UNIVERSITÄT
KOBLENZ · LANDAU
Institut WeST

FB 4
Informatik

Categorization and Recognition of Ontology Refactoring Pattern

Gerd Gröner
Steffen Staab

Nr. 9/2010

Arbeitsberichte aus dem Fachbereich Informatik

Die Arbeitsberichte aus dem Fachbereich Informatik dienen der Darstellung vorläufiger Ergebnisse, die in der Regel noch für spätere Veröffentlichungen überarbeitet werden. Die Autoren sind deshalb für kritische Hinweise dankbar. Alle Rechte vorbehalten, insbesondere die der Übersetzung, des Nachdruckes, des Vortrags, der Entnahme von Abbildungen und Tabellen – auch bei nur auszugsweiser Verwertung.

The "Arbeitsberichte aus dem Fachbereich Informatik" comprise preliminary results which will usually be revised for subsequent publication. Critical comments are appreciated by the authors. All rights reserved. No part of this report may be reproduced by any means or translated.

Arbeitsberichte des Fachbereichs Informatik

ISSN (Print): 1864-0346
ISSN (Online): 1864-0850

Herausgeber / Edited by:

Der Dekan:
Prof. Dr. Zöbel

Die Professoren des Fachbereichs:
Prof. Dr. Bátori, Prof. Dr. Burkhardt, Prof. Dr. Diller, Prof. Dr. Ebert, Prof. Dr. Furbach, Prof. Dr. Grimm, Prof. Dr. Hampe, Prof. Dr. Harbusch, Prof. Dr. Lämmel, Prof. Dr. Lautenbach, Prof. Dr. Müller, Prof. Dr. Oppermann, Prof. Dr. Paulus, Prof. Dr. Priese, Prof. Dr. Rosendahl, Prof. Dr. Schubert, Prof. Dr. Staab, Prof. Dr. Steigner, Prof. Dr. Sure, Prof. Dr. Troitzsch, Prof. Dr. von Kortzfleisch, Prof. Dr. Walsh, Prof. Dr. Wimmer, Prof. Dr. Zöbel

Kontaktdaten der Verfasser

Gerd Gröner, Steffen Staab
Institut WeST
Fachbereich Informatik
Universität Koblenz-Landau
Universitätsstraße 1
D-56070 Koblenz
EMail: groener@uni-koblenz.de, staab@uni-koblenz.de

Categorization and Recognition of Ontology Refactoring Pattern

Gerd Gröner and Steffen Staab

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

Abstract. Ontologies play an important role in knowledge representation for sharing information and collaboratively developing knowledge bases. They are changed, adapted and reused in different applications and domains resulting in multiple versions of an ontology. The comparison of different versions and the analysis of changes at a higher level of abstraction may be insightful to understand the changes that were applied to an ontology. While there is existing work on detecting (syntactical) differences and changes in ontologies, there is still a need in analyzing ontology changes at a higher level of abstraction like ontology evolution or refactoring pattern. In our approach we start from a classification of model refactoring patterns found in software engineering for identifying such refactoring patterns in OWL ontologies using DL reasoning to recognize these patterns.

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 a 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]. A combination 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 categorizations of 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 two issues lack a thorough investigation: (i) A high-level categorization of ontology changes like the well established refactoring patterns in software engineering. (ii) An automatic recognition of refactoring patterns for OWL ontologies that goes beyond mere syntactic comparison.

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 like adding and deleting a single axiom.

There are different approaches that detect ontology changes by a syntactical comparison of the classes 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 (modular) OWL ontologies in [8, 9]. However, their focus is mainly 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 logic (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. Section 2 motivates the problem of schema changes and describes shortcomings of existing approaches. In Sections 4, 5 and 6, 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 Sections 7 and 8. The evaluation is given in Section 9. We analyze related work in Section 10, followed by the conclusion.

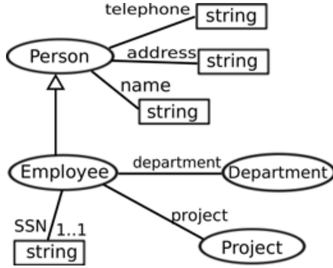
2 An Ontology Refactoring Scenario

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

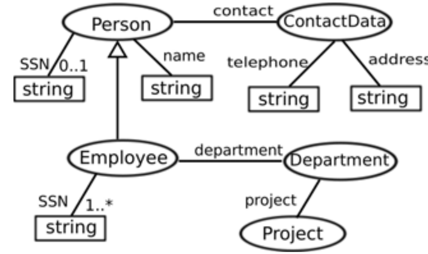
In this section, we consider an ontology change (or ontology evolution) from version V to $V\prime$ that includes multiple elementary changes. An example is displayed in Fig. 1 and 2. Snippets of the corresponding ontology versions are depicted below. In order to highlight the changed axioms in the example, 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.



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

Fig. 1. Ontology Version V



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

Fig. 2. Ontology Version $V\prime$

We recognize three refactorings from version V to $V\prime$. First: the pattern *Pull-Up Property* moves a property restriction SSN (here a datatype property restriction) from class *Employee* to 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 $\exists_{\geq 1}\ SSN.string$. In the example, the datatype property restriction with the maximal cardinality restriction is moved to the superclass *Person*. The minimal cardinality restriction remains in the class *Employee*. 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 *contact* with range *ContactData*. We refer to such an object property as a reference from class *Person* to class *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 $V\prime$. For instance, the movement of the datatype property SSN

from the class *Employee* to its superclass is represented in the ontology by the deleted axiom $Employee \sqsubseteq \exists_{=1}\ SSN.string$ and the newly added axioms (version $V\prime$) $Person \sqsubseteq \exists_{\leq 1}\ SSN.string$ 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 and helpful for the ontology engineer to characterize changes at a higher abstraction level like by the identification of refactoring patterns instead of indicating a large collection of added and deleted axioms. For instance, consider the second mentioned refactoring which extracts the datatype properties *telephone* and *address* to the newly created class *ContactData*. Obviously, such a high-level change characterization 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 of OWL ontologies. 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 for shortcomings of syntactical and structural comparisons.

Consider again the third refactoring (*Move Of Property*) from Fig. 1 and 2, where the object property *project* is moved from the class *Employee* to the class *Department*. 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 classes *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 a property of *Department* as well. Therefore, 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 *Department* itself. This is not possible with a purely syntactical comparison.

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 minimal 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 Foundations and Background

In this section, we start with some modeling principles that are assumed and basic definitions that we require for the language. We use the Web Ontology Language (OWL), or more precisely OWL DL to represent ontologies. The second contribution of this section is the definition of refactoring patterns.

3.1 Modeling Assumptions and Definitions

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 $V\prime$. We use the term reference for an object property restriction. The range of this property is called the referenced class or the target class of a reference.

An inverse reference is used to describe a reference that uses the inverse property and the domain and range classes are switched. For instance, the class C has a reference (object property restriction p) to class D that is an axiom like $C \sqsubseteq \exists\ r.D$ where the right side of the axiom is the property restriction p on the object property r. The inverse reference from D to C uses the inverse object property r^- and is described by an axiom $D \sqsubseteq \exists\ r^-.C$.

In the considered refactoring patterns we are analyzing the change of property restrictions in classes rather than changes of properties. If property restrictions are changed this is always realized by axioms in class definitions like subclass axioms. Hence, we use the term "add property" that means adding a property restriction to a class definition and likewise "delete property" has the meaning of deleting a property restriction to a class definition. In the same way, "move property" is a change that deletes a property restriction in a class definition and adds this property restriction to another class definition.

