Anomalies in Ontologies with Rules

Joachim Baumeister^a and Dietmar Seipel^a

^aUniversity of Würzburg, Institute of Computer Science, Am Hubland, 97074 Würzburg, Germany

Abstract

For the development of practical semantic applications, ontologies are commonly used with rule extensions. Prominent examples of semantic applications are Semantic Wikis, Semantic Desktops, but also advanced Web Services and agents. The application of rules increases the expressiveness of the underlying knowledge in many ways. Likewise, the integration creates new challenges for the design process of such ontologies, but also existing evaluation methods have to cope with the extension of ontologies by rules.

Since the verification of OWL ontologies with rule extensions is not tractable in general, we propose to verify ontologies at the symbolic level by using a declarative approach: With the new language DATALOG^{*}, known anomalies can be easily specified and tested in a compact manner. We introduce supplements to existing verification techniques to support the design of ontologies with rule enhancements, and we focus on the detection of anomalies that especially occur due to the combined use of rules and ontological definitions.

Key words: evaluation, verification, ontology engineering, OWL, RIF-BLD, SWRL

1. Introduction

The use of ontologies has shown its benefits in many applications of intelligent systems in the last years. Recent examples are the development of Semantic Wikis, e.g., [7,28], and Semantic Desktops, e.g., [27]. Most prominently, the Semantic Web initiative [35] coordinates the specification and life cycle of ontology languages in the context of the Semantic Web [1]. The semantic web stack, e.g., see [16], describes the architecture of the Semantic Web at a technical level including languages for ontologies and rules, but also key technologies such as Unicode and encryption. Whereas, the implementation of lower parts of the semantic web stack has successfully led to a standardization, the upper parts, especially rules and the logic framework, are

still heavily discussed in the research community, see for example [16,19,26]. This insight has led to many proposals for rule languages compatible with the semantic web stack, e.g., the definition of RIF-BLD (Basic Logic Dialect of the Rule Interchange Format, [33]), SWRL (semantic web rule language) which originates from RULEML, and similar approaches [17]. It is generally agreed that the combination of ontologies with rule-based knowledge is essential for many interesting semantic web tasks such as the realization of semantic web agents and services. SWRL allows for the combination of a high-level abstract syntax for Horn-like rules with OWL, and a model theoretic semantics is given for the combination of OWL with SWRL rules. The XML syntax was derived from RULEML. With RIF-BLD an analogous XML serialization of rules is in the process of standardization. RIF-BLD specifies an interchange format for rule languages and proposes an integration with RIF-BLD/OWL languages.

Email addresses: joba@uni-wuerzburg.de
(Joachim Baumeister), seipel@informatik.uni-wuerzburg.de
(Dietmar Seipel).

However, with the increased expressiveness of such ontologies new demands for development and maintenance guidelines arise. Thus, conventional approaches for evaluating and maintaining ontologies need to be extended and revised in the light of rules, and new measures need to be defined to cover the implied aspects of rules and their combination with conceptual knowledge in the ontology. Concerning the expressiveness of the ontology language, we focus on the basic elements of OWL DL, which should make the work transferable to ontology languages other than OWL, and we mostly describe methods for the syntactic analysis of the considered ontology. We also focus on the basic features of rule languages such as SWRL and RIF-BLD: they correspond to a rule language of Horn clauses with class or property descriptions as literals with equality and a standard first-order semantics.

1.1. Verification at the Symbolic Level

Due to the combination of OWL and rules, however, the general detection of all anomalies is an undecidable task. Whereas for fragments of RIF-BLD or SWRL — such as ELP [19] — tractable reasoning can be provided, the identification of redundant and deficient knowledge still requires syntactic methods that investigate the concepts and rules at the symbolic level. Here, the term *verification* denotes the syntactic analysis of ontologies at the symbolic level for detecting anomalies. On the one hand, the discussed issues of the presented work originate from the evaluation of taxonomic structures in ontologies introduced by Gómez-Pérez [11]. On the other hand, in the context of rule ontologies, classical work on the verification of rule-based knowledge — see for instance Preece and Shinghal [22,23] — has to be reconsidered. In this work, the verification of ontologies (mostly taxonomies) and rules (based on predicate logic), respectively, has been investigated separately. However, the combination of taxonomic and other ontological knowledge with a rule extension leads to new evaluation issues that can cause redundant or even inconsistent behavior. For example, a very obvious redundancy may be due to the coexistence of the taxonomic relation `subClassOf(A, B)` and the rule $A \Rightarrow B$. One contribution of our work is the extension of classic measures by novel anomalies that result from the combination of rule-based and ontological knowledge. Here, the concept of dependency graphs from deductive databases can be used [8].

Of course, the collection of possible anomalies presented in this paper may always be incomplete, and ad-

ditional elements of the ontology language may also introduce new possibilities of occurring anomalies. For this reason, we propose the *declarative specification* of anomalies by DATALOG*, that allows for flexibly including new and application-relevant anomalies. Here, the axioms of the ontology and the given rules are mapped to corresponding DATALOG* facts and rules, respectively. Thus, the anomaly predicates described in the remainder of the paper can be directly applied.

In detail, we investigate the implications and problems that emerge from rule definitions in combination with some of the following ontological descriptions:

- (i) class relations like *subclass*, *complement*, *disjointness*,
- (ii) basic property characteristics like *transitivity*, *symmetry*, *ranges* and *domains*, and *cardinality restrictions*.

We distinguish between the following categories of anomalies:

- *Circularity* in taxonomies and rule definitions.
- *Redundancy* due to duplicate or subsuming knowledge.
- *Inconsistency* because of contradicting definitions.
- *Deficiency* comprising subtle issues describing questionable ontology design.

Since we mainly describe syntactic checks of ontologies, the presented work is different from the evaluation of ontologies with respect to the intended semantic meaning: the OntoClean methodology [14] is an example for semantic checks of taxonomic decisions made in ontologies. We also do not consider common errors that can be implemented due to the incorrect understanding of logical implications of OWL descriptions, as described by Rector et al. [25].

1.2. Integration of Verification Methods

In general, the verification of ontologies with rules should not be seen as an isolated task, but is understood as a subtask of the *evaluation phase*, that is proposed in almost all methodologies for ontology development [12]. In the past, a variety of methodologies was introduced, that structure the development and evolution process into distinct phases, for example the On-To-Knowledge methodology [32], METHONDOLOGY [12], and the extensive CommonKADS methodology [29]. Here, the presented verification methods can be integrated as a sub-task into the evaluation phase, and they are used after every significant modification of the working ontology.

1.3. Structure of the Paper

The paper is organized as follows: The next section gives basic definitions and describes the expressiveness of the underlying knowledge representation; in the context of this work a subset of OWL DL is used. Then, the four main classes of anomalies are discussed in more detail. In Section 3, we introduce anomalies concerning the circularity of definitions. Anomalies uncovering inconsistent knowledge are shown in Section 4. We deal with redundancy in Section 5 and describe deficient knowledge in Section 6. We present some technical details of the evaluation mechanism of DATALOG* in Section 7. The paper is concluded with a discussion.

2. Expressiveness and Basic Notions

For the analysis of ontologies with rules we restrict the considered constructs to a subset of OWL DL; in fact, many anomalies can occur when using the simple profile OWL 2 EL [13]: we investigate the implications of rules that are mixed with subclass relations and/or the property characteristics transitivity, symmetry, cardinality, complement, and disjointness.

For example, in a university domain there might exist classes like Person, Student, and Professor, that are connected by properties such as

- `subClassOf(Student, Person)`,
- `subClassOf(Professor, Person)`,
- `disjointClasses(Student, Professor)`.

Figure 1 shows a graphical version of the class definitions.

Fig. 1. A simple ontology example with a disjoint relation.

Class Atoms and Property Atoms. Given a class C and a property P : when used in rules we call $C(x)$ a class atom and $P(x, y)$ a property atom. For the following, it will be useful to extend the relations on classes and properties to relations on class and property atoms. Given two atoms A, A' , we write $\rho(A, A')$, if both atoms have the same argument tuple, and their predicate symbols are linked by a relation ρ , i.e., if A and A' both are

- class atoms, such that $A = C(x)$, $A' = C'(x)$, and $\rho(C, C')$, or
- property atoms, such that $A = P(x, y)$, $A' = P'(x, y)$, and $\rho(P, P')$.

E.g., the relation ρ can be `subClassOf`, `disjointClasses`, `objectComplementOf`, etc. From a relationship $\rho(A, A')$ it follows that A and A' are of the same type (either class or property atoms).

2.1. Specification in DATALOG*

The detection of anomalies has been done using the PROLOG meta-interpreter DATALOG*, which we have implemented in SWI PROLOG [36]. Due to their compactness and conciseness, we give the corresponding formal definitions in DATALOG* for the anomalies, which are evaluated using a mixed bottom-up/top-down approach based on DATALOG and PROLOG concepts, respectively.

An intuitive understanding of the presented, mixed rule sets is possible without fully understanding the new inference method. For the interested reader, we introduce technical details of the evaluation mechanism of DATALOG* as well as some supporting predicates in Section 7.

