Semantic Recognition of Ontology Refactoring

Gerd Gröner, Fernando Silva Parreiras, and Steffen Staab

WeST — Institute for Web Science and Technologies
University of Koblenz-Landau
{groener, parreiras, staab}@uni-koblenz.de

Abstract. Ontologies are used for sharing information and are often collaboratively developed. They are adapted for different applications and domains resulting in multiple versions of an ontology that are caused by changes and refactorings. Quite often, ontology versions (or parts of them) are syntactical very different but semantically equivalent. While there is existing work on detecting syntactical and structural changes in ontologies, there is still a need in analyzing and recognizing ontology changes and refactorings by a semantically comparison of ontology versions. In our approach, we start with a classification of model refactorings found in software engineering for identifying such refactorings in OWL ontologies using DL reasoning to recognize these refactorings.

1 Introduction

Ontologies share common knowledge and are often developed in distributed environments. They are combined, extended and reused by other users and knowledge engineers in different applications. In order to support reuse of existing ontologies, remodeling and changes are unavoidable and lead to different ontology versions. Quite often, ontology engineers have to compare different versions and analyze or recognize changes. In order to improve and ease the understandability of changes, it is more beneficial for an engineer to view a more abstract and high-level change description instead of a large number of changed axioms (elementary changes) or ontology version logs like in [1]. Combinations of elementary syntactic changes into more intuitive change patterns are described as refactorings [2] or as composite changes [3].

However, the recognition of refactorings (or changes in general) is difficult due to the variety of possible changes that may be applied to an ontology. Especially if the comparison of different ontology versions is not only realized by a pure syntactical comparison, e.g. a comparison of triples of an ontology, but rather by a semantic comparison of entities in an ontology and their structure.

The need to detect high-level changes is already stated in [1, 4, 5]. High-level understanding of changes provides a foundation for further engineering support like visualization of changes and extended pinpointing focusing on entailments of refactorings rather than individual axiom changes. In order to tackle the described problem, the following issues need to be thoroughly investigated: (i) A high-level categorization of ontology changes like the well established refactoring

patterns in software engineering. (ii) An automatic recognition of refactorings for OWL ontologies that goes beyond mere syntactic comparisons.

The recognition of refactorings is a challenging task due to the variety of possible changes and insufficient means for a semantic comparison of ontology versions. In particular, we identify the issue that we require a semantic comparison of different versions of classes rather than their syntactical comparison. Semantic comparison allows for taking available background knowledge into account while abstracting from elementary changes.

There are different approaches that detect ontology changes by a syntactical comparison like in [4, 6] or the combination of adding and deleting RDF-triples to high-level changes in [5]. A structural comparison using matching algorithms is considered in [7]. More related to our research is the work on version reasoning for ontologies in [8, 9]. However, their focus is on integrity checking, entailment propagation between versions and consistency checking of ontology mappings.

In this paper, we tackle the problem of refactoring recognition using description logics (DL) reasoning in order to semantically compare different versions of an OWL DL ontology. We apply the semantic comparison in heuristic algorithms to recognize refactoring patterns. Extrapolating from [2, 3] we have defined different refactoring patterns of how OWL ontologies may evolve.

We organize this paper as follows. Sect. 2 motivates the problem of schema changes and describes shortcomings of existing approaches. In Sect. 3, we give an overview of the considered refactoring patterns and describe in detail two of them. The comparison of ontology versions and the recognition of the refactoring patterns using DL reasoning is demonstrated in Sect. 4 and 5. The evaluation is given in Sect. 6. We analyze related work in Sect. 7, followed by the conclusion.

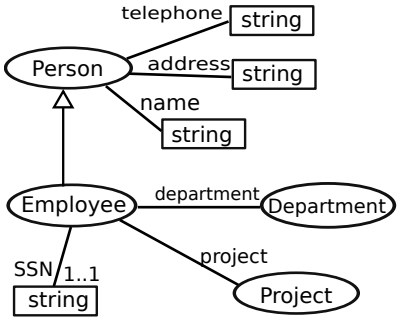
2 An Ontology Refactoring Scenario

In order to clarify the problem we tackle, we start with a motivating example that highlights the problem followed by some argumentation in favor of a semantic version comparison for recognizing refactorings.

2.1 Motivating Example

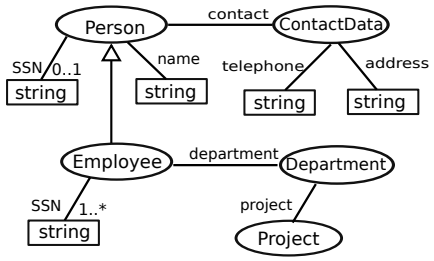
In this section, we consider an ontology change from version V to $V\prime$ including multiple elementary changes. An example is displayed in Fig. 1 and 2. Snippets of the corresponding ontology versions are depicted below. In order to highlight the changed axioms in the example, we mark axioms that are deleted from version V with (d) and axioms that are added are marked with (a) at the end of the line. $Person$, $Employee$, $Project$, $ContactData$ and $Department$ are OWL classes, $Employee$ is a subclass of $Person$. The properties $name$, SSN, $telephone$ and $address$ are datatype properties with range $string$ and $project$, $department$ and $contact$ are object properties.