A refactoring pattern is an abstract description (or template) of an ontology change or evolution that is applied to realize a certain ontology remodeling. The kind of remodeling depends on the ontology engineer and is mainly a collection of *best practise* ontology remodeling and evolution 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 some restrictions known in OWL Lite. Even the more restrictive OWL Lite language fully covers OWL DL except of cardinality restrictions greater than 1 and individuals in class definitions (cf. [10]). The required language restriction is given in Def. 1.

Definition 1 (Language Restrictions). *We restrict OWL DL ($\mathcal{SHOIN}$) 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 Ontology Refactoring Patterns

A basic and famous definition of patterns is given in [11] that defines a pattern as a description of a recurring problem and the corresponding basic solution of the problem such that this pattern can be applied multiple times if the defined problem occurs. A refactoring pattern describes a certain modeling problem (of an ontology) and the corresponding remodeling steps in order to solve the problem. A refactoring pattern consists of the following elements in our approach:

1. The *Name* of the pattern.
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 including the applied changes of the ontology.

Hereafter, we demonstrate refactoring patterns in detail. The patterns are split into three groups and we provide examples for at least one pattern of each group in order to demonstrate the usage of this pattern. We omit examples for those pattern that are quite similar to other patterns.

4 Adding and Deleting Classes

In this group, we collect patterns that add or delete classes combined with moving of properties (object and datatype properties) between these classes. Compared to other refactoring patterns, properties are only moved in combination with creating or deleting classes in these patterns. Furthermore, we categorized these refactoring patterns into three sub-groups. The first sub-group extracts a class from an existing class. This is realized by creating a new class and move properties from an existing class to this new class. In the second sub-group, the pattern *Inline Class* deletes one class and moves the properties to another class. Finally, the third sub-group collects more complex and hierarchical changes like extracting sub- and superclasses simultaneously.

4.1 Extracting a Class

In this subsection, we present three patterns that extract (move) properties from a class to a newly created class. The patterns *Extract Subclass* and *Extract Superclass* are specializations of *Extract Class* where the newly created class is either a sub- or a superclass of the existing class.

Extract Class The first refactoring pattern we consider is *Extract Class* that captures the extraction of properties from an existing class into a newly created class.

Problem Description In version V there is a named class C with property restrictions $p_1, \ldots, p_n$ on properties (datatype and object properties) $r_1, \ldots, r_n$ defined in the ontology version V. An ontology engineer identifies that these property restrictions $p_1, \ldots, p_n$ are still related to this class but should be grouped together and extracted into a new class.

Solution A new class D is created and all the identified or selected property restrictions $p_1, \ldots, p_n$ are moved from C to D. A new object property q is created and an axiom for the reference (object property restriction) on q to the new class D is added that requires for the class C to have a reference to the class D. For instance, we add the axiom $C \sqsubseteq \exists\ q.D$.

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 ontology version like $ContactData \sqsubseteq \exists\ address.string$. The corresponding axioms of the moved properties are removed in the class definition of the class $Person$. Finally, the reference to the new class is added to $Person$, e.g., by the added axiom $Person\ \sqsubseteq\ \exists\ contact.ContactData$.

Extract Subclass This refactoring pattern captures the extraction of properties from an existing class into a newly created subclass.

Problem Description In version V there is a named class C with property restrictions $p_1, \ldots, p_n$ that are property restrictions on properties defined in the ontology V. These property restrictions $p_1, \ldots, p_n$ are related to this class but should be grouped together and extracted into a newly created subclass.

Solution A new class D is created and all the identified or selected property restrictions $p_1, \ldots, p_n$ are moved from C to D. In the ontology this is realized by deleting subclass axioms like $C \sqsubseteq \exists r_i.E$ with C as the subclass and adding subclass axioms with the new class D as subclass, e.g., $D \sqsubseteq \exists r_i.E\ (i = 1, \ldots, n)$. r_i is the property of the property restriction p_i. A further axiom is added to $V\prime$ to represent the subclass relation: $D \sqsubseteq C$.

Extract Superclass This refactoring pattern captures the extraction of properties from an existing class into a newly created superclass.

Problem Description In version V there is a named class C with property restrictions $p_1, \ldots, p_n$ on properties of the ontology versions V. An ontology engineer identifies that these property restrictions should be moved to a superclass that does not exist yet.

Solution A new class D is created and property restrictions $p_1, \ldots, p_n$ are moved from C to D. In the ontology this is realized by deleting subclass axioms like $C \sqsubseteq \exists r_i.E$ and adding subclass axioms with the new class D as subclass, e.g., $D \sqsubseteq \exists r_i.E$ $(i = 1, \ldots, n)$. r_i is the property of the property restriction p_i. A further axiom is added to $V\prime$ to represent the subclass relation: $C \sqsubseteq D$.

4.2 Deleting a Class

The *Inline Class* refactoring pattern describes the deletion of a class including the movement of properties.

Inline Class This refactoring pattern is the inverse of *Extract Class*. A class is deleted and the property restrictions are moved to a class that references this class.

Problem Description There is a named class C and a referenced class D in the ontology version V. The class D contains property restrictions $p_1, \ldots, p_n$ on properties defined in V and the class D is referenced by C with an object property restriction on property q (e.g., there is an axiom $C \sqsubseteq \exists q.D$).

Solution The class D is deleted and all property restrictions $p_1, \ldots, p_n$ are moved from class D to class C. The reference from class C to D is removed, e.g. by deleting the corresponding axiom on q with range of class$D$ like $C \sqsubseteq \exists q.D$. All sub and superclass relations of D are neglected, these relations do no longer exist in $V\prime$.

Example As already mentioned, this is the inverse of the *Extract Class* refactoring pattern. Hence, we can illustrate this by using the example of Fig. 1 and 2 the other way around, i.e. from version $V\prime$ to version V and delete the class $ContactData$ that is referenced by the class $Person$. The property restrictions on the properties *address* and *telephone* of $ContactData$ are moved to the class $Person$.

4.3 Complex Hierarchical Change

We present two refactoring patterns with more complex changes, i.e. at least two classes are involved in a refactoring. Both change the hierarchical structuring of classes.

Collapse Hierarchy A class hierarchy is collapsed by merging a sub- and superclass to one new class. The intuition behind this pattern is that these two classes are not very different and could be represented by one class.

Problem Description In version V there is a named class C that is the superclass of the class (subclass) D. The class C contains property restrictions $p_1, \ldots, p_n$ on properties defined in V and the subclass contains property restrictions $r_1, \ldots, r_n$, respectively. The ontology engineer wants to merge both classes into one, including all property restrictions and the superclasses of C and D.

Solution A new class E is created. All property restrictions are moved from C and D to E. As a next step, all superclasses of D except of the class C are also superclasses of the new class E. For each superclass F of D, we add an axiom $E \sqsubseteq F$ to the ontology and remove the corresponding axiom that define the superclasses of D like $D \sqsubseteq F$. The class F has to be a named class. Due to inheritance, the superclasses of C are also superclasses of D. The classes C and D are removed form the new ontology version $V\prime$.

Extract Hierarchy This pattern extracts sub- and superclasses like *Extract Subclass* and *Extract Superclass* but there can be multiple sub- and superclasses extracted by a change from version V to $V\prime$.