Variables such as $A, B, C, \dots, A'$, or B_i can be used for both class atoms and property atoms, whereas $As, Bs, \dots$ denote sets of class atoms and property atoms. The relationship `subClassOf(A, A')` describes that A is a subclass of A' .

Rules $B_1 \wedge \dots \wedge B_n \Rightarrow A$ are represented as non-grounded DATALOG* facts `rule(Bs=>A)` (with variable symbols), where $Bs = [B_1, \dots, B_n]$ is the list of body atoms, A is the head atom, and $\Rightarrow$ is a binary infix functor. Without loss of generality, we can assume that the rule heads are atomic, since rules with conjunctive rule heads can be split into several rules. In the bodies of DATALOG* and PROLOG rules, conjunction (and) is denoted by “,”; disjunction (or) is denoted by “;”, and negation by “not”.

2.1.1. Incompatible Classes: Disjointness and Complements

We use `disjointClasses(C1, C2)` to define the disjointness between two OWL classes. The construct `objectComplementOf` points to instances, that do not belong to a specified class. The disjointness relation between a class $C1$ and a class $C2$ is equivalent to the relation `subClassOf(C1, objectComplementOf(C2))`.

Two classes $C1$ and $C2$ are *incompatible*, if there exists a disjointness or a complement relationship between

them. This is described by the following PROLOG predicate used in DATALOG*:

```
incompatible(C1, C2) :-
  ( subClassOf( C1, objectComplementOf(C2) )
  ; disjointClasses(C1, C2) ).
```

For incompatible classes $C1$ and $C2$ there cannot exist an instance x with $C1(x) \wedge C2(x)$.

2.1.2. Taxonomic Relationships and Rules

An obvious equivalence exists between a transitive subclass relationship $\text{subClassOf}(B, A)$ — where A and B are both class atoms or both property atoms with the same arguments — and the rule $B \Rightarrow A$ with a single body atom B , that has the same argument as A . Thus, we combine them into the single formalism derives in DATALOG*:

```
derives(C1, C2) :-
  ( subClassOf(C1, C2)
  ; rule([B] => A),
    B =.. [C1, X1], A =.. [C2, X2],
    var(X1), X1 == X2 ).
```

The PROLOG call “ $T =.. Xs$ ” splits a term T into a list $Xs = [F, X1, \dots, Xn]$ consisting of the functor F and the arguments $X1, \dots, Xn$. Above, the functors $C1$ and $C2$ of the class atoms A and B , respectively, are class names and both atoms have one argument. The call “ $\text{var}(X1), X1 == X2$ ” tests, if these arguments $X1$ and $X2$ are bound to the same variable.

With the existence of equivalence definitions $E_1 \equiv E_2$ in an ontology language, e.g., the OWL definitions equivalentClasses and $\text{equivalentObjectProperties}$, we can further extend the definition of derives : an element $E1$ is derived by an element $E2$, if the elements are equivalent classes or properties. In DATALOG*, we extend derives with the following PROLOG rule:

```
derives(E1, E2) :-
  ( equivalentClasses(E1, E2)
  ; equivalentObjectProperties(E1, E2) ).
```

Since such an equivalence is symmetrical, the predicate derives always creates cyclic derivations of equivalent elements with length 1.

We compute the transitive closure tc_derives of derives using the following standard DATALOG* scheme:

```
tc_derives(E1, E2) :-
  derives(E1, E2).
tc_derives(E1, E3) :-
  derives(E1, E2), tc_derives(E2, E3).
```

The following PROLOG predicates with calls to DATALOG* facts generalize tc_derives and incompatible to atoms:

```
tc_derives_atom(A1, A2) :-
  A1 =.. [P1|Xs], A2 =.. [P2|Xs],
  tc_derives(P1, P2).

incompatible_atoms(A1, A2) :-
  A1 =.. [P1|Xs], A2 =.. [P2|Xs],
  incompatible(P1, P2).

tc_incompatible_atoms(A1, A2) :-
  A1 =.. [P1|Xs], A2 =.. [P2|Xs],
  tc_derives(P1, P3),
  incompatible(P3, P4),
  tc_derives(P2, P4).
```

As described before, the binary built-in predicate $=..$ of PROLOG splits given atoms A_i into their predicate symbol P_i and their list Xs of arguments; using the same variable Xs for both atoms requires the argument lists to be identical. We cannot evaluate these rules using forward chaining, since $=..$ cannot be applied if Xs is an unknown list.

For two (class or property) atoms A and B we say that A *implies* B , if $A = B$ or $\text{tc_derives_atom}(A, B)$. The first of the following supporting PROLOG rules turns the transitive closure into a reflexive transitive closure; the second extends it to negated atoms, i.e. literals, where $\sim$ denotes negation:

```
implies(A, B) :-
  ( A = B ; tc_derives_atom(A, B) ).
implies(~A, ~B) :-
  implies(B, A).
```

2.1.3. Remark on Examples

In the following we give examples for most of the described anomalies. Here, we use the benchmark university domain LUBM [15], because of its popularity and intuitive understanding. We use the prefixes a : and b : for classes and properties in order to paraphrase, that these elements are contained in the ontologies a and b .

3. Circularity

Circular definitions in the ontology have a severe impact on the reasoning capabilities of the underlying knowledge. Here we distinguish circular definitions in the taxonomic structure of the ontology as described by Gómez-Pérez [11], circular dependencies in the rule base as considered, e.g., by Preece and Shinghal [22], but also circular dependencies that can occur due to the mixture of taxonomic and rule-based knowledge.

3.1. Circularity in Taxonomy and Rules

Circular definitions can occur in the taxonomy, in rules, and in property relations.

3.1.1. Exact Circularity in Taxonomy and Rules

The following DATALOG^{*} predicate finds pairs $[E, F]$ of subsequent elements $E = E_{i-1}$ and $F = E_i$ from a cyclic chain

$$E_1, E_2, \dots, E_n, E_{n+1} = E_1,$$

where E_{i-1} derives E_i , for $2 \leq i \leq n+1$, such that all elements E_i of the chain are either classes or properties.

```
anomaly(exact_circularity, [E, F]) :-
  derives(E, F), E \= F,
  tc_derives(F, E).
```

Cycles with $n = 1$ commonly occur due to the inclusion of equivalent classes and properties in the predicate derives. For the subClassOf relation alone (included in derives), the described circular relationships are commonly detected by existing tools.

Example: Consider two ontologies a and b with classes – subClassOf(a:Professor, a:Person) and – subClassOf(b:Employee, b:Person).

Then, the following incorrect alignments create an undesired circularity in the taxonomy with $n = 4$:

- equivalentClasses(a:Professor, b:Person) and
- equivalentClasses(a:Person, b:Employee).

The example is depicted in Figure 2, where the incorrect alignment between the concepts of two ontologies a and b produce the circular dependence.

3.1.2. Circularity between Rules and Taxonomy

A rule $B_1 \wedge \dots \wedge B_n \Rightarrow A$, such that the head atom A implies some body atom $B = B_i$, leads to a cir-

Fig. 2. An example of a circular alignment of concepts of two different ontologies due to the incorrect use of equivalence relations.

cularity. The rule should be considered as a restricted subClassOf relation between A and B , which may result in the detection of a misapplied taxonomic definition between them. The circularity can be found with by following DATALOG^{*} predicate:

```
anomaly(circularity, A-Bs) :-
  rule(Bs=>A), member(B, Bs),
  implies(A, B).
```

Example: Since subClassOf(Professor, Person), the following rule — defining a specific restriction on instances of classes Person and Professor — creates a partially cyclic definition:

```
Person(X) ∧ Teacher(X, Y) ∧
University(Y) ⇒ Professor(X).
```

3.2. Circular Properties

Property descriptions can also be the source of circularity, when a chain of properties P_i connects a class C by a chain

$$C = C_1 \xrightarrow{P_1} C_2 \xrightarrow{P_2} \dots \xrightarrow{P_{n-1}} C_n = C$$

of classes C_i , with $n \geq 2$, to itself; at least one of the properties should not be symmetric. We say that a property P connects two classes D and E and denote this by $D \xrightarrow{P} E$, if there exists a property between two classes D' and E' , such that D transitively derives or is equal to D' and E' transitively derives or is equal to E .

Often such a circularity leads to infinite models of the ontology. In pure description logic reasoners, various blocking methods [2,18] ensure termination of the proof procedure in case of existentially quantified cycles. However, the extension of ontologies by rules requires new methods, and decidability is not guaranteed in the general case. Typical sources of circularity are the incorrect use of inverse and symmetrical properties during the matching of two ontologies. In the general

case however, a cyclic property chain may sometimes be an intentional design decision in ontology modeling and should be therefore not treated as an anomaly.

Example: We consider two ontologies a and b with the following classes and alignments: `equivalentClasses(a:Lecture, b:Course)` and `equivalentClasses(a:Professor, b:Professor)`. The following further properties are defined in the ontologies:

- `lectures(a:Professor, a:Lecture)` and
- `teaches(b:Professor, b:Course)`.

