Antipattern detection: How to debug an ontology without a reasoner

Catherine Roussey, Ondrej Zamazal

To cite this version:

Catherine Roussey, Ondrej Zamazal. Antipattern detection: How to debug an ontology without a reasoner. Second International Workshop on Debugging Ontologies and Ontology Mappings (WoDOOM), May 2013, Montpellier, France. p. 45 - p. 56. hal-00861052

HAL Id: hal-00861052
https://hal.science/hal-00861052

Submitted on 11 Sep 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Antipattern Detection: How to Debug an Ontology without a Reasoner

Catherine Roussey[1] and Ondřej Zamazal[2]

[1] Irstea, 24 Av. des Landais, BP 50085, 63172 Aubiére, France
catherine.roussey@irstea.fr
[2] Knowledge Engineering Group, University of Economics Prague, Czech Republic
ondrej.zamazal@vse.cz

Abstract. In ontology design, an Ontology Design Pattern (ODP) is a modeling solution to a recurrent ontology design problems. As opposed to ODP, antipatterns are bad modeling practices. This paper deals with detection of antipattern for debugging purpose of huge ontologies. We focus on the detection of the Onlyness Is Loneliness (OIL) antipattern. We propose an antipattern detection method based on ontology transformations and SPARQL queries. This approach does not need reasoner to detect antipattern. Our method detects candidates of OIL antipattern. These candidates localize class definitions where OIL occurrences can appear. This enables to draw ontology developer's attention to avoid errors during ontology development process. We conduct some experiments to detect OIL antipattern in an OWL ontology corpus obtained from the Watson ontology search engine.

Keywords: OWL, OWL-DL, ontology, ontology pattern, antipattern, SPARQL, ontology transformation

1 Introduction

In ontology design an Ontology Design Pattern (ODP) is a modeling solution to a recurrent ontology design problems as defined in [7]. As opposed to ODP, antipatterns are bad modeling practices implemented in ontologies. Antipatterns produce the side effect of inferring wrong or undesired knowledge or of preventing the capabilities to infer the desired knowledge [16]. First, ontology antipatterns might help with guiding and training new ontology developers (educational purpose). Second, antipatterns can be directly used for ontology design purpose since ontology designers could take advantage of antipattern detection using some ontology editor during ontology development. Finally, detection of ontology antipatterns can contribute to ontology quality assessment.

In our previous work we first published our catalogue of antipatterns [6]. We have also provided some recommandations about ontologies repairing processes based on the antipattern detection. These patterns basically resulted in unsatisfiable classes or modeling errors due to the misuse or misunderstanding of Description Logics (DL) expressions. In [18] we proposed a general approach of

antipattern detection based on SPARQL queries which was applied on selected antipatterns from the catalogue [6].

The motivation for the work presented in this paper was twofold. On the one side, in [18] we show that each antipattern needs its own specific detection method. On the other side, reasoners might have troubles to tackle ontologies with complex axioms. But, reasoner output is usually prerequisite for a detection of complex antipatterns. Therefore, here we come up with an extension of our general detection method for one specific antipattern and in the situation that we cannot apply a reasoner. Instead of reasoner we apply ontology transformation pre-processing step and we evaluate it on one of the most complex antipatterns from our catalogue, *Onlyness Is Loneliness* (OIL) antipattern. Transformation have several goals: harmonization of ontology developer's implementation style, simulation of reasoner inferences and simplification of class definition axioms. It enables us to detect candidates of OIL antipattern. They are localized in class definitions where OIL occurrences can appear. Thus, they draw ontology developer's attention so that (s)he can avoid errors arised during ontology development process. We conduct some experiments to detect OIL antipattern in an OWL ontology corpus obtained from the Watson ontology search engine.[3]

The rest of the paper is structured as follows. Next section gives a brief overview of ontology design patterns and antipatterns for ontology development. Section 3 describes the OIL antipattern that is used to run our experiments. Next, Section 4 describes the detection method and our transformation rules. Section 5 describes the experiment setup and the results of the experimentation. Finally, Section 6 wraps up the paper by providing conclusions and future work.

2 Related Work

From Regarding educational purpose of bad modeling practices in DL, authors in [17] describe common difficulties in understanding of the logical meaning of expressions for newcomers to DL. Explicit antipatterns are then proposed in [6] presenting a catalogue of antipatterns based on DL expressions and proposing some recommendations on how to repair them.

To the best of our knowledge research in antipatterns are mainly connected to ontology debugging task. One of the earliest work was the OntoClean method [8], which defined a set of meta-properties applied to classes and a set of procedures to check and correct the subsumption relations between classes. Other source of antipatterns is [19], where authors proposed four terminological patterns applied on class names to detect possible errors along taxonomy. Next, the OntOlogy Pitfalls Scanner (OOPS) [15] enables an ontology developer to detect common pitfalls during the development of ontology.

