n \geq 1 \wedge x \geq 0 \wedge x + n < |p.c| - 1$.

Theorem 8. *Non Local Invalidations:* When a path is invalid and it is free of local and direct-triple-disjointness invalidations, then there must exist at least one sub-path of two or more triples that conflicts with the ontology.

Proof-Sketch: Given a logically invalid annotation path p_{inv} that is free of local- and direct-triple-disjointness invalidations of m triples of the form

$/t1/../t_m$. From the absence of local invalidations follows that each triple $t \in p_{inv}$ is valid separately. From the absence of intra-triple-disjointness invalidations follows that the intermediate concept that is build by every neighbored pair of triples is satisfiable. Thus, the unsatisfiability of p_{inv} cannot have a local reason. Non-local invalidations are induced by chains of restrictions that conflict with the reference ontology. Each chain of restriction of an annotation path can also be represented as a sub-path of p_{inv}.

Definition 9. *Minimal Invalid Sub-path (MIS):* An invalid sub-path p_s that is free of local or triple disjointness invalidations of an annotation path p is minimal, iff the removal of the first or last triple of p_s yields a satisfiable concept of $p_s.concept$ in O.

We will now discuss which OWL constructs can cause non local invalidations.

- **Chain of Restrictions:** There is is a chain of restrictions defined on concepts in O that produces a clash with the chain of restrictions of the MIS. The chain can be created analogues to our annotation concept or it can be realized with named concepts. The definition may be defined inverse as our annotation concepts or non-inverse. An example was shown in listing 1.3.
- **Transitive Properties:** Transitive properties may also result in an invalidation of an annotation path. An example is shown in listing 1.4. The annotation concept of the annotation path $A/has/B/has/C/has/D$ is unsatisfiable due to the transitivity of the property has, which has the consequence that D has an inverse property definition for has to B and A implicitly. Because D may only have an inverse relation has to C the annotation concept is unsatisfiable.
- **Property Chains:** A property chain has the form $p1\ o\ p2 \rightarrow p3$. It expresses that if there is a chain where some individual $i1$ has a property $p1$ to another individual $i2$ and this individual has a property $p2$ to an individual $i3$, then the individual $i1$ has an assertion for the property $p3$ to $i3$. Of course the chain can have an arbitrary length. This can be seen as a flexible case of transitivity, where the properties in the chain do not need to have an equivalent or sub-property relation in order to produce a transitive chain. Thus, property chains can also be used to produce non-local invalidations.

```
1 Class(A), Class(B), Class(C), Class(D)
2 Class(ClassMyAnnotation)
3 ObjectProperty(has)
4 transitive(has)
5 EquivalentClass(D and inv (has) only C)
6 EquivalentClass(MyAnnotation, D and inv (has)
7     some (C and inv (has) some (B and inv (has) some A)))
8 disjoint(C,B)
```

Listing 1.4. Example ontology for inter-triple invalidations by transitive properties

4.4 An Algorithm for the Detection of a Minimal Invalid Sub-Path

An algorithm for the detection of a minimal invalid sub-path of an annotation path p can be based on a structural search over conflicting axioms in the reference ontology. The last section has shown that such non local conflicts can occur due to many different OWL constructs. Of course a MIS can be the result of a combination of the described causes, which makes an exhaustive search even more complex. The efficiency of such an algorithm is further reduced by the fact that reasoning over sub-, super-, and equivalent-properties and -classes is required. In addition, for such a detection method the first and last triple of a sub-path are not known in advance. This makes another approach that directly operates on definition 9 of a minimal invalid sub-path more efficient. The corresponding algorithm is shown in in listing 1.5. The algorithm takes an invalid path p that is free of local and direct-triple disjointness invalidations as input and returns the index of the start and end triple of the detected MIS.

```
1  (int,int) getMinimalInvalidSubpath(p,O) {
2      r = p.tripleCount()-1;
3      // Find the right border of the MIS
4      while (O.unsatisfiable(createSubPath(p,1,r))
5              r--;
6          }
7      l = 2;
8      // Find the left border of the MIS
9      while (O.unsatisfiable(createSubPath(p,l,r+1))
10             l++;
11         }
12     return(l-1,r+1)
13 }
```