If `lectures` and `teaches` are incorrectly aligned as inverse properties, then a property cycle is created.

We consider common property and range restrictions and further restrictions like the quantifiers `someValuesFrom` and `allValuesFrom`. Circular properties are detected in `DATALOG*` as follows.

```
anomaly(circular_property, C, Ps) :-
  tc_connected_classes(C, Ps, C),
  member(P, Ps),
  not(symmetricObjectProperty(P)).
```

The call `anomaly(circular_property, C, Ps)` computes classes `C` that are connected to themselves by a chain `Ps` of properties; the chain `Ps` is computed by using the `DATALOG*` predicate `tc_connected_classes`, which will be given in Section 7. If at least one of the properties is not symmetric, then we have found a circular chain.

4. Inconsistency

Contradictory knowledge contained in ontological knowledge and rules often yields unintended and unexpected conclusions. In the past, possible inconsistencies were investigated separately for both taxonomic knowledge [11] and rule-based knowledge [22]. In the context of this paper, we focus on inconsistent knowledge that can be detected at the symbolic level. In the common case, the consistency of ontological knowledge with (general) rules cannot be derived in a tractable manner.

Typical examples of inconsistencies are *contradicting rule* consequences for two rules with subsuming rule antecedents. For taxonomic knowledge, the *partition error*, which is given by a subclass of two or more classes that are contained in a disjoint partition (pairwise disjoint classes), is very common. In the following, we additionally discuss inconsistencies that may occur due to the combined use of rules and ontology definitions.

4.1. Partition Error in Taxonomy

The partition error [10] is commonly created due to the incorrect combination of disjoint and derives relations: There exists a partition error on the class level, when a class C is the subclass of two disjoint classes C_i, C_j . Similarly, a partition error on the instance level occurs, when an instance X was created from two disjoint classes.

Example: Consider the ontology a with a class `Person` having two disjoint subclasses `Teacher` and `Student`:

- `subClassOf(a:Teacher, a:Person)`,
- `subClassOf(a:Student, a:Person)`, and
- `disjointClasses(a:Teacher, a:Student)`.

The alignment of the class `b:TA` (`TeachingAssistant`) of the ontology b as a subclass of both `a:Teacher` and `a:Student` would introduce a partition error, see for example Figure 3.

The following `DATALOG*` predicate detects partition errors, where X is either a subclass or an instance of the disjoint classes $C1$ and $C2$. The Prolog term `X-[C1,C2]` is used as a syntactic data structure for X and the group of the disjoint classes $C1$ and $C2$.

```
anomaly(partition_error, X-[C1, C2]) :-
  incompatible(C1, C2),
  ( ( derives(X, C1), derives(X, C2) )
  ; ( classAssertion(C1, X),
      classAssertion(C2, X) ) ).
```

Fig. 3. An example of a partition error, where the concept TA (`Teaching Assistant`) inherits from the disjoint concepts `Teacher` and `Student`.

Since OWL 2, it is also possible to define disjointness between properties, asserting that a given collection of properties is pairwise exclusive. A partition error for properties can be defined analogously to the DATALOG* predicate given above.

4.2. Incompatible Rule Antecedent

A rule $B_1 \wedge \dots \wedge B_n \Rightarrow A$ has an incompatible antecedent, if there exists an incompatibility relationship between two body atoms B_i and B_j , e.g., a disjoint or complement relationship. Note that, according to our definitions this means that $B_i = C_i(x)$ and $B_j = C_j(x)$ are class atoms with the same argument x , and that C_i and C_j are disjoint or complements.

Example: Consider the ontology *a* with two disjoint classes *Teacher* and *Student*, i.e., `disjointClasses(a:Teacher, a:Student)` is defined. The following alignment rule would introduce an incompatible rule antecedent:

`a:Student(X) ∧ a:Teacher(X) ⇒ b:TA(X)`

The example is similar to the partition error shown in Figure 3, where the contradicting concept is inherited by two disjoint concepts. In this example, the contradicting concept is derived by a rule having incompatible concepts in the antecedent.

The following DATALOG* predicate detects incompatible rule antecedents.

```
anomaly(incompatible_antecedent, Bs=>A) :-
  rule(Bs=>A), sub_sequence([Bi, Bj], Bs),
  tc_incompatible_atoms(Bi, Bj).
```

The basic PROLOG predicate `sub_sequence` selects a sub-sequence of (not necessarily consecutive) elements of a given list. Note, that the call `tc_incompatible_atoms` instantiates the rule `Bs=>A`.

An incompatible rule antecedent can be also considered to be a redundancy, since it is responsible for an unused rule, that never fires. However, we classify this anomaly as an inconsistency, because the incompatible antecedent may very likely be the result of a defective alignment of classes.

4.3. Self-Contradicting Rule

An anomaly similar to the incompatible rule antecedent is described by the following: A rule is called *self-contradicting*, if there exists an incompatibility relationship between the head atom A and at least

one body atom B , i.e., A and B are disjoint or complements. Note that, according to our definitions this means that $A = C(x)$ and $B = D(x)$ are class atoms with the same argument x , and that C and D are disjoint or complements.

Example: For two ontologies, the relationship `disjointClasses(a:Teacher, b:Student)` can be responsible for creating a self-contradicting rule, for example `b:Student(X) ∧ b:Lecture(Y) ∧ b:teaches(X,Y) ⇒ a:Teacher(X)`.

The following DATALOG* predicate derives instances of rules that are self-contradicting:

```
anomaly(self_contradicting_rule, Bs=>A) :-
  rule(Bs=>A), member(B, Bs),
  tc_incompatible_atoms(A, B).
```

If a self-contradicting rule is activated, then the derived consequent contradicts its antecedent.

4.4. Contradicting Rules

Consider two instances r and r' of rules, such that for every body atom B of r there exists a body atom B' of r' , such that B' implies B . The rules r and r' are contradicting, if their head atoms A and A' are contradicting. If r' would fire, then also the more general rule r would fire, which derives contradicting conclusions.

Example: For two ontologies *a* and *b*, the incorrect equivalence relationship between the properties *a:lectures* and *b:inLecture* will cause the following rules to be contradicting:

- `a:Per(X) ∧ a:Lec(Y) ∧ a:lectures(X,Y) ⇒ a:Teacher(X),`
- `b:Per(X) ∧ b:Lec(Y) ∧ b:inLecture(X,Y) ⇒ b:Student(X),`

where *Person* and *Lecture* are abbreviated by *Per* and *Lec*, respectively, with the relationship `disjointClasses(a:Teacher, b:Student)` and the equivalence relationships between the corresponding classes *Person* and *Lecture* in the ontologies *a* and *b*. The example is depicted in Figure 4.

Contradicting rule instances can be detected based on a suitable subsumption relation $\supseteq$ for clauses, which we will also use for detecting rule subsumption in a later section. Therefore, we extend the relation *implies* from atoms to negative literals: $\neg A$ implies $\neg B$, if B implies A . Moreover, we call the disjunction $\alpha = \neg B_1 \vee \dots \vee \neg B_n$ the body clause of a rule

Fig. 4. An example of the incorrect alignment of the properties lectures and inLecture resulting in contradicting rules.

$$r = B_1 \wedge \dots \wedge B_n \Rightarrow A.$$

Definition 1 (Subsumption) Given two disjunctions $\alpha = L_1 \vee \dots \vee L_n$ and $\beta = K_1 \vee \dots \vee K_m$ of arbitrary (positive or negative) literals L_i and K_j . $\alpha \supseteq \beta$, if there exists a substitution θ , such that for all L_i there exists a K_j , where $L_i\theta$ implies K_j .

In comparison, the standard subsumption relation would require $L_i\theta = K_j$ instead of $L_i\theta$ implies K_j .

The following DATALOG* rule derives instances of rules that contradict each other:

```
anomaly(contradicting_rules, [R1, R2]) :-
  rule(R1), rule(R2),
  contradicting_rules(R1, R2).
```

This DATALOG* rule is supported by the following PROLOG rule:

```
contradicting_rules(Bs1=>A1, Bs2=>A2) :-
  negate_atoms(Bs1, Cs1),
  negate_atoms(Bs2, Cs2),
  clause_subsumes(Cs1, Cs2),
  tc_incompatible_atoms(A1, A2).
```

The PROLOG predicate `negate_atoms` transforms a list $[B_1, \dots, B_n]$ of atoms into a list $[\sim B_1, \dots, \sim B_n]$ of negated atoms. The rule is further supported by the following PROLOG predicates `clause_subsumes`:

```
clause_subsumes(Cs1, Cs2) :-
  checklist(implies, Cs1, Cs2).
```

In case of subsumption, the body of the more general rule r_1 always fires when the body of r_2 fires. Note that, based on the call of the PROLOG predicate `clause_subsumes`, the predicate above computes in-

stances of r_1 and r_2 , such that the body of the instance of r_1 subsumes the body of the instance of r_2 .¹ The consequences $A_1 = C_1(x)$ and $A_2 = C_2(x)$ are contradicting, if the corresponding classes C_1 and C_2 are incompatible, i.e., disjoint or complements.