There are several tools which can be used for antipattern detection. They are mostly available inside ontology editors and require the use of a reasoner to provide their justifications. For instance Pellint [5] focuses on the detection and

[3] `http://watson.kmi.open.ac.uk/`

repair of antipatterns to improve ontology reasoning performance. The Protégé Explanation Workbench [9] and SWOOP [12] provide justifications of inconsistencies in ontologies based on the output from DL reasoners. SWOOP has also a repair plug-in to help user during the debugging process [11]. However, using a reasoner for this purpose is not always possible, since in some big ontologies reasoners fail to provide any result [13]. Furthermore, the antipatterns repertory that these tools can detect is fixed.

Next, Ontology Pre-processor Language (OPPL) [10] enables pattern-based manipulation with ontologies. Authors of [14] describe an experiment using OPPL to detect ontology design patterns in a repository of biomedical ontologies. They also use transformations to harmonize the ontology, but their set of transformations only normalize 'syntactic sugar' conventions of OWL 2.

3 "Onlyness Is Loneliness" (OIL) Antipattern

One of the most common error made by ontology developer is captured by *Onlyness Is Loneliness (OIL) antipattern*. This antipattern can be manifested by one of the following sets of DL axioms:

$$C_3 \sqsubseteq \forall r.C_1; C_3 \sqsubseteq \forall r.C_2; \;\; Disj(C_1, C_2); \tag{1}$$
$$C_3 \equiv \forall r.C_1; C_3 \sqsubseteq \forall r.C_2; \;\; Disj(C_1, C_2); \tag{2}$$
$$C_3 \equiv \forall r.C_1; C_3 \equiv \forall r.C_2; \;\; Disj(C_1, C_2); \tag{3}$$
$$C_3 \sqsubseteq \forall r.C_1 \sqcap \forall r.C_2; \;\; Disj(C_1, C_2); \tag{4}$$
$$C_3 \equiv \forall r.C_1 \sqcap \forall r.C_2; \;\; Disj(C_1, C_2); \tag{5}$$

An OIL antipattern occurrence is defined by two disjoint classes C_1 and C_2 and a third class C_3 whose definition contains two universal restrictions using a property r. The first universal restriction expresses that instances of C_3 can only be linked with r to instances of C_1.[4] The second one expresses that instances of C_3 can only be linked with r to instances of C_2.

Based on our experience the origin of this error is that an ontology developer forgets that there is already a constraint about the class C_3 using an universal restriction and (s)he adds a new constraint about this class using another universal restriction. Moreover, one of these constraints can be inherited from any of the parent classes.

This error is already mentioned in several works such as [17] or [15]. In [15], the OIL antipattern is linked to the pitfall number 14, related to the misuse of universal restriction. Notice that even if the OIL antipattern is declared as Pitfall 14 in the catalogue of common pitfalls, the OntOlogy Pitfalls Scanner is not yet able to detect it.

[4] To be detectable, property r must have at least a value, normally specified as a (minimum) cardinality restriction for that class, or with existential restrictions.

Detection of OIL antipattern is a difficult task (similarly to other antipatterns) due to various problems. First, antipatterns have various manifestations. For example, we present above 5 logical formulae related to OIL but we could imagine more formulae. Furthermore, ontology developers can have very different implementation styles when designing an OWL ontology. For example, some developers prefer to write long class definitions. In that case, a class is defined by a conjunction of unnamed classes (or anonymous classes), e.g., $C \sqsubseteq (\exists R.X) \sqcap (\forall R.Y)$. In that case parts of antipattern can also be located at different places. Others can prefer to write short definitions. A class is defined by a set of atomic axioms,[5] e.g., $C \sqsubseteq \exists R.X; C \sqsubseteq \forall R.Y$. Each implementation style corresponds to a different logical formula of the OIL antipattern. Finally, we also have to consider that parts of the OIL antipattern can be asserted by the ontology developer or inferred by a reasoner.

4 OIL Antipattern Detection Method

General detection approach was presented in [18] based on SPARQL queries and reasoner output. However, none of those presented methods gives significant results for the OIL antipattern. Here, we describe a method aiming at detection of OIL antipattern in OWL ontologies. The main objective of this new method is to enable detection of OIL antipattern without reasoner output. As we experienced during debugging/repairing of complex ontologies (e.g. HydrOntology) by using the reasoner (e.g. Pellet)[6] and the Explanation Workbench tool,[7] applying reasoner for the antipattern detection is not always possible. It turns out that reasoner may fail to provide any results on complex ontologies. Generally, the more the number of repaired axioms increases, the more time reasoner needs to take in order to provide justifications for unsatisfiable classes.