Problem Description In version V there is a named class C with property restrictions $p_{1_1}, \ldots, p_{1_{k_1}}, p_{2_1}, \ldots, p_{2_{k_2}}, \ldots, p_{m_1}, \ldots, p_{m_{k_m}}$ and further property restrictions $p_{m+1_1}, \ldots, p_{m+1_{k_{m+1}}}, p_{m+2_1}, \ldots, p_{m+2_{k_{m+2}}}, \ldots, p_{m+n_1}, \ldots, p_{m+n_{k_{m+n}}}$ on properties defined in the ontology version V. An ontology engineer identifies that these property restrictions are moved to m new superclasses and n new subclasses. Each property restriction is moved to exactly one new super- or subclass.

Solution New classes $C_1, \ldots, C_m, C_{m+1}, \ldots, C_{m+n}$ are created that represent the super- and subclasses of C. Firstly, the new super- and subclass relations are created by adding the corresponding axioms.

- For each new superclass C_i ($i = \{1, \ldots, m\}$) a subclass axiom $C \sqsubseteq C_i$ is added to the ontology.
- For each new subclass C_i ($i = \{m+1, \ldots, m+n\}$) a subclass axiom $C_i \sqsubseteq C$ is added to the ontology.

Secondly, the property restrictions are moved to the corresponding super- and subclasses:

- For each superclass C_i ($i = \{1, \ldots, m\}$) and every property restriction p_j ($j = \{i_1, \ldots, i_{k_i}\}$) the property restriction p_j is moved from C to C_i.
- For each subclass C_i ($i = \{m+1, \ldots, m+n\}$) and every property restriction p_j ($j = \{i_1, \ldots, i_{k_i}\}$) the property restriction p_j is moved from C to C_i.

5 Moving of Property Restrictions

Move of Property In the *Move Of Property* pattern, property restrictions of a class are moved to another existing class.

Problem Description A named class C has property restrictions $p_1, \ldots, p_n$ on properties and it has a reference r to another named class D that is described by an object property restriction in the definition of class C. The ontology engineer would like to move the property restrictions ($p_1, \ldots, p_n$) from the class C to the referenced class D.

Solution The identified property restrictions are moved to the class D. The ranges in these moved property restrictions $p_1, \ldots, p_n$ remain unchanged.

Example In the example of Fig. 1 and 2, the object property restriction on the object property $project$ should be moved from the class $Person$ to $Department$. The class $Department$ is already referenced by the class $Person$ 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.

Pull-Up Property Property restrictions of a class are moved to an existing superclass.

Problem Description A named class C has property restrictions $p_1, \ldots, p_n$ on properties. There is a superclass D that is a named class too. These property restrictions from the class C should be moved to the superclass D.

Solution The identified property restrictions are moved to the superclass D. The ranges in these moved property restrictions $p_1, \ldots, p_n$ are unchanged.

Example In the example of Fig. 1 and 2, the datatype property restriction on the datatype property SSN with the maximal cardinality restriction should be moved from the class $Employee$ to $Person$. That means, the property that requires for each $Employee$ to have at most one SSN should be removed from the employee definition and it should be a property of the superclass $Person$. The minimal cardinality restriction on SSN (≥ 1) remains unchanged. In the ontology, this is realized by deleting the axiom in the definition of class $Employee$: $Employee \sqsubseteq \exists{\geq 1}\ SSN.string$ and by adding the axiom $Person \sqsubseteq \exists{\geq 1}\ SSN.string$ to the ontology.

Push-Down Property The *Push-Down Property* pattern covers the movement of property restrictions of a class to an existing subclass. This pattern is the inverse of *Pull-Up Property*.

Problem Description A named class C has property restrictions $p_1, \ldots, p_n$ on properties. A named class D is the subclass of C. The goal is to move the property restrictions from the class C to its subclass D.

Solution The ranges of the moved property restrictions remain unchanged. The property restrictions are moved to the subclass D.

6 Modifying Property Restrictions

Refactoring patterns that modify property restrictions are describes in this section.

6.1 Adding and Deleting Inverse References

We describe two pattern, one for adding and one for deleting an inverse reference.

Unidirectional to Bidirectional Reference From a conceptual modeling point of view, the refactoring pattern *Unidirectional to Bidirectional Reference* changes a reference from a class to another class into a bidirectional reference, i.e. the reference in the other direction is added. In an ontology, this is realized by property restrictions that use the inverse property of the existing (unidirectional) reference.

Problem Description There is an object property restriction p on a property, defined in the named class C that references another named class D. The aim of the ontology engineer is to add a property restriction to the class D that references the class C and is an inverse of the object property that is restricted in the object property restriction p.

Solution An object property is created that is inverse to the existing property from the property restriction p. A property restriction r is defined in the class D using the same quantifier as in p. The property in the property restriction r is the new inverse property. The range is the class C.

Example An example for this pattern could be created using the reference *department* from the class *Employee* to *Department*, given by the axiom $Employee \sqsubseteq \exists\ department.Department$. Adding the inverse reference would be realized by creating an object property like *employee* that is the inverse of *department* (e.g. $employee = department^{-}$) and the corresponding property restriction on the class definition of the class *Department* is added to the ontology. For instance, the axiom $Department \sqsubseteq \exists\ employee.Employee$ is inserted into the ontology.

Bidirectional to Unidirectional Reference The refactoring pattern *Bidirectional to Unidirectional Reference* is the inverse of *Unidirectional to Bidirectional Reference*, i.e. for a reference to a class its existing inverse reference is removed, resulting in a unidirectional reference.

Problem Description There is an object property restriction p on an object property, defined in the class definition of class C. The class C references the class D. In the named class D, there is also an object property restriction r that references the class C and the property restricted by p is the inverse property of the property that is restricted by the property restriction r.

Solution The property restriction r is removed from the class definition of the class D.

6.2 Changing Reference Cardinality Restrictions

Cardinality Change *Cardinality Change* is a refactoring pattern that describes the change of a cardinality for an existing property restriction (datatype and object property restriction). Cardinalities can be restrictions using *equal*, *greater or equal* and *less or equal* compared with a natural number. Cardinalities only occur in combination with existential quantifies. In case there is no cardinality specified, this represents implicitly an at least one cardinality restriction. A cardinality restriction with equal relation (=) can be represented by two restrictions using greater/less or equal relations. Hence, we can neglect the case of equality condition in cardinality restrictions.

Problem Description There is a property restriction p with cardinality restriction using a comparison relation $\leq$ or $\geq$ and a natural number n. The ontology modeler is interested in changing the number n in the restriction.

Solution The number is changed according to the need of the modeler. In the ontology, this is realized by deleting the axiom describing the cardinality restriction and adding a new axiom with the new number as a cardinality.

Example In the running example of Fig. 1 and 2 the cardinality restriction of the datatype property restriction on SSN is changed. The maximal restriction is changed from 1 to ∞. In the ontology the axiom $Employee \sqsubseteq \exists_{=1}\ SSN.string$ is deleted and the axiom $Employee \sqsubseteq \exists_{\geq 1}\ SSN.string$ is added.

7 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.

7.1 Knowledge Base Merging

The first step of the recognition is a syntactical comparison between ontology version V and $V\prime$. We compare the names (IRIs) of classes and properties of both versions.

Based on the comparison of named classes and properties we build a common knowledge base that captures both, the original version V and the modified version $V\prime$. Furthermore, we have to make the classes of the different versions distinguishable in order to allow a semantic comparison. For each named class C that occurs in both versions V and $V\prime$ we build the common 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. (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. We refer to this procedure as generalization.

7.2 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 instances of this class, is denoted using semantic brackets $[\![C]\!]$.