The described anomaly can be generalized to two (not necessarily disjoint) sets of rules that derive two semantically contradicting conclusions. However, this generalized type of anomaly cannot be detected in a purely syntactic manner.

4.5. Multiple Functional Properties

Functional properties are not allowed to have more than one value for each individual. Therefore, the functional definition of a property can be canonically restricted to a property with a minimum cardinality restriction ≥ 0 and a maximum cardinality restriction ≤ 1 .

For this reason, we can easily detect a semantic error, if a functional property has a maximum cardinality restriction greater than 1. Please note, that a property which transitively derives a functional property is also functional.

The detection of an inconsistently defined maximum cardinality restriction can be done in DATALOG* as follows:

```
anomaly(multiple_functionality, Q) :-
  functionalObjectProperty(Q),
  ( P = Q ; tc_derives(P, Q) ),
  max_cardinality_restriction(C, P, X), X > 1.
```

With the introduction of OWL 2, also qualified cardinality restrictions are allowed; thus, an additional predicate can be introduced to check, if particular instances of a property exceed a corresponding qualified restriction. An inconsistent property restriction may be the result of an incorrectly performed ontology integration, e.g., the wrong alignment of functional and non-functional properties.

5. Redundancy

Redundant knowledge is created by ontological definitions and rules that can be removed from the knowledge base without changing the intended semantics. In

¹ If we would replace the call to the PROLOG goal $G = \text{contradicting_rules}(R1, R2)$ by $\text{not}(\text{not}(G))$ in the body of the anomaly rule above, then we would check for subsumption without creating instances.

most cases, redundancies can be clearly identified. Typical redundancies for ontologies like identical concepts have already been discussed, for example in [11]. Also, a separate discussion of rule-based redundancies like subsuming rules can be found for instance in [23].

In the following, we introduce further redundancies that can occur due to the combination of ontological definitions and rules.

5.1. Identity

We call identical formal definitions of classes, properties or rules, that can be only discriminated by their different names, *identity errors*. They can occur if some implied knowledge is not explicitly stated in the ontology, thus uncovering an incompleteness error.

For example, identical classes may be distinguished by the developer by the introduction of an additional property for one of the identical classes. Also identity of classes or rules can be created by the integration of overlapping ontologies that share (partially) identical concepts.

5.2. Redundancy by Repetitive Taxonomic Definition

The redundant definition of taxonomic knowledge of classes and properties was already described by Gómez-Pérez [11]. Let X, Y be either two classes or two properties. We distinguish two types of repetition:

- *direct repetition*, where $\text{subClassOf}(X, Y)$ is defined more than once in the ontology;
- *indirect repetition*, where $\text{subClassOf}(X, Y)$ is defined, but this relation can be also derived by a chain $\text{subClassOf}(X, X_1), \text{subClassOf}(X_1, X_2), \dots, \text{subClassOf}(X_n, Y)$ with $n \geq 1$.

Direct and indirect repetition corresponding to the instantiation of classes and properties can be also defined on instance-of instead of subclass relations. A repetitive definition can easily occur due to the (correct) alignment of two classes or properties. In such cases, repetitions are not an undesirable redundancy, but an intended behavior.

5.3. Rule Subsumption

A rule $r = B_1 \wedge \dots \wedge B_n \Rightarrow A$, can be mapped to a logically equivalent disjunction $\text{clause}(r) = \neg B_1 \vee \dots \vee \neg B_n \vee A$.

We say that a rule r subsumes another rule r' , for short $r \supseteq r'$, if $\text{clause}(r) \supseteq \text{clause}(r')$. This means, that the head A of r subsumes the head A' of r' , and the

body clause of r subsumes the body clause of r' with respect to the same substitution θ .

A subsumed rule r' can be removed without changing the semantics of the ontology. Subsuming rules can be detected by the following DATALOG* predicate, where the PROLOG predicate $\text{rule_subsumes_check}$, which we do not list here, is used for checking subsumption:

```
anomaly(subsumed_rule, [R1, R2]) :-
    rule(R1), rule(R2),
    rule_subsumes_check(R1, R2),
    not(rule_subsumes_check(R2, R1)).
```

5.4. Redundant Implication

A rule r (over class or property atoms) has a *redundant implication* of a parent, if some body atom B implies the head atom A . This can be seen as a special case of rule subsumption, since the implication can be seen as a rule $B \Rightarrow A$, which subsumes the rule r .

Example: Given the subclass relation $\text{subClassOf}(\text{Professor}, \text{Teacher})$, the following rule redundantly derives the parent Teacher:

$\text{Professor}(X) \wedge \text{Lecture}(Y) \wedge \text{teaches}(X, Y) \Rightarrow \text{Teacher}(X).$

The example is depicted in Figure 5.

Fig. 5. An example for a rule redundantly deriving an already known parent.

In DATALOG*, such a redundancy can be defined as follows:

```
anomaly(implication_of_superclass, Bs=>A) :-
    rule(Bs=>A), member(B, Bs),
    implies(B, A).
```

Besides the obviously redundant inclusion of B in the antecedent, this anomaly might also point to an incorrectly assigned subsumption relation between A and B . On the one hand, there exists a separate subsumption

between A and B. On the other hand, the rule defines an additional restricted subsumption if the rule body not only contains B but further atoms. Therefore, the anomaly may also point to an inconsistent mapping between A and (an ancestor of) B. For $B \equiv A$, the equivalence may be incorrectly assigned, since the rule condition denotes a restriction on the implication. This error is similar to *circularity between rules and taxonomy*, but with an inverse subclass relation.

With the introduction of Property Chain Inclusion (ObjectPropertyChain) in OWL 2, it similarly becomes possible to redundantly derive a property chain by a rule. For instance, the rule `worksFor(Person, Lab) ∧ locatedIn(Lab, Org) ∧ ... ⇒ worksFor(Person, Org)` describes a redundant implication, if the following property chain was already defined: `ObjectPropertyChain(worksFor locatedIn) worksFor`. When the rule contains additional atoms in the rule body, the detection of this anomaly points to an incorrectly defined `ObjectPropertyChain`, since the additional atoms may define a more restricted constraint on the particular inclusion.

5.5. Redundant Implication of Transitivity or Symmetry

The following two anomalies can be interpreted as special cases of a rule subsumption.

Transitivity. A rule of the form

$$r = P(x, y) \wedge Q(y, z) \wedge \beta \Rightarrow R(x, z)$$

with a transitive property R in the head is redundant, if the properties P, Q , and R are equivalent. The reason is that in this situation the more general rule $r_t = P(x, y) \wedge Q(y, z) \Rightarrow R(x, z)$ without β can be derived by the OWL reasoner. We always assume a property P to be equivalent to itself.

Example: For a transitive property `sub`, which should abbreviate `subOrganizationOf`, the following rule redundantly repeats the transitive definition:

$$\text{sub}(X, Y) \wedge \text{sub}(Y, Z) \wedge \dots \Rightarrow \text{sub}(X, Z).$$

A redundant definition of a transitive property can be detected using the following DATALOG* predicate:

```
anomaly(redundant_transitivity_hb, Rule) :-
  rule(Rule),
  head_predicate(Rule, R),
  rule_transitivity(Rule, R, Rule_t),
  rule_subsumes_check(Rule_t, Rule).
```

If `Rule` is the DATALOG* representation of an arbitrary rule r with the head predicate R , then the supporting PROLOG predicate `rule_transitivity/3`, cf. Section 7, tests if R is transitive and constructs the DATALOG* representation `Rule_t` of the rule $r_t = P(x, y) \wedge Q(y, z) \Rightarrow R(x, z)$, such that P, Q , and R are equivalent.

Then, we can check if r_t subsumes r ; this depends on the arguments of the predicates P, Q , and R in r . E.g., r_t subsumes the rule r from above, but it does not subsume the rule $r' = P(x, y) \wedge Q(y, z) \wedge \beta \Rightarrow R(z, x)$.

Symmetry. An analogous anomaly can occur for symmetrical properties R in rule heads: if R is equivalent to the property P , then the rule

$$r = P(x, y) \wedge \beta \Rightarrow R(y, x)$$

is redundant, since the more general rule $r_s = P(x, y) \Rightarrow R(y, x)$ without β can be derived by the OWL reasoner. In DATALOG* we detect such a redundancy as follows:

```
anomaly(redundant_symmetry_hb, Rule) :-
  rule(Rule),
  head_predicate(Rule, R),
  rule_symmetry(Rule, R, Rule_s),
  rule_subsumes_check(Rule_s, Rule).
```

If `Rule` is the DATALOG* representation of an arbitrary rule r with the head predicate R , then the supporting PROLOG predicate `rule_symmetry/3`, cf. Section 7, tests if R is transitive and constructs the DATALOG* representation `Rule_s` of the rule $r_s = P(x, y) \Rightarrow R(y, x)$, such that P and R are equivalent. Then, we can check if r_s subsumes r , which again depends on the arguments of the predicates P and R in r .