We recognize three refactorings from version V to $V\prime$. First: the pattern Move Of Property moves a datatype property restriction SSN from class $Employee$ to



$Employee \sqsubseteq Person$
$Employee \sqsubseteq \exists\ project.Project\ (d)$
$Employee \sqsubseteq \exists\ department.Department$
$Employee \sqsubseteq \exists_{=1}\ SSN.string\ (d)$
$Person \sqsubseteq \exists\ name.string$
$Person \sqsubseteq \exists\ address.string\ (d)$
$Person \sqsubseteq \exists\ telephone.string\ (d)$

Fig. 1. Ontology Version V



$Employee \sqsubseteq Person$
$Employee \sqsubseteq \exists\ department.Department$
$Employee \sqsubseteq \exists_{\geq 1} SSN.string\ (a)$
$Person \sqsubseteq \exists\ name.string$
$Person \sqsubseteq \exists_{\leq 1}\ SSN.string\ (a)$
$Person \sqsubseteq \exists\ contact.ContactData\ (a)$
$ContactData \sqsubseteq \exists\ telephone.string\ (a)$
$ContactData \sqsubseteq \exists\ address.string\ (a)$
$Department \sqsubseteq \exists\ project.Project\ (a)$

Fig. 2. Ontology Version $V\prime$

its superclass *Person*. In version V there are implicitly two cardinality restrictions in the property restriction $\exists_{=1}\ SSN.string$. This is semantically equivalent with the restrictions $\exists_{\leq 1}\ SSN.string$ and $\exists_{\geq 1}\ SSN.string$. The datatype property restriction with the maximal cardinality restriction is moved to the superclass *Person*. Second: Extract Class moves the datatype properties *address* and *telephone* to a newly created class *ContactData* that does not contain further properties. In version $V\prime$ the class *Person* has a further object property restriction on *contact* with range *ContactData*. Third: Move Of Property moves an object property *project* from the class *Employee* to the class *Department*.

As demonstrated in the ontology excerpt below the diagrams, the refactorings are syntactically represented by a number of added and deleted axioms from version V to $V\prime$. E.g., the movement of the property *SSN* from *Employee* to its superclass is represented in the ontology by the deleted axiom $Employee \sqsubseteq \exists_{=1}\ SSN.string$ and the added axioms $Person \sqsubseteq \exists_{\leq 1}\ SSN.string$ and $Employee \sqsubseteq \exists_{\geq 1}\ SSN.string$. In order to improve the understanding and recognition of changes between ontology versions, we argue that it is more intuitive for the ontology engineer to characterize changes at a higher abstraction level like by the recognition of refactorings instead of indicating a large collection of added and deleted axioms. For instance, consider the second mentioned refactoring which extracts the datatype properties *address* and *telephone* to the newly created class *ContactData*. Obviously, such a high-level change charac-

terization is more intuitive for an ontology engineer than a listing of changed axioms. In this refactoring at least two axioms are deleted and three axioms are added to the ontology.

2.2 Discussion of Shortcomings

We already argued for the need of a semantic comparison of the versions rather than a syntactic or a purely structural comparison. This is mainly due to the various possibilities of defining classes in OWL compared to RDF(S) like class definitions using intersection, union or property restrictions. We give two examples of shortcomings for syntactical and structural comparisons.

Consider again the third refactoring (Move Of Property) from Fig. 1 and 2. Breaking down this refactoring to axiom changes, we would delete the axiom $Employee \sqsubseteq \exists\ project.Project$ and add the axiom $Department \sqsubseteq \exists\ project.Project$. Now, we slightly extend this refactoring. Suppose there are two subclasses of $Department$, $InternalDepartment$ and $ExternalDepartment$ and the property restriction $\exists\ project.Project$ is moved to both subclasses $InternalDepartment$ and $ExternalDepartment$ rather than to the superclass $Department$. In this case, the ontology contains two new axioms and one is still removed. If there is a further axiom in the ontology describing that each department is either an internal or an external department ($Department \equiv InternalDepartment \sqcup ExternalDepartment$) and there is no other department, we can conclude that after the refactoring *project* is still a property of $Department$. Hence, we identify a refactoring that moves a property (*project*) from a class to another class ($Department$) but without changing an axiom that contains the class itself.

As a second example, we demonstrate shortcomings of structural (and frame-based) comparisons which compare classes and their connections, i.e. domain and range of properties. Consider again the move of the datatype property SSN with maximal cardinality restriction from the class $Employee$ to $Person$. Here, we compare the class $Employee$ in both versions. The cardinality restriction that restricts the class $Employee$ to exactly one SSN is explicitly stated in version V. Semantically, in version $V\prime$ the restriction for class $Employee$ is exactly the same due to inheritance and the conjunction of the minimal and maximal restrictions which also results in exactly one SSN property. This equivalence of the class $Employee$ in both versions is not detected by a purely structural comparison.

3 Modeling and Categorizing Refactoring Patterns

A first step towards the recognition of refactoring patterns is the categorization of well-known patterns, adopted from [2] and also presented as *composite changes* for ontology evolution in RDF(S) [3]. Hereafter, we demonstrate two such refactoring patterns in detail.

3.1 Modeling Foundations and Assumptions

For a more compact notation, we describe a class in version V with C and we use $C\prime$ to refer to this class in version $V\prime$. The class of the range of an object property restriction is called the referenced class. A refactoring pattern is an abstract description of an ontology change or evolution that is applied to realize a certain ontology remodeling. The kind of remodeling is mainly a collection of *best practise* ontology remodeling steps. A refactoring is an instantiation of a refactoring pattern, i.e. a concrete change of an ontology.