Listing 1.5. An algorithm for the detection of the minimal invalid sub-path

The algorithm uses some helper methods. The method *p.tripleCount()* returns the number of triples of p, *createSubPath(p,l,r)* returns the OWL expression of a sub-path of p that starts at index l and ends at index r. The methods assumes that the leftmost triple of p has the index 1 and the last triple of p has the index $p.tripleCount()$. The method *O.unsatisfiable(owlexp)* returns *true*, if the OWL expression *owlexp* is unsatisfiable in the ontology O.
The first loop is used to find the right boundary of a MIS. This is realized by sequentially creating a sub-path of p that begins at position 1 and end at position r, where r is decremented in each iteration. The loop terminates as soon as the created sub-path gets satisfiable. Therefore, the right boundary of the MIS must be at position $r + 1$. The reason for this is that analogues to theorem 6, there can exist no complete MIS before position r. Otherwise r cannot be satisfiable. After the right boundary was found it is guaranteed that the sub-path between 1 and $r + 1$ is invalid. However, it is not yet sure that it is minimal. Therefore, the left boundary of the MIS needs to be found. This is realized by creating a sub-path that begins at position l and ends at position $r + 1$, where l starts at 2 and it is incremented in each iteration. As soon as such a sub-path

gets satisfiable the left boundary of the MIS has been found at position $l-1$. The detected MIS complies with definition 9 because both iterations guarantee that the removal of the first or last triple of the MIS result in a valid sub-path of p. The algorithm guarantees that is can find one MIS. If a path contains multiple MIS, we propose to remove them iteratively with the help of the proposed algorithm.

Theorem 9. When the algorithm of listing 1.5 is used on a path that is free of local and direct-triple-disjointness invalidations that contains multiple MIS, then the leftmost inner MIS is detected.

Proof-Sketch: Given an annotation path $p = /t_1/t_2/.../t_n$. In the first iteration sub-paths starting at t_1 of p are created. The unsatisfiable sub-path with the minimum number of triples is considered to be a MIS-candidate. According to theorem 6 there can exist no other complete MIS in the path that ends before the MIS candidate. It is only possible that there exists another MIS that starts before and ends after or at the same position as the detected one. In the next loop the minimality of the MIS is guaranteed by chopping elements from the start. As a consequence the algorithm detects the leftmost inner-MIS.

5 Implementation Considerations

In this paper we have defined error-types on annotation paths. The goal is to tell the user, which steps of a path are responsible for the invalidation including an explanation of the type of invalidation. The detection of most invalidation types is straight forward. It is just a query to the reasoner that is equivalent to the definition of the specific invalidation type. Non local invalidations can be tracked by the proposed MIS algorithm. However, testing each triple of every invalid annotation path can be an expensive task. Therefore, we will briefly discuss properties of annotation path that can be used to enormously reduce the number of queries to the reasoner. In a typical scenario there is a set of valid annotations V and a set of invalid annotations I. Both sets are a direct result of the classification of the reference ontology with the added annotation concepts. We now define properties that hold between the elements of V and I.

Theorem 10. *Globally-valid path-postfix:* Given a set of valid annotations V and one invalid annotation i. If there exists an annotation v in V with a common postfix (ending with the same sequence of triples) with i, then the corresponding sub-path of i cannot introduce local or direct-triple-disjointness invalidations.

Proof. No annotation path $\in V$ can contain local or direct-triple-disjointness invalidations. Otherwise it would not be satisfiable. If a path is satisfiable also every postfix of it must be satisfiable due to the monotonicity of OWL. An annotation concept is a specialization of the last concept-step. The longer a path is, the more specific is the annotation concept. When there exists an annotation path $v \in V$ which has the same postfix f as i, then $i.concept$ is a more specific concept than $f.concept$. Thus, the additional specialization must induce the error. It is represented by the prefix of i which does not match f.