Often such redundancies can be explained by an erroneous assumption of the transitivity or symmetry during an ontology matching process. Then, the rules define a more restrictive condition of transitivity and symmetry, respectively, if the conjunctions β are not empty. For this reason, the anomalies may be either classified as inconsistent mappings of the properties, or as incorrect alignments of transitivity and symmetry.

5.6. Redundancy in the Antecedent of a Rule

Redundancy in the antecedent may occur because of redundant derivations of classes or properties, or because of already defined property relations.

5.6.1. Redundant Derivation in the Antecedent

A redundancy in the antecedent of a rule occurs in a rule $B_1 \wedge \dots \wedge B_n \Rightarrow A$, if some body atom B_i implies another body atom B_j . Here, B_j is redundant in the rule body and may be removed.

Example: The subclass relationship `subClassOf(TeachingAssistant, Person)` makes the atom `Person(X)` redundant in the following rule:

```
Person(X) ∧ TeachingAssistant(X) ∧ ...
⇒ Employee(X)
```

The DATALOG* implementation for finding the anomaly is as follows:

```
anomaly(redundant_derivation, Bs=>A) :-
  rule(Bs=>A), sub_sequence([Bi, Bj], Bs),
  ( implies(Bi, Bj)
  ; implies(Bj, Bi) ).
```

As a special case, this form of redundancy can occur in the ontology, if $B_i \equiv B_j$, e.g., due to the definition of equivalence relations. The anomaly may alternatively point to an incorrect mapping between the elements B_i and B_j , when these two elements were aligned from different ontologies.

5.6.2. Redundant Use of Transitivity and Symmetry

With the definition of special property characteristics in OWL, further anomalies may occur. For equivalent properties P, Q, R , there may exist the following redundancies:

- A rule $P(x, y) \wedge Q(y, z) \wedge R(x, z) \wedge \beta \Rightarrow A$ has a redundant body atom $R(x, z)$, if the properties P, Q, R , are *transitive*.
- A rule $P(x, y) \wedge Q(y, x) \wedge \beta \Rightarrow A$ has a redundant body atom $Q(x, y)$, if the properties P and Q are *equivalent* and *symmetric*.

In DATALOG* – with a supporting PROLOG rule – this can be detected using the PROLOG predicate `clause_subsumes_check`:

```
anomaly(redundant_transitivity_b, Rule) :-
  rule(Rule),
  body_predicate(Rule, R),
  rule_transitivity(
    Rule, R, [Pxy, Qyz]=>Rxz),
  rule_to_clause(Rule, [_|Cs]),
  clause_subsumes_check(
    [~Pxy, ~Qyz, ~Rxz], Cs).
```

We first construct three atoms Rxz , Pxy , and Qyz for equivalent properties, where R is a transitive property that occurs in the body of a rule *Rule* together with P and Q . Then, we form a clause from the negations of the three atoms and check if it subsumes the body clause Cs of *Rule*. The body clause Cs is obtained by applying the predicate `rule_to_clause` and omitting the first element of the result, which is the head of *Rule*.

```
anomaly(redundant_symmetry_b, Rule) :-
  rule(Rule),
  body_predicate(Rule, Q),
  rule_symmetry(Rule, Q, [Pxy]=>Qyx),
  rule_to_clause(Rule, [_|Cs]),
  clause_subsumes_check([~Pxy, ~Qyx], Cs).
```

We construct two atoms Pxy and Qyx for equivalent properties, where P is a symmetric property that occurs in the body of a rule *Rule* together with Q . Then, we form a clause from the negations of the two atoms, and we check if it subsumes the body clause Cs of *Rule*.

Example: For two ontologies *a* and *b*, the symmetric properties

- `a:worksWith(a:Person, a:Person)` and
- `b:collaborates(b:Person, b:Employee)`

were defined to be equivalent. With the alignment `equivalentClasses(a:Person, b:Person)` and the relationship `subClassOf(b:Employee, b:Person)`, the rule $a:P(X) \wedge a:worksWith(X, Y) \wedge b:collaborates(Y, X) \Rightarrow b:E(Y)$, where *Person* and *Employee* are abbreviated by *P* and *E*, respectively, redundantly includes one of the two symmetric properties; either the use of `worksWith` or `collaborates` is redundant. In Figure 6 the concepts and properties together with their alignments are shown.

Fig. 6. The rule redundantly uses a symmetrical property: $a:P(X) \wedge a:worksWith(X, Y) \wedge b:collaborates(Y, X) \Rightarrow b:E(Y)$

Like the redundant definitions of transitivity and symmetry as described in Section 5.5, these anomalies *can*

point to an incorrect mapping of properties during an ontology alignment process.

5.7. Unsupported Rule Condition

A rule r has an unsupported condition, if at least one of its body atoms B neither unifies with an input atom (e.g., a given instantiation of the ontological concepts) nor with the consequent of another rule. The corresponding DATALOG* predicate is shown below:

```
anomaly(unsupported_condition, Bs=>A) :-
  rule(Bs=>A), member(B, Bs),
  not(call(B)), not(rule(_=>B)).
```

The rule even checks if some call of the atom B is successful.

5.8. Unsatisfiable Rule Condition

An unsatisfiable condition can occur due to the rich semantics of OWL, for instance, if complement or disjoint descriptions are incorrectly aligned. The rule antecedent is unsatisfiable, if two body literals $B_i = C_i(x)$ and $B_j = C_j(x)$ are incompatible.

The definition of an unsatisfiable condition is given in DATALOG* as follows:

```
anomaly(unsatisfiable_condition, Bs=>A) :-
  rule(Bs=>A), sub_sequence([Bi, Bj], Bs),
  tc_incompatible_atoms(Bi, Bj).
```

The anomaly was also described as the inconsistency *incompatible rule antecedent* in Section 4.2, because the occurrence of such a rule in a (merged) ontology may also point to an incorrect alignment of a disjoint or complement description.

5.9. Redundant Cardinalities

When using properties to define relations between classes, the relation can be further specialized by cardinality restrictions. However, sometimes the cardinalities are redundant due to the semantics of some special properties in OWL. One example is the use of the minimal cardinality ≥ 0 , since all instances of a property have a link to zero or more individuals in its domain definition. The detection of a redundant min-cardinality restriction can simply be done using the following DATALOG* predicate:

```
anomaly(redundant_mincardinality_0, Q) :-
  min_cardinality_restriction(C, P, 0).
```

Example: The property `teaches(Person, Person)` defines a redundant cardinality restriction, that can be omitted.

```
<owl:Restriction>
  <owl:onProperty rdf:resource='#teaches' />
  <owl:minCardinality
    rdf:datatype='&xsd;nonNegativeInteger'>
    0 </owl:minCardinality>
</owl:Restriction>
```

Another example for redundant cardinality restrictions is a max-cardinality restriction ≤ 1 for functional properties. If a super-property of the property is functional, then the cardinality restriction is also redundant.

Example: For a property `hasID(Organization, &xsd:string)`, a max-cardinality restriction with ≤ 1 is defined. If the property is also functional, then the restriction can be omitted.

```
<owl:Restriction>
  <owl:onProperty rdf:resource='#hasID' />
  <owl:maxCardinality
    rdf:datatype='&xsd;nonNegativeInteger'>
    1 </owl:maxCardinality>
</owl:Restriction>
```

The restriction is redundant, because the functionality of a property requires its uniqueness for the entire ontology.

The following DATALOG* predicate detects redundant max-cardinality restrictions:

```
anomaly(subsumed_maxcardinality_1, Q) :-
  functionalObjectProperty(Q),
  ( P = Q ; tc_derives(P, Q) ),
  max_cardinality_restriction(C, P, 1).
```

Since a functionality definition is intuitively well-defined, this concept should be preferred when compared to a max-cardinality restriction.

6. Deficiency

Deficiency is more subtle than the previously presented categories of anomalies. The following anoma-

lies consider the completeness, understandability and maintainability of ontologies. Possible sources of such anomalies are imprecision during the manual development of (large) ontologies, effects of the evolution of ontologies, e.g., [31], and erroneous side-effects of the integration of ontologies.

Since deficiencies mostly detect areas in an ontology with problematic design, we also call them *design anomalies*. The identification of a such an anomaly is the starting point of a *refactoring*. Refactoring methods describe procedures to eliminate the corresponding design anomaly without changing the meaning of the remaining knowledge.

Originally, design anomalies had been identified and investigated for relational databases. In the last years, software engineering research has coined the term *bad smells* for parts of the source code that do not produce false behavior, but are badly designed and should be improved for better maintainability, cf. [21,9]. Recently, some approaches were presented to transfer this idea to the conceptual properties of different knowledge representations [3,6] and OWL ontologies [4,5].

In the following, we present a set of possible anomalies that affect the design of an ontology. However, these can be only seen as indicators for an actual anomaly. In any case the user has to decide whether and how to remove the possible issue. The presented design anomalies mainly focus on the detection of badly designed ontology concepts. For some anomalies their use in rules is taken into account, whereas other anomalies can occur independent of the existence of rule-based knowledge.