Our recognition approach works correctly for a slightly restricted subset of OWL DL where we add two restrictions (Def. 1). The second restriction is also known from OWL Lite (cf. [10]). Both restrictions are necessary in order to avoid exponential computation complexity or even infinite computations in the proposed algorithms that are used in Sect. 4.2 like the ExtractReferenceClasses-Algorithm, e.g., if there are further object property restrictions that appear in the range of another object property restriction.

Definition 1 (Language Restrictions). *We restrict OWL DL ($\mathcal{SHOIN}(D)$) by the following additional conditions:*

1. *In each property restriction $\exists p.E$ and $\forall p.E$, E is a named class. The same condition is also required for cardinality restrictions.*
2. *Individuals are not allowed in class definitions, i.e. no oneOf constructor.*

3.2 Overview of Refactoring Pattern

We start with an overview of the analyzed refactoring patterns and describe how they change an ontology (cf. Table 1). They are adopted from [2, 3].

The first group of refactorings (No. 1-6) extract or merge classes and move properties to or from the extracted or deleted class. Extract Subclass and Extract Superclass are specializations of Extract Class. The refactorings in the second group (No. 7-9) move properties between existing classes. In No. 8 and 9, the properties are moved within a class hierarchy. Finally, the third group collects refactorings that add, delete or modify object property restrictions. Either the inverse object property is added or removed to a class description (No. 10, 11) or in No. 12 cardinality restrictions are modified.

3.3 Detailed Refactoring Descriptions

In this subsection, we give detailed descriptions of two refactoring patterns (Extract Class and Move of Property) and example representations in OWL in order to substantiate our approach. The recognition algorithms and the results for these two examples are given later on in Sect. 4. A comprehensive description of the other considered patterns from Table 1 and the recognition of them is presented in [11]. A refactoring pattern consists of the following elements:

1. Each pattern has a *Name* (cf. pattern overview in Table 1).

No.	Pattern Name	Description
1.	Extract Class	Properties of a class are extracted to a newly created class.
2.	Extract Subclass	Properties of a class are extracted to a newly created subclass.
3.	Extract Superclass	Properties of a class are extracted to a newly created superclass.
4.	Collapse Hierarchy	A subclass and its superclass are merged to one class.
5.	Extract Hierarchy	A class is divided into a class hierarchy. Properties are extracted to the newly created sub- and superclasses.
6.	Inline Class	A class that is referenced by another class is deleted and all its properties are moved to the class that had referenced this class.
7.	Move Of Property	At least one property is moved from a class to a referenced class.
8.	Pull-Up Property	At least one property is moved from a class to its superclass.
9.	Push-Down Property	At least one property is moved from a class to its subclass.
10.	Unidirectional to bidirectional Reference	An object property restriction is added to the target class of an existing object property restriction, where the object property is the inverse property.
11.	Bidirectional to unidirectional Ref.	The inverse property restriction of an object property restriction is removed.
12.	Cardinality Change	The cardinality restriction of a property restriction is changed.

Table 1. Analyzed Refactoring Patterns

2. The *Problem Description* characterizes a modeling structure of an ontology and indicates when this pattern is applicable.
3. The *Solution* describes how the problem is (or could be) solved. This contains the required remodeling steps in order to realize the refactoring.
4. The *Example* demonstrates the technical details of this refactoring.

Extract Class Refactoring Pattern An example of the Extract Class refactoring and the corresponding DL representation is already given in the running example from Fig. 1 and 2 in Sect. 2.

Problem Description In version V, there is a named class C with property restrictions containing the properties $p_1, \ldots, p_n$. An ontology engineer identifies some of the properties $p_{i_1}, \ldots, p_{i_n}$ ($\{p_{i_1}, \ldots, p_{i_n}\} \subseteq \{p_1, \ldots, p_n\}$) that are related to this class but should be grouped together and extracted into a new class D. Finally, a property restriction from class C to the new class D is needed.

Solution A new class D is created and all the selected property restrictions on $p_{i_1}, \ldots, p_{i_n}$ are moved from C to D. An axiom for the object property restriction on p to the new class D is added, e.g. the axiom $C \sqsubseteq \exists\ p.D$.

Example In the example of Fig. 1 and 2, the engineer identifies the property restrictions containing the properties *address* and *telephone* of the class $Person$ in V that should be extracted to a new class. The new class $ContactData$ is created in version $V\prime$ and the identified property restrictions are added by adding axioms to the new class like $ContactData \sqsubseteq \exists\ address.string$. The corresponding axioms of the moved properties are removed from the class definition of the class $Person$. Finally, the object property restriction to the new class is added to $Person$, e.g., by the new axiom $Person \sqsubseteq \exists\ contact.ContactData$.

Move of Property Refactoring Pattern An example of the Move Of Property refactoring and the corresponding description of the ontologies in OWL are already described in the running example of Sect. 2 (Fig. 1 and 2).

Problem Description A named class C has a property restriction on the property p and on the object property r, with the named class D in the range of the definition. The ontology engineer would like to move this property restriction from the class C to the referenced class D.

Solution The identified property restriction on the property p is moved to the class D. The range of this moved property p is unchanged.

Example In the example of Fig. 1 and 2, the property *project* should be moved from the class *Employee* to *Department*. The class *Department* is already referenced by *Employee* with the object property *department*. In version $V\prime$, the corresponding axiom $Employee \sqsubseteq \exists project.Project$ is deleted and the axiom $Department \sqsubseteq \exists project.Project$ is added to the ontology.