This new method consists of two steps. First transformation rules are applied, see Section 4.1, and then SPARQL query (or, in general, SPARQL queries), see Section 4.2, is (are) executed using *PatOMat ontology pattern detection tool*, see Figure 1.[8] This tool is part of the PatOMat suite of tools, which is focused on the pattern detection in ontologies and their transformations. This detection tool is based on Jena 2.6.2 [1] and Pellet 2.0.1 [2], and enables the processing of a set of SPARQL queries over a set of ontologies in a batch. It produces a report in terms of numbers of patterns detected and details for each ontology. It processes either only asserted axioms or both inferred and asserted axioms of given ontology.

Our transformations have several objectives:

[5] We defined an atomic axiom as a constraint (necessary condition $\sqsubseteq$ or sufficient condition $\equiv$) associated to a named class C using at most one DL constructor ($\forall$, $\exists$, $\neg$ or $\sqcap$) and its associated operands: one class and one property for $\forall$, $\exists$ and two classes for $\sqcap$. All these classes should be named classes.

[6] http://owl.cs.manchester.ac.uk/explanation/

[7] http://owl.cs.manchester.ac.uk/explanation/

[8] http://owl.vse.cz:8080/DetectionTool/

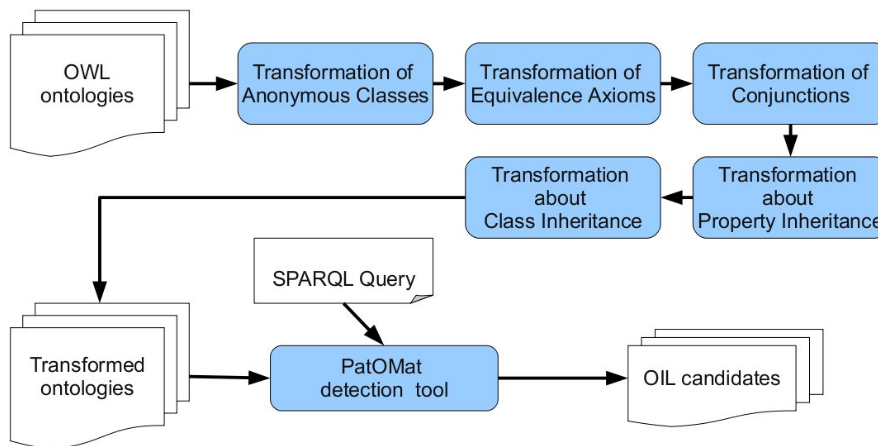

Fig. 1. Overview of the OIL detection method

- to decompose the class definition to simpler set of atomic axioms,
- to harmonize the different implementation styles of ontology developers,
- to simulate inferences in order to avoid applying a reasoner.

Our transformation rules only add new axioms and do not remove any original axioms from the ontology. The transformation rules have to be applied in the specific order. The rules are incremental, which means that the new axioms created by one transformation rule are already considered by a subsequent transformation rule.

According to [18], the use of a reasoner is mandatory to detect a disjoint axiom. In this previous work, we tried to detect OIL antipattern with asserted disjointness axioms. Since only few OIL occurrences were detected, we came up with the following hypothesis:

Hypothesis On the one side the disjoint axiom of the OIL antipattern is difficult to detect without a reasoner output and on the other side it turns out that asserted disjoint axioms do not help in detection. Thus, we limit the detection of OIL antipattern to the two first atomic axioms:

$$C_3 \sqsubseteq \forall r.C_1; C_3 \sqsubseteq \forall r.C_2; \tag{6}$$

We defined a class definition that contains this set of axioms as an OIL candidate.

4.1 Applied Transformation Rules

Here, we will explain our transformation rules one by one. For clarity purpose the transformation rules presented in this section are merely dedicated to the

OIL detection. For the general antipattern detection, there is a larger set of transformation rules.

TR AC: Transformation of Anonymous Classes TR AC consists of the following transformation rules:

$$C \sqsubseteq \forall r.(X \sqcup Y); \Rightarrow C \sqsubseteq \forall r.Or_X_Y; Or_X_Y \equiv X \sqcup Y; \tag{7}$$
$$C \sqsubseteq \forall r.(X \sqcap Y); \Rightarrow C \sqsubseteq \forall r.And_X_Y; And_X_Y \equiv X \sqcap Y; \tag{8}$$
$$C \sqsubseteq \forall r.(\exists s.Z); \Rightarrow C \sqsubseteq \forall r.Some_S_Z; Some_S_Z \equiv \exists s.Z; \tag{9}$$
$$C \sqsubseteq \forall r.(\forall s.Z); \Rightarrow C \sqsubseteq \forall r.Only_S_Z; Only_S_Z \equiv \forall s.Z; \tag{10}$$
$$C \equiv \forall r.(X \sqcup Y); \Rightarrow C \equiv \forall r.Or_X_Y; Or_X_Y \equiv X \sqcup Y; \tag{11}$$
$$...$$