6.1. Lazy Class/Property

The usage of a class or property is often a good indicator for its actual utility. We call a class or a property of an ontology *lazy*, if it is never or rarely used in real-world applications. More precisely, an element is possibly lazy when

- the element represents a leaf in the hierarchy, and
- no rules use this element, and
- there exist no instances of the element, and
- no other element uses this element as a property.

There exist a number of reasons for the occurrence of this anomaly: Lazy elements may occur due to the integration of ontologies (including terms that are not useful or relevant any more), or due to the evolution of an ontology (previously useful concepts were replaced by specializations or generalizations).

In DATALOG*, a possibly lazy element E can be detected as follows:

```
anomaly(lazy_element, E) :-
    element(E),
    not(subClassOf(_, E)),
    not(rule_predicate(E)),
    not(instance(_, E)).
```

The supporting predicate `element` is defined in PROLOG:

```
element(E) :-
    ( class(E)
    ; objectProperty(E)
    ; datatypeProperty(E)
    ; transitiveObjectProperty(E)
    ; symmetricObjectProperty(E)
    ; functionalObjectProperty(E)
    ; inverseObjectProperties(E, _) ).
```

The constraints stated above can be relaxed by checking for very few rules with the considered element in their head or body. Then, these rules have to be inspected by the user and marked as *not useful*.

6.2. Chains of Inheritance

The hierarchy of classes and properties define the backbone of every ontology. Simple subclass relations are used to describe the inheritance of concepts and property relations. During the evolution of (manually built) ontologies or due to the imprecise integration of ontologies, the intended subclass structure of classes and properties can degenerate to subclass cascades in some parts of the hierarchy.

A taxonomic chain

$$C_1, C_2, \dots, C_n$$

of pairwise different classes C_i , where C_{i-1} is a subclass of C_i , for $2 \leq i \leq n$, is called a *chain of inheritance*, if all intermediate classes $C_2, \dots, C_{n-1}$ are not participating in any other subclass relations except the ones in the chain (isolated `subClassOf`), see Figure 7. The intermediate elements C_i may be not useful for applications, when

- (i) there exist no or very few individuals for the elements C_i and
- (ii) the elements C_i are not (extensively) used in ontological definitions, e.g., restrictions, or in rules.

Fig. 7. A chain of inheritance of classes $C_1, \dots, C_n$.

A maximal chain of inheritance can be detected in DATALOG* as follows.

```

anomaly(chain_of_inheritance, Cs) :-
    maximal_simple_path(
        isolated_subClassOf, Cs).

isolated_subClassOf(C1, C2) :-
    subClassOf(C1, C2),
    not( ( subClassOf(C1, C), C \= C2 ) ),
    not( ( subClassOf(C, C2), C \= C1 ) ).

```

The PROLOG predicate `maximal_simple_path` from a graph library computes simple paths, which at both ends cannot be extended by an `isolated_subClassOf`.

If the user has decided to eliminate the useless elements of the chain, then the particular elements have to be removed separately by the refactoring *collapse hierarchy* [5].

6.3. Lonely Disjoint Class

We call a class a *lonely disjoint*, if this class is not disjoint with any of its siblings, but it is disjoint with classes that are mutual siblings in another branch of the taxonomy. See Figure 8 for an example, where class C is a lonely disjoint, since C is disjoint to the classes $C_1, \dots, C_n$, that are siblings but not a sibling of C .

It is worth noticing that the class C does not need to be disjoint to all C_i in the disjoint partition, but to a sufficiently large number of classes C_i (in Figure 8, class C_{n+1} is not disjoint with C).

Fig. 8. Lonely disjoint – a distant class C disjoint to a collection of siblings $C_1, \dots, C_n$.

The DATALOG* implementation of this anomaly is as follows:

```

anomaly(lonely_disjoint, C) :-
    class(C), siblings(Cs),
    checklist(disjointClasses(C), Cs),
    not((sibling(C, M),
        disjointClasses(C, M))).

```

The PROLOG meta-predicate `checklist/2` calls `disjointClasses(C, D)` for all members D of the list Cs to determine, whether C and D (the C is always the same) are disjoint. The rule is supported by two PROLOG rules for `sibling` and `siblings`, respectively, including aggregation, which we will see in Section 7.

A lonely disjoint class is often created by the manual modification of the ontology: a class is moved to another branch of the taxonomy, but the attached disjointness descriptions are not re-aligned appropriately. Furthermore, the anomaly can also occur due to incorrect alignments during an ontology integration task. The existence of a lonely disjoint class can cause unintended reasoning results or even errors. The developer of the ontology has to decide manually about detected lonely disjointness. The elimination of an actual anomaly is quite simple: the disjointness property is just removed from the lonely disjoint class.

6.4. Over-Specific Property Range

Sometimes the developers of an ontology tend to be very specific when defining value ranges for properties. During the practical use of the ontology, it often turns out that the values are too specific and that a coarser range with less values would fit the considered domain much better.

Example: The value range $R_{temp} = \{very\ high, high, normal, low, very\ low\}$ of a property *temp* (for temperature) may contain five possible values, but the actual application of the property uncovers that a more general range $R'_{temp} = \{high', normal, low'\}$ with three values would work much better. A typical example for this situation is the alignment of two ontologies, where the value range of a specific concept is shrunk in order to match with a foreign concept. A further example is the planned use of the developed system by human operators: here, a smaller and more comprehensible range of values is less prone to errors caused by manual data entry.

If rules are defined containing this property, then the anomaly can be identified by the existence of many analogous rules for the particular values. In our example, rules for the values *high* and *very high* could be present. In such cases, the refactoring *coarsen value range* [5] forms groups of equivalent values, e.g., $high' = \{high, very\ high\}$ and $low' = \{low, very\ low\}$.

The following DATALOG* predicate detects over-specific property ranges by determining pairs of rules that have variants *has_value*(P, Vi) in their antecedent (with $i = 1, 2$). This type of rule pair is found by deleting the variants from the rule bodies and subsequently testing for their equality.

```
anomaly(over_specific,
  [R1, R2, has_value(P, [V1, V2])]) :-
  rule(R1), rule(R2),
  R1 = Bs1=>A, R2 = Bs2=>A, R1 \= R2,
  delete(has_value(P, V1), Bs1, Bs),
  delete(has_value(P, V2), Bs2, Bs).
```

An analogous DATALOG* predicate can be defined for over-specific property ranges in rule consequences. The anomaly is removed by replacing the original values of the property with the aggregated ones in every ontological definition, e.g., in restrictions or rules. A detailed example of this refactoring is shown in [5].

It is worth noticing that the elimination of an over-specific property range can introduce new redundancies or even inconsistencies. For example, redundant rules

are commonly produced when there exist equivalent rules with values that were coarsened to a single value. Inconsistent rules can occur due to a semantically inconsistent mapping function. In consequence, after the elimination of an anomaly, it is reasonable to undergo a subsequent check for redundant or inconsistent definitions.

6.5. Property Clump

In ontologies, properties are commonly used to define relations and attributes between classes and individuals, respectively. The repeated and identical use of a collection of properties in many classes is a deficiency called *property clump*. A property clump is comparable to the repeated use of code fragments in traditional software, so-called *clones*.

For ontologies, a *property clump* $\mathcal{P}_C = (C, \mathcal{P})$ is formed by set $C = \{C_1, \dots, C_n\}$ of at least two classes, that all share the same set $\mathcal{P} = \{P_1, \dots, P_m\}$ of properties; these properties can describe data type properties and object properties. Such an unintentionally repeated definition of properties in different classes can occur due to the manual development and evolution of an ontology.

The following DATALOG* rules find all maximal sets Cs of classes that have the same set Ps of properties in common:

```
anomaly(property_clump, [Cs, Ps]) :-
  setof( C, class_has_properties(C, Ps), Cs ),
  length(Cs, N), N > 1.

class_has_properties(C, Ps) :-
  setof( P, class_has_property(C, P), Ps ).
```

We use two nested aggregations based on the powerful PROLOG meta-predicate *setof*/3. The inner aggregation computes a class and its properties. The outer aggregation computes all classes having the same set of properties.

The repeated use of properties of a clump $\mathcal{P}_C = (C, \mathcal{P})$ can be caught by a new class $C_{\mathcal{P}}$, which gets the properties in $\mathcal{P}$. The original classes $C \in C$ are linked to $C_{\mathcal{P}}$ instead of linking them to the properties $\mathcal{P}$. For ontologies with rules, we have to change all rules with property atoms $P(x, y)$ for $P \in \mathcal{P}$ in their antecedent or consequent. The definition of such an abstract property class $C_{\mathcal{P}}$ may increase the compactness and the maintainability (with respect to changes, extensions, fixes) of the ontology.

Fig. 9. Refactoring a property clump to an n-ary relation with the abstract property class P .

An example of a property clump $\mathcal{P} = \{P_1, P_2, P_3, P_4\}$ used by three classes C_1 , C_2 , and C_3 is depicted in Figure 9 (left); the refactored design using an abstract property class C_p is shown at the right of the figure.