Theorem 11. *Globally-valid-triples:* A triple that is an element of a path $\in V$ cannot produce a local invalidation when it is used in a path in I.

Proof. The proof of theorem 11 is a direct consequence of theorem 6. Any triple that is an element of a valid path cannot be logically invalid because otherwise the path would be invalid.

These two properties of a globally valid postfix and triples can be used to find local invalidations or intra-triple-disjointness invalidations very efficiently. As a first step the longest common postfix from i and the annotations in V can be detected. If such a postfix is found it is guaranteed that the corresponding postfix in i cannot contain local or direct-triple disjointness invalidations. In addition all triples in all paths of V can be considered as locally valid triples. Thus, if they occur in i they do not need to be tested for local invalidations.
Finally, when the annotation concepts are represented in form of normalized concepts (see definition 7) in the ontology, it is guaranteed that all triples that correspond to satisfiable named concepts are locally valid and that no direct-triple-disjointness invalidations can exist between succeeding triples, that correspond to satisfiable named concepts in the normalized representation.
All these considerations can lead to a major speedup for the detection of invalidations because triples and combinations of triples that are known to be valid do not need to be checked for specific error-types (domain-invalidation, range-invalidation, ...) and in order to guarantee that the input path of the MIS algorithm is free of local or direct triple-disjointness invalidations, only the potentially invalid triples and combinations of triples need to be checked.

6 Related Work

The basis of this research is a declarative annotation method for XML-Schema published in [7]. The annotation method has two representations: path expressions and complex OWL formulas. Only the path expressions can be changed by the annotators. Therefore, we have proposed methods to track errors in annotation paths. In order to find errors in the corresponding complex OWL formula also general ontology debugging solutions could be used. However, preliminary experiments with the well known OWL1 tool Swoop [12] have shown that Swoop was not able to detect the root-cause of many non local invalidations that only used OWL1 language constructs. In this case the concept was detected to be invalid but no explanation could be generated. When explanations could be generated it was very tedious for the annotator to actually discover which elements in the path were responsible for the problem. The integrated repair tool of Swoop could not help either. In contrast our method can precisely track, which elements in the path are responsible for the invalidation and it is a reasoner-independent black-box approach. In addition we have defined error-types that indicate the reason for the invalidation with respect to the steps of the path.
A fundamental publication in the field of ontology debugging is [14]. It introduces the term of minimal unsatisfiable sub Tboxes (MUPS). A MUPS is a minmal

set of axioms that is responsible for a concept to be unsatisfiable. When one axiom gets removed from the MUPS the concept gets satisfiable unless there are additional MUPS for the concept. This definition is somehow analogues to our definition of the minimal invalid sub-path. In [5] an optimized black box algorithm for the computation of the MUPS is presented. The Black-Box algorithm basically tries to find the MUPS in a trial and error fashion, which requires a high number of expensive reclassifications. In order to get all justifications the authors calculate a first justification (MUPS) and afterwards use a variant of the Hitting Set Algorithm [13] to obtain all other justifications for the unsatisfiability. The goal of general ontology debugging approaches is: Given an ontology with at least one unsatisfiable concept find a set of axioms that need to be removed in order to obtain a coherent ontology. There can be multiple sets of such axioms (also called diagnoses). Therefore, it is beneficial to rank the possible repairs either by assuming that the set of removed axioms should be minimal [14] or by selecting the diagnosis [4] that best fits the modeling intention by asking an oracle/user. This is a major difference to the annotation maintenance scenario, where the ontology cannot be changed by the annotator and only changes of the the path expression are allowed. Therefore, we search especially for steps in the path that lead to an error.
An alternative approach to debug ontologies are patterns/anti-patterns (in particular logically detectable anti-patterns) as proposed in [2, 1]. Those patterns concentrate on common modeling errors that are made on ontology artifacts such as concepts. They can provide well understandable explanations for common errors on simple concepts. Because the subject of such patterns is a concept and not an annotation path their usefulness for annotation paths is limited to simple cases.