4 DL-Reasoning for Ontology Comparison

In this section, we describe the usage of DL reasoning in order to semantically compare ontology versions. We distinguish between three types of comparisons: (i) A *syntactic* comparison checks whether for a class or property in the ontology V there is an entity with the same name in $V\prime$. (ii) The *structural* comparison compares classes and their structure, i.e. sub- and superclass relations and object property restrictions of this class. Hence, a class with all "connected" classes is compared in both versions. (iii) In a *semantic* comparison, classes of both versions are compared using subsumption checking, testing the equivalence, sub- and superclass relations between a class by comparing the interpretations.

4.1 Combining Knowledge Bases

The first step towards a semantic version comparison (Sect. 4.2) is to allow reasoning on two versions of an ontology, e.g., by checking class subsumption of classes from two versions. This requires a renaming of classes that appear in both versions with the same name, otherwise we can not compare them by reasoning. Hence, we start with comparing the names of classes and properties of both versions and rename them. We build a combined, additional knowledge base that captures both, the original version V and the new version $V\prime$. This combined ontology is only a technical mean that is used in order to enable a semantic comparison of classes that appear in both versions. The ontology versions V and $V\prime$ remain unchanged and the semantics given by both versions is also not affected.

For each named class C that occurs in both versions V and $V\prime$, we build the combined knowledge base as follows: (i) The class C is renamed, e.g., C_1 for the class in version V and a class C_2 for the class in version $V\prime$. (ii) Both classes C_1 and C_2 are subclasses of the superclass C. With this step, we guarantee that C_1 and C_2 are still related to each other. C_1 and C_2 are not disjoint. (iii) In

every class expression (anonymous class) if C_i occurs as a class in the range of a property restriction, the class C_i is replaced by its superclass C.

4.2 Semantic Version Comparison

We distinguish between the name or label of a class (C) and the intensional description of the class, i.e. the object and datatype properties that describe the class. The extension of a class, i.e. the set of inferred instances of this class, is denoted using semantic brackets $[\![C]\!]$. A statement like $[\![C]\!] \sqsubseteq A$ means the subsumption $C \sqsubseteq A$ can be inferred.

We use $\hat{C}$ as a representation of the class C in a conjunctive normal form, i.e. $\hat{C} \equiv C_1 \sqcap \ldots \sqcap C_n$ where $\forall i = 1, \ldots, n$ there is an axiom in the ontology $C \sqsubseteq C_i$ and C_i is a class expression. Hence, C is subsumed by each C_i. In order to ease the comparison of classes in two versions, we apply a normalization and reduction of $\hat{C}$ resulting in a reduced conjunctive normal form $\tilde{C}$.

Definition 2 (Reduced Conjunctive Normal Form). *A class definition in conjunctive normal form $\hat{C}$ is reduced to $\tilde{C}$ by the following steps:*

1. *Nested conjunctions are flattened, i.e. $A \sqcap (B \sqcap C)$ becomes $A \sqcap B \sqcap C$.*
2. *Negations are normalized such that in all negations $\neg C$, C is a named class.*
3. *If $B \sqsubseteq A$ can be inferred and $A \sqcup B$ is a class expression in $\hat{C}$, the expression is replaced by A in $\tilde{C}$.*

The main advantage of the normalization is a unique representation that can be assumed for the class definition C which is exploited in the comparison later on. This unique representation is ensured by Lemma 1. The reduced conjunctive normal form $\tilde{C}$ is used in the comparison algorithms later on. We will see, that we are only interested in class expressions C_i that are either property restrictions or named superclasses.

Lemma 1 (Uniqueness of the Reduced Conjunctive Normal Form). *$\hat{C} \equiv C_1 \sqcap \ldots \sqcap C_n$ is a class in conjunctive normal form and $\tilde{C}$ is the reduced conjunctive normal form of the class C. For each class expression C_i $(i = 1, \ldots, n)$ one of the following conditions hold: (i) C_i is a named class, (ii) C_i is a datatype or object property restriction or (iii) C_i is a complex class definition that can neither be a named superclass of C nor a property restriction.*

Proof. It is easy to see whether C_i satisfies the first or second condition, i.e. either C_i is a named class or a property restriction (including qualified property restrictions). In the following, we prove the third condition, assumed that C_i is neither a named class nor a property restriction. We consider the remaining possible class constructors that are allowed according to the language restriction from Def. 1. We show that either the third condition is satisfied or the expression is not allowed after the reduction:

- if $C_i \equiv \neg D$ then C_i cannot be a named superclass of C and (iii) is satisfied.

- $C_i \equiv \neg\forall R.D$ or $C_i \equiv \neg\exists R.D$ is not allowed after the reduction according No. 2 in Def. 2
- $C_i \equiv D \sqcap E$ is not allowed as restricted in No. 1 in Def. 2 (flattening).
- $C_i \equiv D \sqcup E$ then C_i cannot be a named superclass of C. Trivial equivalent representations like $C_i \equiv D \sqcup E$ and $E \sqsubseteq D$ are not allowed (cf. No. 3).

□

Algorithm: Diff(Class C, Ontology versions V, $V\prime$)
Input: Class C and two ontology versions (V, $V\prime$)
Output: Set of class expressions that subsume $C\prime$ in $V\prime$ but not C in V

1: /* Compute the new additional class expressions in $C\prime$ of $V\prime$ */
2: $\mathcal{D} := \emptyset$
3: **for** each asserted class expression A of $\tilde{C}\prime$ ($\tilde{C}\prime \sqsubseteq A$ is asserted in $V\prime$) **do**
4: **if** $[\![C]\!] \not\sqsubseteq A$ in V **then**
5: $\mathcal{D} := \mathcal{D} \cup \{A\}$
6: **end if**
7: **end for**
8: **Return** $\mathcal{D}$.