The introduction of a new class, that captures related aspects of another class, is also discussed in the ontology design pattern *n-ary relations* [20], where a new class is created in order to link the instances of n individuals to an instance of a single class. With the identification of a property clump, incorrectly modeled n -ary relations may be uncovered. The extraction of such repetitions into a single data structure is a common refactoring, which improves the compactness and maintainability of the implementation.

7. Implementation in DATALOG*

The introduced anomalies have been also defined by an implementation in the new language DATALOG*. Using this language, we have developed a new approach that extends the DATALOG paradigm and mixes it with PROLOG. The analysis can be run using the system DISLOG Developers' Kit (DDK) [30]. This toolkit provides a module including the presented implementation of DATALOG* and the anomaly predicates as well as the shown examples.

For the interested reader, we introduce some technical details of the evaluation mechanisms of DATALOG* in the following. For the detection of anomalies a number of further DATALOG* and PROLOG predicates was used. We describe their implementation in Section 7.3 and Section 7.4.

7.1. Mixing DATALOG and PROLOG: Forward and Backward Chaining

The detection of anomalies in rule ontologies could not be formulated using PROLOG's backward chain-

ing or DATALOG's forward chaining alone, since we need recursion on cyclic data, function symbols (mainly for representing lists), non-grounded facts, disjunction, negation, and aggregation (using meta-predicates) in rule bodies, and stratification.

DATALOG and PROLOG. We distinguish between DATALOG* rules and PROLOG rules: DATALOG* rules are forward chaining rules that may contain function symbols (in rule heads and bodies) as well as negation, disjunction, and PROLOG predicates in rule bodies. DATALOG* rules are evaluated bottom-up, and all possible conclusions are derived.

The supporting PROLOG rules are evaluated top-down, and — for efficiency reasons — only on demand, and they can refer to DATALOG* facts. The PROLOG rules are also necessary for expressivity reasons: they are used for some computations on complex terms, and — more importantly — for computing very general aggregations of DATALOG* facts.

Forward and Backward Chaining. DATALOG* rules cannot be evaluated in PROLOG or DATALOG alone for the following reasons: Current DATALOG engines cannot handle function symbols and non-grounded facts, and they do not allow for the embedded computations (arbitrary built-in predicates), which we need here in this work. Standard PROLOG systems cannot easily handle recursion with cycles, because of non-termination, and are inefficient, because of subqueries that are posed and answered multiply. Thus, they have to be extended by some DATALOG* facilities (our approach) or memoing/tabling facilities (the approach of the PROLOG extension XSB [24]). Since the embedding system, the DDK [30], is developed in SWI-PROLOG, we have implemented a new inference machine that can handle mixed, stratified DATALOG*/PROLOG rule systems.

The evaluation of DATALOG* programs mixes forward-chained evaluation of DATALOG with SLD-

resolution of PROLOG, see Figure 10. A DATALOG* rule $A \leftarrow B_1 \wedge \dots \wedge B_n$ can contain atoms B_i which are evaluated backward in PROLOG.

Fig. 10. Mixing Forward and Backward Chaining.

7.2. Stratified Evaluation of DATALOG*

For the ontology evaluation we have implemented two layers (strata) $\mathcal{D}_1$ and $\mathcal{D}_2$ of DATALOG* rules:

- The upper layer $\mathcal{D}_2$ consists of the rules for the predicate anomaly/2 and some DATALOG* rules that are stated together with them.
- The lower layer $\mathcal{D}_1$ consists of all other DATALOG* rules. For example, the rules for predicates derives and tc_derives are in $\mathcal{D}_1$.

$\mathcal{D}_1$ is applied to the DATALOG* facts for the following basic predicates, which have to be derived from the underlying ontology document:

rule, class, subClassOf,
objectComplementOf, incompatible,
equivalentObjectProperties,
equivalentClasses,
transitive/symmetricObjectProperty,
min/max_cardinality_restriction,
property_restriction, class_has_property.

The resulting DATALOG* facts are the input for $\mathcal{D}_2$. The *stratification* into two layers is necessary, because $\mathcal{D}_2$ refers to $\mathcal{D}_1$ through *negation* and *aggregation*. Most PROLOG predicates in this paper support the layer $\mathcal{D}_2$.

7.3. Further DATALOG* Predicates

The following DATALOG* predicate computes a chain Ps of properties that connect two classes C and D using transitive closure:

```
tc_connected_classes(C, [P], D) :-
    connected_classes(C, P, D).
tc_connected_classes(C, [P|Ps], D) :-
    connected_classes(C, P, E),
    tc_connected_classes(E, Ps, D),
    not(member(P, Ps)).
```

```
connected_classes(C, P, D) :-
    tc_derives(C, C_),
    property_restriction(C_, P, D_),
    tc_derives(D_, D).
```

7.4. Further Supporting PROLOG Predicates

Head and Body. The head and body predicates of a rule can be determined using the following pure PROLOG predicates:

```
head_predicate(_=>A, P) :-
    functor(A, P, _).

body_predicate(Bs=>_, P) :-
    member(B, Bs), functor(B, P, _).

rule_predicate(E) :-
    rule(Rule),
    ( head_predicate(Rule, E)
      ; body_predicate(Rule, E) ).
```

Siblings. The following PROLOG rules define siblings and *aggregate* the siblings Z of a class X to a list Xs using the PROLOG meta-predicate setof/3, respectively:

```
sibling(X, Y) :-
    subClassOf(X, Z),
    subClassOf(Y, Z), X \= Y.

siblings(Zs) :-
    setof(Z, sibling(X, Z), Zs).
```

These rules could also be evaluated in DATALOG* using forward chaining. But, since we need siblings only for certain lists Zs, this would be far too inefficient. The call to setof/3 above succeeds for every class X having siblings, and it computes the list Zs of all siblings Z of X. On backtracking, the siblings of the other classes X are computed. This means, setof/3 does a *grouping* on the variable X. Within setof/3, the call sibling(X, Z) computes one class X and its siblings Z.

Transitivity. Given a DATALOG* rule Rule and a predicate R, the following PROLOG rule tests if R is transitive and then constructs three atoms Rxz, Pxy, and Qyz, where P and Q are body predicates of Rule that

are equivalent to R . Finally, it forms a DATALOG* rule Rule_t from the three atoms.

```
rule_transitivity(Rule, R, Rule_t) :-
  transitiveObjectProperty(R),
  body_predicate(Rule, P),
  equivalentObjectProperties(R, P),
  body_predicate(Rule, Q),
  equivalentObjectProperties(R, Q),
  Pxy =.. [P,X,Y], Qyz =.. [Q,Y,Z],
  Rxz =.. [R,X,Z],
  Rule_t = [Pxy, Qyz]=>Rxz.
```

Symmetry. Given a DATALOG* rule Rule and a predicate R , the following PROLOG rule tests if R is symmetric and then constructs two atoms Pxy and Ryx , where P is a body predicate of Rule that is equivalent to R . Finally, it forms a DATALOG* rule Rule_s from the two atoms.

```
rule_symmetry(Rule, R, Rule_s) :-
  symmetricObjectProperty(R),
  body_predicate(Rule, P),
  equivalentObjectProperties(R, P),
  Pxy =.. [P,X,Y], Ryx =.. [R,Y,X],
  Rule_s = [Pxy]=>Ryx.
```

8. Discussion

For the last couple of years, ontologies have played a major role for building intelligent systems. Currently, the standard ontology language OWL is extended by rule-based elements using, e.g., the rule interchange format RIF or the semantic web rule language SWRL. With the introduction of OWL 2 RL a profile of OWL is defined, that is especially useful for the interchange with rule-based knowledge. We have shown, that with the increased expressiveness of ontologies — now also including rules — a number of new evaluation issues has to be considered. In this paper, we have presented a collection of typical anomalies that arise during practical ontology development, especially when aligning and integrating existing ontologies.

When reviewing the described anomalies, we see that most issues only depend on OWL axioms with a low expressivity, i.e., many anomalies can occur even when using the simple OWL 2 EL profile [13,34]. Only the following anomalies take advantage of more expressive OWL axioms requiring the profiles OWL 2 QL or OWL 2

RL: Circular Properties (Sec. 3.2), Multiple Functional Properties (Sec. 4.5), Redundant Implication of Transitivity or Symmetry (Sec. 5.5), Redundant Use of Transitivity and Symmetry (Sec. 5.6.2), and Redundant Cardinalities (Sec. 5.9). About a half of the issues required the existence of rules in the knowledge base. Of course, the presented anomalies only gave a brief insight into the collection of possible verification issues.