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 from $\tilde{C}$.

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. *Flattening of nested conjunctions, 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 $B \sqsubseteq A$ holds 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 definition of the reduced conjunctive normal form $\tilde{C}$ is used in the comparison algorithms below. We will see later on, 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 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 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 superclass of D. Trivial equivalent representations like $C_i \equiv D \sqcup E$ and $E \sqsubseteq D$ are not allowed (cf. No. 3 in Def. 2).

□

We demonstrate two algorithms that detect different and common class expressions of a class in two versions. The Diff-Algorithm (Fig. 3) computes all class expressions that subsume the class $C\prime$ in version $V\prime$, but not C in V. In order 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 $Diff(C, V\prime, V)$.

Algorithm: Diff(Class C, Ontology version V, Ontology version $V\prime$)
Input: A class C and two ontology versions (V, $V\prime$)
Output: A set of class expressions which subsumes $C\prime$ in $V\prime$ but not class C in V.

1: /* Compute the new additional class expressions in $C\prime$ of $V\prime$ */
2: $\mathcal{D} := \emptyset$
3: **for** each class expression $A\ of\ \tilde{C}\prime$ of $V\prime$ **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.

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 from C in both versions.

We use the ExtractReferenceClasses-Algorithm from Fig. 5 to obtain the classes that are referenced by the class C, i.e. the range of a property restriction in the class definition of C. The algorithm works as follows. The input class expression C is a reference to another class (object property restriction) like $\exists contact.ContactData$. The result is the class that is referenced, e.g. $ContactData$. The algorithm uses set operations and returns a set of classes.

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

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

Algorithm: Common(Class C, Ontology version V, Ontology version $V\prime$)
Input: A class C and two ontology versions (V, $V\prime$).
Output: A set of class expressions, which subsumes 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 class expression $A \in \tilde{C}$ of 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 class expression $A \in \tilde{C}\prime$ of $V\prime$ **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.

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 recognition 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 refactoring patterns 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 expression 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.

8 Refactoring Pattern Recognition

In this section, we demonstrate the recognition of the already introduced refactoring patterns

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

Algorithm: ExtractReferenceClasses(Class expression C, Ontology version V)
Input: Class expression C that is an object property restriction e.g., $\exists_{=1} contact.ContactData$ and an ontology version (V)
Output: A set of classes which are referenced by the class expression 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.

Algorithm: Recognize-ExtractClass(Ontology version V, Ontology version $V\prime$)
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.*

Algorithm: Recognize-ExtractSubclass(Ontology version V, Ontology version $V\prime$)
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: **for** all classes D that are classes in $V\prime$ but not in V and $D \sqsubseteq C\prime$ holds **do**
4: $\mathcal{D}_1 := Diff(C, V, V\prime)$ AND $\mathcal{D}_2 := Diff(C, V\prime, V)$
5: **if** $\mathcal{D}_1 = \emptyset$ AND $\forall D_2 \in \mathcal{D}_2 : D \sqsubseteq D_2$ AND $\forall D_2 \in \mathcal{D}_2 :$ $IsPropertyRestriction(D_2)$ **then**
6: $E := D$
7: **end if**
8: **end for**
9: **end for**
10: **Return** E

Fig. 7. Algorithm for Recognizing *Extract Subclass.*

The algorithm in Fig. 6 returns the extracted class (version $V\prime$) 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 differently in both versions 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 reference to another class, i.e., a change only extracts one class. Therefore, we require that $\mathcal{D}_1$ is a singleton set (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 set. 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 singleton). The recognition result for the example in Fig. 1 and 2 is as follows:
$\mathcal{D}_1 = \{\exists contact.ContactData\}$ (reference in $V\prime$)
$\mathcal{D}_2 = \{\exists address.string,\ \exists telephone.string\}$ (property restrictions in V)
$\mathcal{RC} = \{ContactData\}$ (The only reference in $\mathcal{D}_1$ is $\exists contact.ContactData$.)
$RC = ContactData$ and $D_2 = \exists address.string$ and $[\![RC]\!] \sqsubseteq D_2$ holds.

Extract Subclass One recognizes the refactoring according to the algorithm in Fig. 7. As in the algorithm for *Extract Class* classes C and $C\prime$ that changed from version V to $V\prime$ are selected (line 2) and the difference is computed (line 4). In line 3, a class is selected that is new in version $V\prime$ and does not exist in version V and this class must be a subclass of $C\prime$. In line 4, the conditions of *Extract Subclass* are checked. $\mathcal{D}_1$ is empty, i.e. no property restriction is moved to C. All extracted property restrictions are contained in the set $\mathcal{D}_2$. The second condition checks whether all class restrictions in $\mathcal{D}_2$ are moved to the new class D and the third condition guarantees that all class expressions in $\mathcal{D}_2$ are property restrictions. This can be easily checked as stated in Lemma 1.

Algorithm: Recognize-InlineClass(Ontology version V, Ontology version $V\prime$)
Input: Ontology versions V and $V\prime$
Output: Deleted 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** $|\mathcal{D}_2| = 1$ **then**
5: $D_2 \in \mathcal{D}_2$:
6: **if** $IsObjectPropertyRestriction(D_2)$ **then**
7: $\mathcal{RC} := ExtractReferenceClasses(D_2, V)$
8: **if** $|\mathcal{RC}| = 1$ AND $\forall D_1 \in \mathcal{D}_1 : \exists RC \in \mathcal{RC} : [\![RC]\!] \sqsubseteq D_1$ AND $\forall D_1 \in \mathcal{D}_1 : IsPropertyRestriction(D_1)$ **then**
9: $E := RC$
10: **end if**
11: **end if**
12: **end if**
13: **end for**
14: **Return** E

Fig. 8. Algorithm for Recognizing *Inline Class*.

Extract Superclass The recognition of *Extract Subclass* works like the algorithm for *Extract Superclass*. The only difference is that we are interested in the superclasses D of C, i.e. we check whether $C\prime \sqsubseteq D$ holds (line 3, Fig. 7).

Inline Class

The *Inline Class* refactoring pattern is the inverse of the *Extract Class* pattern. The recognition works quite similar to the recognition of *Extract Class* (Fig. 6). We describe the recognition of *Inline Class* in the algorithm given in Fig. 8.

The conditions are reverted from the conditions in the recognition of *Extract Class*. $\mathcal{D}_1$ contains the moved property restrictions and $\mathcal{D}_2$ the property restriction (lines 3-5) that is the reference to the class D that becomes the inline class in version $V\prime$. There is only one referenced class (line 8). The other conditions in line 8 ensure that all property restrictions in $\mathcal{D}_1$ are in the class RC in the original version V. (here, RC is the class that becomes the inline class in version $V\prime$). In contrast to the recognition of *Extract Class*, we know that the property restrictions from the set $\mathcal{D}_1$ are in the class C in the version $V\prime$ from the result of the *Diff-Algorithm*.

Collapse Hierarchy The recognition of *Collapse Hierarchy* is demonstrated in Fig. 9. In lines 2 and 3, the super- and subclass from the version V are selected. Similarly, in line 4 the class $E\prime$ is selected that is only in version $V\prime$. The hierarchy of the classes C and D is collapsed into the new single class $E\prime$. Hence, we check in line 5 whether all class restrictions that subsume C and D are moved to the class E.