Fig. 3. The Diff-Algorithm

We use two algorithms to compare versions V and $V\prime$. The Diff-Algorithm (Fig. 3) computes all class expressions that subsume the class $C\prime$ in version $V\prime$, but not C in V. To compute the difference[1], the Diff-Algorithm is used twice. $Diff(C, V, V\prime)$ returns all class expressions that subsume $C\prime$ in $V\prime$. Class expressions that subsume C of V are the result of $Diff(C, V\prime, V)$. $\tilde{C}\prime$ is a class in reduced conjunctive normal form, the expression A is a conjunct that appears in $\tilde{C}\prime$. We can extract the conjuncts due to the normal form representation. In line 4, it is checked whether the subsumption is inferred in version V.

The Common-Algorithm in Fig. 4 extracts the common class expressions of a class C in both versions. Therefore, the subsumption of the class expressions from one version compared with the other is checked in both directions, i.e. $\mathcal{D}_1$ are class expressions from version V that are subsumed by $V\prime$ and $\mathcal{D}_1\prime$ vice versa. $\mathcal{D}$ is the intersection of $\mathcal{D}_1$ and $\mathcal{D}_1\prime$ and consists of all class expressions A from C in both versions. As in the Diff-Algorithm, A is a conjunct of the reduced conjunctive normal forms ($\tilde{C}$, $\tilde{C}\prime$).

We use the ExtractReferenceClasses-Algorithm from Fig. 5 to obtain the classes that are referenced by a class, i.e. we are looking for the class in the range of a property restriction in a class definition. The algorithm uses set operations and returns a set of classes. However, in the considered refactoring patterns, only one class is extracted. If multiple classes are extracted from one class, this

[1] This definition is different from the stronger definition of DL difference of [12], where the difference requires that the minuend is subsumed by the subtrahend.

Algorithm: Common(Class C, Ontology versions V, $V\prime$)
Input: Class C and two ontology versions (V, $V\prime$)
Output: Set of class expressions that subsume C in V and $C\prime$ in $V\prime$

1: /* Common class expressions $\mathcal{D}$ of C and $C\prime$ in both ontology versions V, $V\prime$ */
2: /* $\mathcal{D}_1$ are class expressions of C in V subsumed in $V\prime$, and $\mathcal{D}_1\prime$ are class expressions of $C\prime$ in $V\prime$ subsumed in V. */
3: $\mathcal{D}_1 := \emptyset$ and $\mathcal{D}_1\prime := \emptyset$
4: **for** each asserted class expression A of $\tilde{C}$ ($\tilde{C} \sqsubseteq A$ is asserted in V) **do**
5: **if** $[\![C\prime]\!] \sqsubseteq A$ in $V\prime$ **then**
6: $\mathcal{D}_1 := \mathcal{D}_1 \cup \{A\}$
7: **end if**
8: **end for**
9: **for** each asserted class expression A of $\tilde{C\prime}$ ($\tilde{C\prime} \sqsubseteq A$ is asserted in $V\prime$) **do**
10: **if** $[\![C]\!] \sqsubseteq A$ in V **then**
11: $\mathcal{D}_1\prime := \mathcal{D}_1\prime \cup \{A\}$
12: **end if**
13: **end for**
14: **Return** $\mathcal{D} := \mathcal{D}_1 \cap \mathcal{D}_1\prime$.

Fig. 4. The Common-Algorithm

is considered as multiple refactorings in succession. The input class expression C is an object property restriction like $\exists contact.ContactData$ (line 2). The result is the class that is referenced (R in line 4), e.g., $ContactData$.

The method $getProperty$ returns the object property (object property name) of the given object property restriction (class expression) C. Such methods are provided by OWL-APIs like [13]. The referenced class can not directly be extracted from the expressions using API operations, since in general the expression could be more complex than just a single OWL class as in our applications with language restrictions. Therefore, we have to implement this algorithm. Methods like $IsObjectPropertyRestriction$ or $IsPropertyRestriction$ are provided by APIs as well. For property restrictions with universal quantifiers, the referenced class can be extracted likewise, but this is not required in our approach.

The Diff- and Common-Algorithm compute for a class C, the class expressions C_i that subsume C. These class expressions are expressions C_i of the reduced conjunctive normal form $\tilde{C}$. Hence, all class expressions of the result of the Diff- and Common-Algorithm are in reduced conjunctive normal form too.

The focus of our approach is to recognize the introduced refactorings rather than identifying arbitrary ontology changes. Hence, we can neglect some of the class expressions that are in the result of the Diff- and Common-Algorithm. All the considered refactoring patterns only change sub- and superclass relations and property restrictions in class definitions. Therefore, the only relevant class expressions in the result set of the Diff- and Common-Algorithm are those class expressions that are named classes (representing superclasses) and property restrictions. According to Lemma 1, we can easily determine whether a class ex-

Algorithm: ExtractReferenceClasses(Class expression C, Ontology version V)
Input: Class expression C that is an object property restriction,
e.g., $\exists contact.ContactData$ and an ontology version (V)
Output: Set of classes which are referenced by the object property restriction C
(e.g., the class $ContactData$)

1: $\mathcal{D} := \emptyset$ /* for the referenced classes */
2: **if** IsObjectPropertyRestriction(C) **then**
3: **for** each class R of version V **do**
4: **if** $[\![C]\!] \sqsubseteq \exists\ getProperty(C).\ R$ **then**
5: $\mathcal{D} := \mathcal{D} \cup \{R\}$
6: **end if**
7: **end for**
8: **end if**
9: **Return** $\mathcal{D}$.