Anomalies were considered concerning the basic elements of the ontology language. When using *built-ins*, the detection of anomalies becomes a difficult task, since the semantics of built-ins can rarely be evaluated at the symbolic level. Simple problems occurring with built-ins are easily detectable, especially the definition of identical knowledge. For instance, the assessment of the Body-mass-index (BMI) in two medical ontologies a and b is redundant given the rules

$$a:\text{hasBMI}(P,W) \wedge \text{op:num-greater-than}(W,25) \\ \Rightarrow a:\text{overweight}(P)$$

and the rule

$$b:\text{calBMI}(P,W) \wedge \text{op:num-greater-than}(W,25) \\ \Rightarrow b:\text{heavy}(P)$$

where $a:\text{hasBMI}$ and $b:\text{calBMI}$ are defined as equivalent, and $a:\text{overweight}$ and $b:\text{heavy}$ are equivalent, respectively. Here, easily detectable inconsistencies can be identified; for example, when the numeric threshold is specified differently in the above rules. The analysis of more complex definitions, however, becomes much more difficult, when the semantics of the built-ins cannot be mapped to the symbolic level.

For all discussed anomalies, we have introduced a declarative approach using DATALOG* for implementing the anomaly checks for ontology verification. Due to its declarative nature, new methods for anomaly detection can be easily added to the existing work. From our point of view, this is crucial, because of the incompleteness of the presented anomalies: in principle, giving a complete overview of possible anomalies is not feasible, since the number of anomalies depends on the expressiveness of the ontology and the rule representation, respectively, that should be verified.

The actual frequency of the introduced anomalies is an interesting issue. However, only a small number of ontologies (mostly toy examples) is available that make use of a rule extension. A sound review of anomaly occurrences would require a reasonable number of practical ontologies with a significant size.

Furthermore, larger systems may also include parts of a non-monotonic rule base. Here, some work has been done on the verification of non-monotonic rule bases [37,38], that has to be re-considered in the presence of an ontological layer.

References

- [1] G. Antoniou, F. van Harmelen, *A Semantic Web Primer*, 2nd ed., MIT Press, 2008.
- [2] F. Baader, U. Sattler, An Overview of Tableau Algorithms for Description Logics, *Studia Logica* 69 (2001) 5–40.
- [3] J. Baumeister, *Agile Development of Diagnostic Knowledge Systems*, IOS Press, AKA, DISKI 284, 2004.
- [4] J. Baumeister, D. Seipel, Smelly owls – design anomalies in ontologies, in: *FLAIRS'05: Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference*, AAAI Press, 2005.
URL <http://ki.informatik.uni-wuerzburg.de/papers/baumeister/2005/FLAIRS05OntoSmells.pdf>
- [5] J. Baumeister, D. Seipel, Verification and refactoring of ontologies with rules, in: *EKAU'06: Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management*, Springer, Berlin, 2006.
URL http://ki.informatik.uni-wuerzburg.de/papers/baumeister/2006/EKAU06_baumeisterSWRL.pdf
- [6] J. Baumeister, D. Seipel, F. Puppe, Refactoring methods for knowledge bases, in: *EKAU'04: Engineering Knowledge in the Age of the Semantic Web: 14th International Conference*, LNAI 3257, Springer, Berlin, 2004.
URL <http://ki.informatik.uni-wuerzburg.de/papers/baumeister/2004/Refactoring-EKAU04.pdf>
- [7] M. Buffa, F. Gandon, G. Ereteo, P. Sander, C. Faron, SweetWiki: A semantic wiki, *Web Semantics* 8 (1) (2008) 84–97.
- [8] S. Ceri, G. Gottlob, L. Tanca, *Logic Programming and Databases*, Springer, Berlin, 1990.
- [9] M. Fowler, *Refactoring. Improving the Design of Existing Code*, Addison-Wesley, 1999.
- [10] A. Gómez-Pérez, Evaluation of taxonomic knowledge on ontologies and knowledge-based systems, in: *KAW'99: Proceedings of the 12th International Workshop on Knowledge Acquisition, Modeling and Management*, 1999.
- [11] A. Gómez-Pérez, Evaluation of ontologies, *International Journal of Intelligent Systems* 16 (3) (2001) 391–409.
- [12] A. Gómez-Pérez, M. Fernández-López, O. Corcho, *Ontological Engineering*, Springer, 2004.
- [13] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, U. Sattler, OWL 2: The next step for OWL, *Web Semantics* 6 (4) (2008) 309–322.
URL <http://www.sciencedirect.com/science/article/B758F-4TP1FC8-1/2/9d2f647c7ac874b8f8baa9cf92cf73a3>
- [14] N. Guarino, C. Welty, Evaluating ontological decisions with OntoClean, *Communications of the ACM* 45 (2) (2002) 61–65.
- [15] Y. Guo, Z. Pan, J. Heflin, LUBM: A benchmark for OWL knowledge base systems, *Web Semantics* 3 (2) (2005) 158–182.
- [16] I. Horrocks, B. Parsia, P. Patel-Schneider, J. Hendler, Semantic web architecture: Stack or two towers?, in: F. Fages, S. Soliman (eds.), *Principles and Practice of Semantic Web Reasoning (PPSWR)*, No. 3703 in LNCS, Springer, 2005.
- [17] I. Horrocks, P. F. Patel-Schneider, S. Bechhofer, D. Tsarkov, OWL rules: A proposal and prototype implementation, *Web Semantics* 3 (1) (2005) 23–40.
- [18] I. Horrocks, U. Sattler, A tableaux decision procedure for *SHOIQ*, in: *IJCAI'05: Proc. of the 19th International Joint Conference on Artificial Intelligence*, 2005.
- [19] M. Krötzsch, S. Rudolph, P. Hitzler, ELP: Tractable rules for OWL 2, in: *ISWC'08: Proceedings of the 7th International Semantic Web Conference*, Springer, Berlin, 2008.
- [20] N. Noy, A. Rector, Defining n-ary relations on the semantic web, Tech. rep., W3C Working Group Note (12 April 2006).
- [21] W. F. Opdyke, *Refactoring Object-Oriented Frameworks*, Ph.D. thesis, University of Illinois, Urbana-Champaign, IL, USA (1992).
- [22] A. Preece, R. Shinghal, Foundation and application of knowledge base verification, *International Journal of Intelligent Systems* 9 (1994) 683–702.
- [23] A. Preece, R. Shinghal, A. Batarekh, Principles and practice in verifying rule-based systems, *The Knowledge Engineering Review* 7 (2) (1992) 115–141.
- [24] P. Rao, K. F. Sagonas, T. Swift, D. S. Warren, J. Freire, XSB: A system for efficiently computing well-founded semantics, in: *Logic Programming and Non-monotonic Reasoning*, 1997.
URL citeseer.ist.psu.edu/article/rao97xsb.html
- [25] A. L. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, C. Wroe, OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns, in: *EKAU'04: Engineering Knowledge in the Age of the Semantic Web: 14th International Conference*, LNAI 3257, Springer, 2004.
- [26] R. Rosati, On the decidability and complexity of integrating ontologies and rules, *Web Semantics* 3 (1) (2005) 61–73.
- [27] L. Sauermaun, G. A. Grimnes, M. Kiesel, C. Fluit, H. Maus, D. Heim, D. Nadeem, B. Horak, A. Dengel, Semantic desktop 2.0: The Gnowsis experience, in: *ISWC'06: Proceedings of the 5th International Semantic Web Conference*, LNCS 4273, 2006.
- [28] S. Schaffert, F. Bry, J. Baumeister, M. Kiesel, Semantic wiki, *IEEE Software* 25 (4) (2008) 8–11.
- [29] G. Schreiber, H. Akkermans, A. Anjewierden, R. de Hoog, N. Shadbolt, W. V. de Velde, B. Wielinga, *Knowledge Engineering and Management - The CommonKADS Methodology*, 2nd ed., MIT Press, 2001.
- [30] D. Seipel, The DisLog Developers' Kit (DDK): <http://www1.informatik.uni-wuerzburg.de/databases/DisLog>.
- [31] L. Stojanovic, A. Maedche, B. Motik, N. Stojanovic, User-driven ontology evolution management, in: *EKAU'02: Ontologies and the Semantic Web*, 13th International Conference, LNAI 2473, Springer, Berlin, 2002.
- [32] Y. Sure, S. Staab, R. Studer, On-to-knowledge methodology (OTKM), in: S. Staab, R. Studer (eds.), *Handbook on Ontologies*, International Handbooks on Information Systems, Springer, 2004.
- [33] W3C, RIF-BLD Specification: <http://www.w3.org/tr/rif-bld> (July 2008).
- [34] W3C, OWL2 Profiles: <http://www.w3.org/tr/owl2-profiles/> (April 2009).
- [35] W3C, Semantic Web activity: <http://www.w3.org/2001/sw/> (May 2009).
- [36] J. Wielemaier, An overview of the SWI-Prolog programming environment, in: *WLPE'03: Proceedings of the 13th International Workshop on Logic Programming Environments*, 2003.
- [37] N. P. Zlatareva, Verification of non-monotonic knowledge bases, *Decision Support Systems* 21 (4) (1997) 253 – 261.
- [38] N. P. Zlatareva, Testing the integrity of non-monotonic knowledge bases containing semi-normal defaults, in: *FLAIRS'04: Proceedings of the 17th International Florida Artificial Intelligence Research Society Conference*, AAAI Press, 2004.