Extract Hierarchy The recognition of the *Extract Hierarchy* refactoring pattern is described in the algorithm in Fig. 10. As already mentioned in Sect. 4.3, *Extract Hierarchy* is the combination of multiple *Extract Subclass* and

Algorithm: Recognize-CollapseHierarchy(Ontology version V, Ontology version V')
Input: Ontology versions V and V'
Output: The new class E

1: $R := \bot$
2: **for** all classes C that are classes in V but not in V' **do**
3: **for** all classes D that are classes in V but not in V' and $D \sqsubseteq C$ holds **do**
4: **for** all classes E' that are classes in V' but not in V **do**
5: **if** $[\![C]\!] \sqsubseteq [\![E]\!]$ AND $[\![D]\!] \sqsubseteq [\![E]\!]$ **then**
6: $R := D$
7: **end if**
8: **end for**
9: **end for**
10: **end for**
11: **Return** R

Fig. 9. Algorithm for Recognizing *Collapse Hierarchy.*

Extract Superclass refactorings, the algorithm is a combination of the algorithms for these other patterns. In lines 3-8 the extracted subclasses are compared with C' and in lines 9-14 the comparison is applied to the superclasses, respectively.

Move of Property The algorithm in Fig. 11 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' are different in both versions V and V' and the referenced classes B and B' are also different in V and V'. The different class expressions of class A and B in both versions are computed (lines 5-6). If all property restrictions are moved correctly from class A to B the four conditions of line 7 have to be satisfied. Finally, the moved property restrictions are the result of the algorithm (line 8 and 13). Algorithms to detect the other move refactorings like the movement of property restrictions within a class hierarchy work similarly.

The recognition of the example from Fig. 1 and 2 is as follows:
Existing references between these two classes: $C_1 = \exists\, department.Department$
Moved property restrictions:
$\mathcal{A}_1 = \{\}$, $\mathcal{A}_2 = \{\exists\, project.Project\}$ $\mathcal{B}_1 = \{\exists\, project.Project\}$ and $\mathcal{B}_2 = \{\}$

Pull-Up Property The recognition of *Pull-Up Property* is demonstrated in Fig. 12. It is quite similar to the more general pattern *Move Of Property* (Fig. 11). We compare the changes pairs of classes A and B in both versions (i.e. A' and B'). The difference is computed in lines 6,7. However, in this pattern $\mathcal{A}_2$ is also empty due to inheritance, i.e. the properties that are moved to the superclass A' are still properties of B'. Therefore, we need here the Common-Algorithm to compute all common class expression in the subclass B. This set also includes the properties that are moved to the superclass because of inheritance. In the conditions in line 8 we have to check whether the moved properties $\mathcal{B}_2$ are in the common-set $\mathcal{C}_1$ to cope with inheritance.

Push-Down Property The recognition of *Push-Down Property* works similarly to the recognition of *Pull-Up Property* as demonstrated in Fig. 12. We must

Algorithm: Recognize-ExtractHierarchy(Ontology version V, Ontology version $V\prime$)
Input: Ontology versions V and $V\prime$
Output: Set of Extracted Classes $\mathcal{E}$

1: $\mathcal{E} := \emptyset$
2: **for** all classes C and $C\prime$ that are different in version V and $V\prime$ **do**
3: **for** all classes $D\prime$ that are classes in $V\prime$ but not in V and $D\prime \sqsubseteq C\prime$ holds **do**
4: $\mathcal{D}_1 := Diff(C, V, V\prime)$ AND $\mathcal{D}_2 := Diff(C, V\prime, V)$
5: **if** $\mathcal{D}_1 = \emptyset$ AND $\forall\ D_2 \in \mathcal{D}_2 : D \sqsubseteq D_2$ AND $\forall\ D_2 \in \mathcal{D}_2 :$ $IsPropertyRestriction(D_2)$ **then**
6: $\mathcal{E} := \mathcal{E} \cup \{D\}$
7: **end if**
8: **end for**
9: **for** all classes $D\prime$ that are classes in $V\prime$ but not in V and $C\prime \sqsubseteq D\prime$ holds **do**
10: $\mathcal{D}_1 := Diff(C, V, V\prime)$ AND $\mathcal{D}_2 := Diff(C, V\prime, V)$
11: **if** $\mathcal{D}_1 = \emptyset$ AND $\forall\ D_2 \in \mathcal{D}_2 : D \sqsubseteq D_2$ AND $\forall\ D_2 \in \mathcal{D}_2 :$ $IsPropertyRestriction(D_2)$ **then**
12: $\mathcal{E} := \mathcal{E} \cup \{D\}$
13: **end if**
14: **end for**
15: **end for**
16: **Return** $\mathcal{E}$

Fig. 10. Algorithm for Recognizing *Extract Hierarchy.*

consider the subclasses in line 3 and have to take into account the inheritance for the comparison of the class A (subclass) instead of B.

Unidirectional to Bidirectional Reference The algorithm in Fig. 13 describes the recognition of the refactoring pattern *Unidirectional to Bidirectional Reference*.

We start with the classes B and $B\prime$ that changed from version V to $V\prime$ and we choose classes A and $A\prime$ from V and $V\prime$ (lines 2,3). The common and different class expressions of A and $A\prime$ are computed (lines 4,5) and in line 6 the different class expressions of B and $B\prime$ are computed. The class expressions of the class A and $A\prime$ are in the set C_A (Common-Algorithm). The conditions in line 7 guarantee that classes A and $A\prime$ are unchanged, nothing is removed from the class $B\prime$ and only one class expression is added to $B\prime$ ($\mathcal{B}_1$ is singleton). In line 9 it is checked whether this additional class expression in $\mathcal{B}_1$ is an object property restriction, since we are looking for a reference back to the class $A\prime$.

We extract the referenced class RC from this object property restriction B_1. This class must be the class $A\prime$ (line 11). In the next step in line 12, we select a class expression C from the common expressions $\mathcal{C}_A$. The referenced class RC_2 is extracted from this class expression C. This referenced class RC must be the class B. Finally, we have to check whether the object properties of the object property restrictions in C and B_1 are inverse properties. This can be easily done with existing APIs for given object property restrictions.

Bidirectional to Unidirectional Reference The recognition of the refactoring pattern *Bidirectional to Unidirectional Reference* works quite similar to

Algorithm: Recognize-MoveOfProperty(Ontology version V, Ontology version $V\prime$)
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: $\mathcal{A}_1 := Diff(A, V, V\prime)$ AND $\mathcal{A}_2 := Diff(A, V\prime, V)$ AND
6: $\mathcal{B}_1 := Diff(B, V, V\prime)$ AND $\mathcal{B}_2 := Diff(B, V\prime, V)$
7: **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**
8: $\mathcal{P} := \mathcal{A}_2$
9: **end if**
10: **end if**
11: **end for**
12: **end for**
13: **Return** $\mathcal{P}$

Fig. 11. Algorithm for Recognizing *Move of Property.*

Algorithm: Recognize-Pull-UpProperty(Ontology version V, Ontology version $V\prime$)
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 subclasses B and $B\prime$ in both versions **do**
4: **if** B and $B\prime$ are also different in version V and $V\prime$ **then**
5: $\mathcal{C}_1 := Common(B, V, V\prime)$
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 $\mathcal{A}_2 = \emptyset$ AND $\mathcal{B}_1 = \emptyset$ AND $\mathcal{B}_2 \neq \emptyset$ AND $\forall E \in \mathcal{B}_2$: $IsPropertyRestriction(E)$ AND $\mathcal{B}_2 \subseteq \mathcal{C}_1$ **then**
9: $\mathcal{P} := \mathcal{B}_2$
10: **end if**
11: **end if**
12: **end for**
13: **end for**
14: **Return** $\mathcal{P}$