Fig. 5. The ExtractReferenceClasses-Algorithm

pression C_i of the result of the algorithms is a superclass, a property restriction or another complex class expression that can be neglected in the comparison.

5 Refactoring Pattern Recognition

In this section, we demonstrate the recognition of the already introduced refactoring patterns Extract Class and Move of Property. The recognition description of the other patterns can be found in [11].

Extract Class This refactoring is illustrated in Fig. 1 and 2. One recognizes the refactoring according to the algorithm in Fig. 6.

The algorithm in Fig. 6 returns the extracted class if the refactoring is successfully recognized, otherwise the result is the empty class ($\bot$). The algorithm works as follows. All named classes C and $C\prime$ that exist in both versions and are different are compared (line 2). In line 3 the difference is computed. For instance, the set $\mathcal{D}_1$ consists of all class expressions which are only in $[\![C\prime]\!]$ of $V\prime$ but not in V. $C\prime$ of $V\prime$ contains exactly one additional object property restriction to another class, i.e. a change only extracts one class. Therefore, we require that $\mathcal{D}_1$ is a singleton (line 4) and that D_1 is an object property restriction (line 6). In line 7, the new class that is referenced by C is extracted. In line 8, we ensure that property restrictions are only moved to one class, i.e. $\mathcal{RC}$ is a singleton. Finally, it is required that all property restrictions are moved correctly to the new class RC (subsumption in line 8). The second and third conditions in line 8 ensure that only property restrictions and no other class expressions are moved and that they are moved to the correct class RC. The result is the referenced class RC ($\mathcal{RC}$ is a singleton).

The recognition algorithms for other extract and merge class refactorings work in the same way. E.g., to recognize an Extract Subclass refactoring, we just

Algorithm: Recognize-ExtractClass(Ontology versions V, $V\prime$)
Input: Ontology versions V and $V\prime$
Output: Extracted Class E

1: $E := \bot$
2: **for** all classes C and $C\prime$ that are different in version V and $V\prime$ **do**
3: $\mathcal{D}_1 := Diff(C, V, V\prime)$ AND $\mathcal{D}_2 := Diff(C, V\prime, V)$
4: **if** $|\mathcal{D}_1| = 1$ **then**
5: $D_1 \in \mathcal{D}_1$:
6: **if** $IsObjectPropertyRestriction(D_1)$ **then**
7: $\mathcal{RC} := ExtractReferenceClasses(D_1, V\prime)$
8: **if** $|\mathcal{RC}| = 1$ AND $\forall\ D_2 \in \mathcal{D}_2 : \exists\ RC \in \mathcal{RC} : [\![RC]\!] \sqsubseteq D_2$ AND $\forall\ D_2 \in \mathcal{D}_2 : IsPropertyRestriction(D_2)$ **then**
9: $E := RC$
10: **end if**
11: **end if**
12: **end if**
13: **end for**
14: **Return** E

Fig. 6. Algorithm for Recognizing *Extract Class*

replace the referenced class (RC) by the corresponding subclass. The recognition result for the example in Fig. 1 and 2 is as follows:

$\mathcal{D}_1 = \{\exists contact.ContactData\}$ (object property restriction in $V\prime$)
$\mathcal{D}_2 = \{\exists address.string,\ \exists telephone.string\}$ (property restrictions in V)
$\mathcal{RC} = \{ContactData\}$ (only one restriction in $\mathcal{D}_1$ ($\exists contact.ContactData$))
$RC = ContactData$ and $[\![RC]\!] \sqsubseteq D_2$ is inferred for all $D_2 \in \mathcal{D}_2$

Move of Property The algorithm in Fig. 7 recognizes the Move of Property refactoring by the following steps. In lines 2-4, it is checked for all classes whether the classes A and $A\prime$ are different in both versions V and $V\prime$ and the referenced classes (range of property) B and $B\prime$ are also different in V and $V\prime$. The common and different class expressions of class A and B in both versions are computed (lines 5-7). If all property restrictions are moved correctly from class A to B the four conditions of line 8 have to be satisfied. Finally, the moved property restrictions are the result of the algorithm (line 9 and 14). Algorithms to detect the other move refactorings like the movement of property restrictions within a class hierarchy work similarly. The recognition of the Move of Property example from Fig. 1 and 2 is as follows:

Common property restrictions of the classes *Employee* and *Department*:
$\mathcal{C}_1 = \{\exists_{=1}\ SSN.string,\ \exists\ department.Department\}$, $\mathcal{C}_2 = \{\}$
Department is referenced by *Employee*: $\exists\ department.Department \in \mathcal{C}_1$
Moved property restrictions (from *Employee* to *Department*):
$\mathcal{A}_1 = \{\}$, $\mathcal{A}_2 = \{\exists\ project.Project\}$, $\mathcal{B}_1 = \{\exists\ project.Project\}$ and $\mathcal{B}_2 = \{\}$

Algorithm: Recognize-MoveOfProperty(Ontology versions V, $V\prime$)
Input: Ontology versions V and $V\prime$
Output: Set of moved property restrictions $\mathcal{P}$