7 Conclusion

Semantic annotation is an important technique for semantic processing of information, in particular for interoperability of heterogeneous information systems. Annotation paths are a declarative yet highly expressive way to describe the semantics of artifacts like schemas or documents. We propose methods and algorithms for the identification of deficits in annotation paths, which is based on an in depth analysis of the possible causes for invalid annotations. We expect that experts who annotate artifacts will be much more efficient, if the position and the cause of errors in annotations paths is automatically determined. This technique is also particulary useful for annotation maintenance as a consequence of an evolution of the reference ontology as it not only identifies the annotations which became logically invalid, but also narrows the inspection area to the shortest possible path and gives indications on the causes of the invalidation in form of invalidation types. The proposed algorithm and methods are built upon the functionality usually provided by generic reasoners for OWL ontologies, so they are not restricted to a specific reasoner or ontology management system.

References

1. Oscar Corcho, Catherine Roussey, Luis Manuel Vilches Blazquez, and Ivan Perez. Pattern-based owl ontology debugging guidelines. In *Proc. of WOP2009*, volume 516. CEUR-WS.org, November 2009.
2. Oscar Corcho, Catherine Roussey, Ondrej Zamazal, and francois scharffe. SPARQL-based Detection of Antipatterns in OWL Ontologies, October 2010. Proc. of EKAW2010 Poster and Demo Track.
3. Johann Eder and Christian Koncilia. Modelling changes in ontologies. In *Proc. of OTM Workshops*, volume 3292 of *LNCS*, pages 662–673. Springer, 2004.
4. G. Friedrich K. Shchekotykhin, P. Rodler. Balancing brave and cautions query strategies in ontology debugging. In *Proc. of DX-2011 Workshop*, pages 122–130. DX Society, 2011.
5. Aditya Kalyanpur, Bijan Parsia, Matthew Horridge, and Evren Sirin. Finding all justifications of owl dl entailments. In *Proc. of ISWC/ASWC*, pages 267–280, 2007.
6. Jacek Kopecký, Tomas Vitvar, Carine Bournez, and Joel Farrell. Sawsdl: Semantic annotations for wsdl and xml schema. *IEEE Internet Comp.*, 11(6):60–67, 2007.
7. Julius Köpke and Johann Eder. Semantic annotation of xml-schema for document transformations. In *OTM'10 Workshops*, volume 6428 of *LNCS*, pages 219–228. Springer, 2010.
8. Julius Köpke and Johann Eder. Semantic invalidation of annotations due to ontology evolution. In *OTM 2011 Conf.*, volume 7045 of *LNCS*, pages 763–780. Springer, 2011.
9. Eric Miller and Frank Manola. RDF primer. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.
10. M. Missikoff, F. Schiappelli, and F. Taglino. A controlled language for semantic annotation and interoperability in e-business applications. In *Proc. of ISWC-03*, pages 1–6, 2003.
11. W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009. Available at http://www.w3.org/TR/owl2-overview/.
12. B. Parsia, E. Sirin, and A. Kalyanpur. Debugging owl ontologies. In *Proc. of 14th Int. Conf. WWW*, pages 633–640. ACM Press, 2005.
13. R. Reiter. A theory of diagnosis from first principles. In Joerg Siekmann, editor, *8th International Conference on Automated Deduction*, volume 230 of *LNCS*, pages 153–153. Springer, 1986.
14. Stefan Schlobach and Ronald Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *IJCAI*, pages 355–362, 2003.
15. A.P. Sheth and C. Ramakrishnan. Semantic (web) technology in action: Ontology driven information systems for search, integration and analysis. *IEEE Data Eng. Bull.*, 26(4):40–48, 2003.
16. Marko Vujasinovic, Nenad Ivezic, Boonserm Kulvatunyou, Edward Barkmeyer, Michele Missikoff, Francesco Taglino, Zoran Marjanovic, and Igor Miletic. Semantic mediation for standard-based b2b interoperability. *IEEE Internet Comp.*, 14(1):52–63, 2010.