Fig. 12. Algorithm for Recognizing *Pull-Up Property.*

Algorithm: Recognize-Unidirectional2Bidirectional(Ontology version V, Ontology version $V\prime$)
Input: Ontology versions V and $V\prime$
Output: The class B with the added reference

```
1: E = ⊥
2: for all classes B and B′ in version V and V′ that are different do
3:   for all classes A and A′ do
4:     C_A := Common(A, V, V′)
5:     A_1 := Diff(A, V, V′) AND A_2 := Diff(A, V′, V) AND
6:     B_1 := Diff(B, V, V′) AND B_2 := Diff(B, V′, V)
7:     if A_1 = ∅ AND A_2 = ∅ AND B_2 = ∅ AND |B_1| = 1 then
8:       B_1 ∈ B_1:
9:       if IsObjectPropertyRestriction(B_1) then
10:        RC := ExtractReferenceClasses(B_1, V′)
11:        if |RC| = 1 AND ∃RC ∈ RC : RC ≡ A′ then
12:          ∃C ∈ C_A
13:          RC_2 = ExtractReferenceClasses(C, V)
14:          if |RC_2| = 1 AND ∃RC_2 ∈ RC_2 : RC_2 ≡ B AND
             IsInversePropertyInRestriction(C, B_1) then
15:            E := B
16:          end if
17:        end if
18:      end if
19:    end if
20:  end for
21: end for
22: Return E
```

Fig. 13. Algorithm for Recognizing *Unidirectional to Bidirectional Reference.*

the recognition of *Unidirectional to Bidirectional Reference* in Fig. 13. The conditions to compare the versions are the same.

Change Cardinality The pattern *Cardinality Change* compares changes of cardinalities in property restrictions for classes in version V with classes from version $V\prime$. The recognition is described in Fig. 14. The differences are computed in line 3. There is exactly one class expression added and one removed from C to $C\prime$ (line 4). Both of these class expressions must be property restrictions with cardinalities (line 7). We have to use some API operations to test whether it is datatype or an object property restriction (lines 8 and 10). In case of object properties, besides the property name we also require the equivalence of the referenced class to avoid meaningless comparisons.

A change of the cardinality restriction leads to either a more general or a more specific property restriction. This is checked in both cases by comparing the subsumption of the class expressions (property restrictions with cardinality) $D_1 \sqsubset D_2$ or $D_2 \sqsubset D_1$ (line 8 or 13).

Algorithm: Recognize-CardinalityChange(Ontology version V, Ontology version $V\prime$)
Input: Ontology versions V and $V\prime$
Output: Class E with changed cardinality restriction

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** $|\mathcal{D}_1| = 1$ AND $|\mathcal{D}_2| = 1$ **then**
5: $D_1 \in \mathcal{D}_1$
6: $D_2 \in \mathcal{D}_2$:
7: **if** $IsCardinalityRestriction(D_1)$ AND $IsCardinalityRestriction(D_2)$ **then**
8: **if** $IsDataTypeRestriction(D_1)$ AND $IsDataTypeRestriction(D_2)$ AND $SameProperty(D_1, D_2)$ AND ($D_1 \sqsubset D_2$ OR $D_2 \sqsubset D_1$) **then**
9: $E := C\prime$
10: **else if** $IsObjectPropertyRestriction(D_1)$ AND $IsObjectPropertyRestriction(D_2)$ AND $SameProperty(D_1, D2)$ **then**
11: $RC_1 := ExtractReferencedClasses(D_1, V\prime)$
12: $RC_2 := ExtractReferencedClasses(D_2, V)$
13: **if** $RC_1 = RC_2$ AND ($D_1 \sqsubset D_2$ OR $D_2 \sqsubset D_1$) **then**
14: $E := C\prime$
15: **end if**
16: **end if**
17: **end if**
18: **end if**
19: **end for**
20: **Return** E

Fig. 14. Algorithm for Recognizing *Cardinality Change.*

9 Evaluation and Discussion

Analysis: We evaluated refactorings for the described refactoring patterns on two ontologies with different sizes. 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 approach to recognize the refactorings. All recognized refactorings were correctly recognized. The performance result is depicted in Table 1.

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 1. Analyzed Refactoring Patterns.

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 1 only the time for the recognition is displayed. The time for matching and merging 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

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 merging step of the ontology versions due to the applied axiom generalization (cf. Sect. 7). (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 approach, 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.

10 Related Work

We group the related work into three categories. Firstly, the syntactical comparisons are analyzed, including also syntactical comparison of RDF triples. 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 realized 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 detected) 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.

11 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.

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., Springer (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., Springer (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. Alexander, C.: A Pattern Language. Towns, Buildings, Construction. Oxford University Press, New York (1977)
12. Teege, G.: Making the Difference: A subtraction Operation for Description Logics. In: Proc. of the 4th Int. Conf. on Knowledge Representation (KR'94). 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 (207)
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

Bisher erschienen

Arbeitsberichte aus dem Fachbereich Informatik

(http://www.uni-koblenz-landau.de/koblenz/fb4/publications/Reports/arbeitsberichte)

Gerd Gröner, Steffen Staab, Categorization and Recognition of Ontology Refactoring Pattern, Arbeitsberichte aus dem Fachbereich Informatik 9/2010

Daniel Eißing, Ansgar Scherp, Carsten Saathoff, Integration of Existing Multimedia Metadata Formats and Metadata Standards in the M3O, Arbeitsberichte aus dem Fachbereich Informatik 8/2010

Stefan Scheglmann, Ansgar Scherp, Steffen Staab, Model-driven Generation of APIs for OWL-based Ontologies, Arbeitsberichte aus dem Fachbereich Informatik 7/2010

Daniel Schmeiß, Ansgar Scherp, Steffen Staab, Integrated Mobile Visualization and Interaction of Events and POIs, Arbeitsberichte aus dem Fachbereich Informatik 6/2010

Rüdiger Grimm, Daniel Pähler, E-Mail-Forensik – IP-Adressen und ihre Zuordnung zu Internet-Teilnehmern und ihren Standorten, Arbeitsberichte aus dem Fachbereich Informatik 5/2010

Christoph Ringelstein, Steffen Staab, PAPEL: Syntax and Semantics for Provenance-Aware Policy Definition, Arbeitsberichte aus dem Fachbereich Informatik 4/2010

Nadine Lindermann, Sylvia Valcárcel, Harald F.O. von Kortzfleisch, Ein Stufenmodell für kollaborative offene Innovationsprozesse in Netzwerken kleiner und mittlerer Unternehmen mit Web 2.0, Arbeitsberichte aus dem Fachbereich Informatik 3/2010

Maria Wimmer, Dagmar Lück-Schneider, Uwe Brinkhoff, Erich Schweighofer, Siegfried Kaiser, Andreas Wieber, Fachtagung Verwaltungsinformatik FTVI Fachtagung Rechtsinformatik FTRI 2010, Arbeitsberichte aus dem Fachbereich Informatik 2/2010

Max Braun, Ansgar Scherp, Steffen Staab, Collaborative Creation of Semantic Points of Interest as Linked Data on the Mobile Phone, Arbeitsberichte aus dem Fachbereich Informatik 1/2010

Marc Santos, Einsatz von „Shared In-situ Problem Solving" Annotationen in kollaborativen Lern- und Arbeitsszenarien, Arbeitsberichte aus dem Fachbereich Informatik 20/2009