1: $\mathcal{P} := \emptyset$
2: **for** all classes A and $A\prime$ in version V and $V\prime$ that are different **do**
3: **for** all referenced classes B and $B\prime$ **do**
4: **if** B and $B\prime$ are also different in version V and $V\prime$ **then**
5: $\mathcal{C}_1 := Common(A, V, V\prime)$ AND $\mathcal{C}_2 := Common(B, V, V\prime)$ AND
6: $\mathcal{A}_1 := Diff(A, V, V\prime)$ AND $\mathcal{A}_2 := Diff(A, V\prime, V)$ AND
7: $\mathcal{B}_1 := Diff(B, V, V\prime)$ AND $\mathcal{B}_2 := Diff(B, V\prime, V)$
8: **if** $\mathcal{A}_1 = \emptyset$ AND $\mathcal{B}_2 = \emptyset$ AND $\mathcal{A}_2 = \mathcal{B}_1$ AND $\forall E \in \mathcal{A}_2 : IsPropertyRestriction(E)$ **then**
9: $\mathcal{P} := \mathcal{A}_2$
10: **end if**
11: **end if**
12: **end for**
13: **end for**
14: **Return** $\mathcal{P}$

Fig. 7. Algorithm for Recognizing *Move of Property*

6 Evaluation and Discussion

Analysis: We evaluated refactorings for the described refactoring patterns on two ontologies. The DOLCE Lite Plus ontology[2] is the smaller ontology with an average version size of 240 classes and 360 subclass axioms. For each pattern, 8 concrete refactorings were applied. The second ontology is a bio-medical ontology OBI[3] with an average size of 1200 classes, 1700 subclass axioms, and 14 concrete refactorings for each pattern. For both ontologies, we changed the original ontology V by adding and deleting classes, properties and axioms according to the pattern description and applied our recognition algorithms. All recognized refactorings were correctly recognized. The performance result is depicted in Table 2.

For the evaluation, we used the Pellet 2.0.0 reasoner in Java 1.6 on a computer with 2.5 GHz CPU and 2 GB RAM. In Table 2 only the time for the recognition is displayed. The time for matching and combining the ontologies (first step of the comparison) is on average 570 msec for the models with about 240 classes and 2900 msec for models with an average size of 1200 classes.

Limitations We identified the following limitations that are further challenges for future work. (i) The refactoring patterns are adopted from existing work on ontology evolution (cf. [3]), but also on object-oriented modeling (cf. [2]). Therefore, we only recognize those elementary ontology changes that are specified in the refactoring recognition. However, there might be a couple of further

[2] http://www.loa-cnr.it/DOLCE.html
[3] http://obi-ontology.org/page/Main_Page

No.	Refactoring	Recognition (Avg. 240)		Recognition (Avg. 1200)	
		Avg.[msec]	Max.[msec]	Avg.[msec]	Max.[msec]
1.	Extract Class	493	605	2050	2520
2.	Extract Subclass	412	480	1910	2430
3.	Extract Superclass	473	580	1860	2540
4.	Collapse Hierarchy	1062	1154	2260	2480
5.	Extract Hierarchy	886	1042	2170	2410
6.	Inline Class	1042	1075	2330	2590
7.	Move Attribute	1085	1240	2680	3230
8.	Pull-Up Attribute	864	1065	2150	2840
9.	Push-Down Attribute	840	957	2820	3360
10.	Unidirectional to bidirectional Ref.	1170	1254	1820	2140
11.	Bidirectional to unidirectional Ref.	1135	1174	1950	2280
12.	Cardinality Change	1180	1265	1740	1870

Table 2. Analyzed Refactoring Patterns

ontology changes that are not considered in our approach. For instance, we do not consider changes of the property range yet which would lead to difficulties in the current approach in the combination step (cf. Sect. 4). (ii) We need a language restriction as described in Definition 1 and reduction according to Definition 2. Otherwise, we can not ensure the recognition.

Lessons Learned Although the proposed semantic comparison between classes of different versions is the main benefit of our work, the comparison is rather a structural-semantic comparison than a purely semantical comparison. The Diff- and Common-Algorithms iterate and compare class expressions that are either superclasses or property restrictions which is a structural class comparison. The algorithms work properly even for more expressive OWL languages that do not satisfy the restrictions and reductions. However, we need these restrictions in order to guarantee a correct recognition.

7 Related Work

We group the related work into three categories. Firstly, the syntactical comparisons are analyzed. Secondly, related work on structural comparisons is presented. Finally, we consider OWL reasoning for ontology comparison.

The detection of changes of RDF knowledge bases is considered in [14]. High-level changes of RDF-graphs and version differences (RDF triples) are represented and detected in [5]. They categorize elementary changes like add and delete operations to high-level changes which are similar to refactoring patterns. Basically, they analyze the difference of RDF-triples of two RDF-graphs instead

of OWL ontologies and the detection is based on a (syntactical) triple comparison, i.e. the high-level change is detected if all its required low-level changes (RDF-triples) are recognized.

Related work on ontology mappings and the computation of structural differences between OWL ontologies is given in [7, 15, 16]. In [7] a fix-point algorithm is used for comparing and mapping related classes and properties based on their names and structure (references to other entities). A heuristic matching is applied to detect structural differences. Benefits of the heuristics are mainly the identification of related classes and properties if their names have changed.

A framework for tracking ontology changes is introduced in [17]. It is implemented as a plug-in for Protégé [18] that creates a change and annotation ontology to record the changes and meta information on changes. This change ontology is used to display the applied changes to the user. Similarly, change logs are used to manage different ontology versions in [1]. The change logs are realized by a version ontology that represents instances for each class, property and individual of the analyzed ontology. The usage of version ontologies (meta ontology) for change representation is also proposed in [19].