Carsten Saathoff, Ansgar Scherp, Unlocking the Semantics of Multimedia Presentations in the Web with the Multimedia Metadata Ontology, Arbeitsberichte aus dem Fachbereich Informatik 19/2009

Christoph Kahle, Mario Schaarschmidt, Harald F.O. von Kortzfleisch, Open Innovation: Kundenintegration am Beispiel von IPTV, Arbeitsberichte aus dem Fachbereich Informatik 18/2009

Dietrich Paulus, Lutz Priese, Peter Decker, Frank Schmitt, Pose-Tracking Forschungsbericht, Arbeitsberichte aus dem Fachbereich Informatik 17/2009

Andreas Fuhr, Tassilo Horn, Andreas Winter, Model-Driven Software Migration Extending SOMA, Arbeitsberichte aus dem Fachbereich Informatik 16/2009

Eckhard Großmann, Sascha Strauß, Tassilo Horn, Volker Riediger, Abbildung von grUML nach XSD soamig, Arbeitsberichte aus dem Fachbereich Informatik 15/2009

Kerstin Falkowski, Jürgen Ebert, The STOR Component System Interim Report, Arbeitsberichte aus dem Fachbereicht Informatik 14/2009

Sebastian Magnus, Markus Maron, An Empirical Study to Evaluate the Location of Advertisement Panels by Using a Mobile Marketing Tool, Arbeitsberichte aus dem Fachbereich Informatik 13/2009

Sebastian Magnus, Markus Maron, Konzept einer Public Key Infrastruktur in iCity, Arbeitsberichte aus dem Fachbereich Informatik 12/2009

Sebastian Magnus, Markus Maron, A Public Key Infrastructure in Ambient Information and Transaction Systems, Arbeitsberichte aus dem Fachbereich Informatik 11/2009

Ammar Mohammed, Ulrich Furbach, Multi-agent systems: Modeling and Virification using Hybrid Automata, Arbeitsberichte aus dem Fachbereich Informatik 10/2009

Andreas Sprotte, Performance Measurement auf der Basis von Kennzahlen aus betrieblichen Anwendungssystemen: Entwurf eines kennzahlengestützten Informationssystems für einen Logistikdienstleister, Arbeitsberichte aus dem Fachbereich Informatik 9/2009

Gwendolin Garbe, Tobias Hausen, Process Commodities: Entwicklung eines Reifegradmodells als Basis für Outsourcingentscheidungen, Arbeitsberichte aus dem Fachbereich Informatik 8/2009

Petra Schubert et. al., Open-Source-Software für das Enterprise Resource Planning, Arbeitsberichte aus dem Fachbereich Informatik 7/2009

Ammar Mohammed, Frieder Stolzenburg, Using Constraint Logic Programming for Modeling and Verifying Hierarchical Hybrid Automata, Arbeitsberichte aus dem Fachbereich Informatik 6/2009

Tobias Kippert, Anastasia Meletiadou, Rüdiger Grimm, Entwurf eines Common Criteria-Schutzprofils für Router zur Abwehr von Online-Überwachung, Arbeitsberichte aus dem Fachbereich Informatik 5/2009

Hannes Schwarz, Jürgen Ebert, Andreas Winter, Graph-based Traceability – A Comprehensive Approach. Arbeitsberichte aus dem Fachbereich Informatik 4/2009

Anastasia Meletiadou, Simone Müller, Rüdiger Grimm, Anforderungsanalyse für Risk-Management-Informationssysteme (RMIS), Arbeitsberichte aus dem Fachbereich Informatik 3/2009

Ansgar Scherp, Thomas Franz, Carsten Saathoff, Steffen Staab, A Model of Events based on a Foundational Ontology, Arbeitsberichte aus dem Fachbereich Informatik 2/2009

Frank Bohdanovicz, Harald Dickel, Christoph Steigner, Avoidance of Routing Loops, Arbeitsberichte aus dem Fachbereich Informatik 1/2009

Stefan Ameling, Stephan Wirth, Dietrich Paulus, Methods for Polyp Detection in Colonoscopy Videos: A Review, Arbeitsberichte aus dem Fachbereich Informatik 14/2008

Tassilo Horn, Jürgen Ebert, Ein Referenzschema für die Sprachen der IEC 61131-3, Arbeitsberichte aus dem Fachbereich Informatik 13/2008

Thomas Franz, Ansgar Scherp, Steffen Staab, Does a Semantic Web Facilitate Your Daily Tasks?, Arbeitsberichte aus dem Fachbereich Informatik 12/2008

Norbert Frick, Künftige Anfordeungen an ERP-Systeme: Deutsche Anbieter im Fokus, Arbeitsberichte aus dem Fachbereicht Informatik 11/2008

Jürgen Ebert, Rüdiger Grimm, Alexander Hug, Lehramtsbezogene Bachelor- und Masterstudiengänge im Fach Informatik an der Universität Koblenz-Landau, Campus Koblenz, Arbeitsberichte aus dem Fachbereich Informatik 10/2008

Mario Schaarschmidt, Harald von Kortzfleisch, Social Networking Platforms as Creativity Fostering Systems: Research Model and Exploratory Study, Arbeitsberichte aus dem Fachbereich Informatik 9/2008

Bernhard Schueler, Sergej Sizov, Steffen Staab, Querying for Meta Knowledge, Arbeitsberichte aus dem Fachbereich Informatik 8/2008

Stefan Stein, Entwicklung einer Architektur für komplexe kontextbezogene Dienste im mobilen Umfeld, Arbeitsberichte aus dem Fachbereich Informatik 7/2008

Matthias Bohnen, Lina Brühl, Sebastian Bzdak, RoboCup 2008 Mixed Reality League Team Description, Arbeitsberichte aus dem Fachbereich Informatik 6/2008

Bernhard Beckert, Reiner Hähnle, Tests and Proofs: Papers Presented at the Second International Conference, TAP 2008, Prato, Italy, April 2008, Arbeitsberichte aus dem Fachbereich Informatik 5/2008

Klaas Dellschaft, Steffen Staab, Unterstützung und Dokumentation kollaborativer Entwurfs- und Entscheidungsprozesse, Arbeitsberichte aus dem Fachbereich Informatik 4/2008

Rüdiger Grimm: IT-Sicherheitsmodelle, Arbeitsberichte aus dem Fachbereich Informatik 3/2008

Rüdiger Grimm, Helge Hundacker, Anastasia Meletiadou: Anwendungsbeispiele für Kryptographie, Arbeitsberichte aus dem Fachbereich Informatik 2/2008

Markus Maron, Kevin Read, Michael Schulze: CAMPUS NEWS – Artificial Intelligence Methods Combined for an Intelligent Information Network, Arbeitsberichte aus dem Fachbereich Informatik 1/2008

Lutz Priese,Frank Schmitt, Patrick Sturm, Haojun Wang: BMBF-Verbundprojekt 3D-RETISEG Abschlussbericht des Labors Bilderkennen der Universität Koblenz-Landau, Arbeitsberichte aus dem Fachbereich Informatik 26/2007

Stephan Philippi, Alexander Pinl: Proceedings 14. Workshop 20.-21. September 2007 Algorithmen und Werkzeuge für Petrinetze, Arbeitsberichte aus dem Fachbereich Informatik 25/2007

Ulrich Furbach, Markus Maron, Kevin Read: CAMPUS NEWS – an Intelligent Bluetooth-based Mobile Information Network, Arbeitsberichte aus dem Fachbereich Informatik 24/2007

Ulrich Furbach, Markus Maron, Kevin Read: CAMPUS NEWS - an Information Network for Pervasive Universities, Arbeitsberichte aus dem Fachbereich Informatik 23/2007

Lutz Priese: Finite Automata on Unranked and Unordered DAGs Extented Version, Arbeitsberichte aus dem Fachbereich Informatik 22/2007

Mario Schaarschmidt, Harald F.O. von Kortzfleisch: Modularität als alternative Technologie- und Innovationsstrategie, Arbeitsberichte aus dem Fachbereich Informatik 21/2007

Kurt Lautenbach, Alexander Pinl: Probability Propagation Nets, Arbeitsberichte aus dem Fachbereich Informatik 20/2007

Rüdiger Grimm, Farid Mehr, Anastasia Meletiadou, Daniel Pähler, Ilka Uerz: SOA-Security, Arbeitsberichte aus dem Fachbereich Informatik 19/2007

Christoph Wernhard: Tableaux Between Proving, Projection and Compilation, Arbeitsberichte aus dem Fachbereich Informatik 18/2007

Ulrich Furbach, Claudia Obermaier: Knowledge Compilation for Description Logics, Arbeitsberichte aus dem Fachbereich Informatik 17/2007

Fernando Silva Parreiras, Steffen Staab, Andreas Winter: TwoUse: Integrating UML Models and OWL Ontologies, Arbeitsberichte aus dem Fachbereich Informatik 16/2007

Rüdiger Grimm, Anastasia Meletiadou: Rollenbasierte Zugriffskontrolle (RBAC) im Gesundheitswesen, Arbeitsberichte aud dem Fachbereich Informatik 15/2007

Ulrich Furbach, Jan Murray, Falk Schmidsberger, Frieder Stolzenburg: Hybrid Multiagent Systems with Timed Synchronization-Specification and Model Checking, Arbeitsberichte aus dem Fachbereich Informatik 14/2007

Björn Pelzer, Christoph Wernhard: System Description:"E-KRHyper", Arbeitsberichte aus dem Fachbereich Informatik, 13/2007

Ulrich Furbach, Peter Baumgartner, Björn Pelzer: Hyper Tableaux with Equality, Arbeitsberichte aus dem Fachbereich Informatik, 12/2007

Ulrich Furbach, Markus Maron, Kevin Read: Location based Informationsystems, Arbeitsberichte aus dem Fachbereich Informatik, 11/2007

Philipp Schaer, Marco Thum: State-of-the-Art: Interaktion in erweiterten Realitäten, Arbeitsberichte aus dem Fachbereich Informatik, 10/2007

Ulrich Furbach, Claudia Obermaier: Applications of Automated Reasoning, Arbeitsberichte aus dem Fachbereich Informatik, 9/2007

Jürgen Ebert, Kerstin Falkowski: A First Proposal for an Overall Structure of an Enhanced Reality Framework, Arbeitsberichte aus dem Fachbereich Informatik, 8/2007

Lutz Priese, Frank Schmitt, Paul Lemke: Automatische See-Through Kalibrierung, Arbeitsberichte aus dem Fachbereich Informatik, 7/2007

Rüdiger Grimm, Robert Krimmer, Nils Meißner, Kai Reinhard, Melanie Volkamer, Marcel Weinand, Jörg Helbach: Security Requirements for Non-political Internet Voting, Arbeitsberichte aus dem Fachbereich Informatik, 6/2007

Daniel Bildhauer, Volker Riediger, Hannes Schwarz, Sascha Strauß, „grUML – Eine UML-basierte Modellierungssprache für T-Graphen", Arbeitsberichte aus dem Fachbereich Informatik, 5/2007

Richard Arndt, Steffen Staab, Raphaël Troncy, Lynda Hardman: Adding Formal Semantics to MPEG-7: Designing a Well Founded Multimedia Ontology for the Web, Arbeitsberichte aus dem Fachbereich Informatik, 4/2007

Simon Schenk, Steffen Staab: Networked RDF Graphs, Arbeitsberichte aus dem Fachbereich Informatik, 3/2007

Rüdiger Grimm, Helge Hundacker, Anastasia Meletiadou: Anwendungsbeispiele für Kryptographie, Arbeitsberichte aus dem Fachbereich Informatik, 2/2007

Anastasia Meletiadou, J. Felix Hampe: Begriffsbestimmung und erwartete Trends im IT-Risk-Management, Arbeitsberichte aus dem Fachbereich Informatik, 1/2007

„Gelbe Reihe"
(http://www.uni-koblenz.de/fb4/publikationen/gelbereihe)

Lutz Priese: Some Examples of Semi-rational and Non-semi-rational DAG Languages. Extended Version, Fachberichte Informatik 3-2006

Kurt Lautenbach, Stephan Philippi, and Alexander Pinl: Bayesian Networks and Petri Nets, Fachberichte Informatik 2-2006

Rainer Gimnich and Andreas Winter: Workshop Software-Reengineering und Services, Fachberichte Informatik 1-2006

Kurt Lautenbach and Alexander Pinl: Probability Propagation in Petri Nets, Fachberichte Informatik 16-2005

Rainer Gimnich, Uwe Kaiser, and Andreas Winter: 2. Workshop "Reengineering Prozesse" – Software Migration, Fachberichte Informatik 15-2005

Jan Murray, Frieder Stolzenburg, and Toshiaki Arai: Hybrid State Machines with Timed Synchronization for Multi-Robot System Specification, Fachberichte Informatik 14-2005

Reinhold Letz: FTP 2005 – Fifth International Workshop on First-Order Theorem Proving, Fachberichte Informatik 13-2005

Bernhard Beckert: TABLEAUX 2005 – Position Papers and Tutorial Descriptions, Fachberichte Informatik 12-2005

Dietrich Paulus and Detlev Droege: Mixed-reality as a challenge to image understanding and artificial intelligence, Fachberichte Informatik 11-2005

Jürgen Sauer: 19. Workshop Planen, Scheduling und Konfigurieren / Entwerfen, Fachberichte Informatik 10-2005

Pascal Hitzler, Carsten Lutz, and Gerd Stumme: Foundational Aspects of Ontologies, Fachberichte Informatik 9-2005

Joachim Baumeister and Dietmar Seipel: Knowledge Engineering and Software Engineering, Fachberichte Informatik 8-2005

Benno Stein and Sven Meier zu Eißen: Proceedings of the Second International Workshop on Text-Based Information Retrieval, Fachberichte Informatik 7-2005

Andreas Winter and Jürgen Ebert: Metamodel-driven Service Interoperability, Fachberichte Informatik 6-2005

Joschka Boedecker, Norbert Michael Mayer, Masaki Ogino, Rodrigo da Silva Guerra, Masaaki Kikuchi, and Minoru Asada: Getting closer: How Simulation and Humanoid League can benefit from each other, Fachberichte Informatik 5-2005

Torsten Gipp and Jürgen Ebert: Web Engineering does profit from a Functional Approach, Fachberichte Informatik 4-2005

Oliver Obst, Anita Maas, and Joschka Boedecker: HTN Planning for Flexible Coordination Of Multiagent Team Behavior, Fachberichte Informatik 3-2005

Andreas von Hessling, Thomas Kleemann, and Alex Sinner: Semantic User Profiles and their Applications in a Mobile Environment, Fachberichte Informatik 2-2005

Heni Ben Amor and Achim Rettinger: Intelligent Exploration for Genetic Algorithms – Using Self-Organizing Maps in Evolutionary Computation, Fachberichte Informatik 1-2005