More closely related to our work are the approaches on DL reasoning applying semantic comparison for versioning and ontology changes in OWL. OWL ontology evolution is analyzed in [20]. However, the focus of this work is not on detecting changes. They tackle inconsistency detection caused by already identified changes and in case of an inconsistency, additional changes are generated to result again in a consistent ontology. In [9] and [21], OWL reasoning on modular ontologies is considered in order to tackle the problem of consistency on mappings between ontologies. While the focus in [21] is on reasoning for consistency of ontology mappings and different from our work, in [9] the problem of consistency management for ontology modules is considered. The ontology modules are connected by conjunctive queries instead of merging based on syntactic matching as in our work. Although, subsumption checking is used to compare classes of versions, a classification and especially a recognition of refactoring pattern or complex changes is missing. The main difference to the related work on semantic comparison is the ability of our approach on recognizing ontology refactoring patterns based on change operations in OWL ontologies.

8 Conclusion and Future Work

In this paper, we have demonstrated a structural-semantic comparison approach to recognize specified refactoring patterns using standard DL reasoning. We provide technical information on the version comparison and recognition algorithms. One can apply the results of this work for schema versioning, semantic difference and conflict detection. Additionally, it paves the way for application of reasoning technologies in change prediction of ontologies as well as for guidance in versioning and evolution of ontologies. In future, we plan to cover additional refactoring patterns and plan to extend our approach by a heuristic mapping between classes and properties to handle name changes.

Acknowledgements This work has been supported by the EU Project MOST (ICT-FP7-2008 216691).

References

1. Plessers, P., Troyer, O.D.: Ontology Change Detection Using a Version Log. In: Proc. of the 4th Int. Semantic Web Conference, Springer LNCS (2005) 578–592
2. Fowler, M., Beck, K., Brant, J., Opdyke, W.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
3. Stojanovic, L., Maedche, A., Motik, B., Stojanovic, N.: User-Driven Ontology Evolution Management. In: EKAW. Volume 2473 of LNCS. (2002) 285–300
4. Klein, M., Fensel, D., Kiryakov, A., Ognyanov, D.: Ontology Versioning and Change Detection on the Web. In: EKAW. Volume 2473 of LNCS. (2002) 197–212
5. Papavassiliou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V.: On Detecting High-Level Changes in RDF/S KBs. In: Proc. of ISWC. Volume 5823 of LNCS., Springer (2009) 473–488
6. Noy, N.F., Kunnatur, S., Klein, M.C.A., Musen, M.A.: Tracking Changes During Ontology Evolution. In: Proc. of ISWC. Volume 3298 of LNCS. (2004) 259–273
7. Noy, N.F., Musen, M.A.: PROMPTDIFF: A Fixed-Point Algorithm for Comparing Ontology Versions. In: AAAI/IAAI. (2002) 744–750
8. Meilicke, C., Stuckenschmidt, H., Tamilin, A.: Repairing ontology mappings. In: AAAI. (2007) 1408–1413
9. Stuckenschmidt, H., Klein, M.: Reasoning and Change Management in Modular Ontologies. Data & Knowledge Engineering **63**(2) (2007) 200–223
10. Horrocks, I., Patel-Schneider, P.F., Harmelen, F.V.: From SHIQ and RDF to OWL: The Making of a Web Ontology Language. J. of Web Semantics **1** (2003) 7–26
11. Gröner, G., Staab, S.: Categorization and Recognition of Ontology Refactoring Pattern. Technical Report 9/2010, University of Koblenz-Landau (2010) Available at: http://www.uni-koblenz.de/~groener/documents/TR092010.pdf.
12. Teege, G.: Making the Difference: A subtraction Operation for Description Logics. In: 4th Int. Conference on Knowledge Representation (KR). (1994) 540–550
13. The OWL API: http://owlapi.sourceforge.net (2010)
14. Zeginis, D., Tzitzikas, Y., Christophides, V.: On the Foundations of Computing Deltas between RDF Models. Proc. of ISWC/ASWC **4825** (2007) 637–651
15. Klein, M., Noy, N.: A Component-Based Framework for Ontology Evolution. In: Proc. of the IJCAI-03 Workshop on Ontologies and Distributed Systems, CEUR-WS. Volume 71., Citeseer (2003)
16. Ritze, D., Meilicke, C., Sváb-Zamazal, O., Stuckenschmidt, H.: A Pattern-based Ontology Matching Approach for Detecting Complex Correspondences. In: Proc. of Int. Workshop on Ontology Matching (OM). (2009)
17. Noy, N., Chugh, A., Liu, W., Musen, M.: A Framework for Ontology Evolution in Collaborative Environments. Proc. of ISWC **4273** (2006) 544–558
18. Protégé - Ontology Editor: http://protege.stanford.edu (2010)
19. Palma, R., Haase, P., Wang, Y., dAquin, M.: D1.3.1 Propagation Models and Strategies. Technical report, NeOn Project Deliverable 1.3.1 (2007)
20. Haase, P., Stojanovic, L.: Consistent Evolution of OWL Ontologies. In: ESWC. Volume 3532 of LNCS., Springer (2005) 182–197
21. Meilicke, C., Stuckenschmidt, H., Tamilin, A.: Reasoning Support for Mapping Revision. J. Log. Comput. **19**(5) (2009) 807–829