The goal of this transformation is to name anonymous class in order to detect any antipattern in new named class definitions. Moreover, this transformation removes brackets and shortens class definitions. Some ontology developers write long definition of class using anonymous classes (or unnamed classes), e.g., as on the left-hand-side of formulae above. To simplify the query of OWL ontologies, class definitions must be transformed in this way. Ideally, class definitions are only composed of atomic axioms, so we need to remove anonymous classes, e.g. $(X \sqcup Y)$, $(X \sqcap Y)$ or $(\exists s.Z)$. The antipattern can be located in the anonymous class definition so we need to create a named class for each of them. In order to easily detect that these named classes come from the transformation rule we apply the following naming convention, e.g., And_X_Y in the case of $(X \sqcap Y)$.

In all, this transformation will be applied on each anonymous class in brackets so that new named class will be created and will replace original anonymous class in all axioms of the ontology. This new class will be defined by the anonymous class content and ideally correctly placed to the taxonomy, e.g. $And_X_Y \equiv X \sqcap Y$ should be a child class of X and of Y and $Or_X_Y \equiv X \sqcup Y$ should be a parent class of X and of Y.

TR EA: Transformation of Equivalence Axioms TR EA consists of the following transformation rule:

$$C_A \equiv C_B; \Rightarrow C_A \sqsubseteq C_B; C_B \sqsubseteq C_A; \tag{12}$$

The $\equiv$ symbol should be transformed into both-sided $\sqsubseteq$, i.e. we create two new axioms with the subClassOf relationship. This transformation is necessary because each antipattern can be written with a $\equiv$ or $\sqsubseteq$ symbol. Due to this transformation rule we can just consider atomic axioms of the form $C \sqsubseteq ...$ for antipattern detection.

In all, this transformation will be applied on all equivalence axioms according to which new two subsumptions (subClassOf) axioms will be added.

TR Conj: Transformation of Conjunctions TR Conj consists of the following transformation rules:

$$\left.\begin{array}{l} C \sqsubseteq \forall r.X_1 \sqcap ... \\ \sqcap \exists r.X_2 \sqcap ... \\ \sqcap \leq x\ r.\top \sqcap ... \\ \sqcap \geq x\ r.\top \sqcap ... \\ \sqcap = x\ r.\top \sqcap ... \\ \sqcap X_i ... \\ \sqcap X_n \end{array}\right\} \Rightarrow \begin{cases} C \sqsubseteq \forall r.X_1; \\ C \sqsubseteq \exists r.X_2 \\ C \sqsubseteq \leq x\ r.\top; \\ C \sqsubseteq \geq x\ r.\top; \\ C \sqsubseteq = x\ r.\top; \\ ...; \\ C \sqsubseteq X_i; \\ ...; \\ C \sqsubseteq X_n; \end{cases} \tag{13}$$

An ontology developer has her/his own implementation style. Some of them prefer to write long axioms using conjunction of anonymous classes. On the contrary, others prefer to define lots of named classes in order to identify small part of knowledge. Depending on the implementation style of the ontology developer, the antipattern detection may be facilitated. For antipattern detection, we recommend to split all the long axioms into several atomic axioms.

In all, this transformation will be applied on conjunction of anonymous classes so that it will be split into its components, e.g. $C \sqsubseteq \forall r.X \sqcap \exists r.Y$ will generate two new axioms $C \sqsubseteq \forall r.X$ and $C \sqsubseteq \exists r.Y$.

TR PI: Transformation about Property Inheritance TR PI consists of the following transformation rule:

$$\begin{array}{l} r_1 \sqsubseteq r; \\ C_A \sqsubseteq \forall r_1.C_B; \Rightarrow C_A \sqsubseteq \forall r.C_B; \end{array} \tag{14}$$

The ontology developer can forget that (s)he has defined a property r_1 as a subproperty of another one, r. Thus, for each axiom using the subproperty r_1 we need to add a redundant axiom using the parent property r. This transformation is applied along the whole taxonomy of properties. The new axiom changes the semantics of the ontology. Thus, it should be added only if the class C_A is already defined by a universal restriction using the r property in order to check that there is no conflict between different universal restrictions using r.

TR CI: Transformation about Class Inheritance TR CI consists of the following transformation rule:

$$C_B \sqsubseteq C_A; C_A \sqsubseteq \forall r.Y; \Rightarrow C_B \sqsubseteq \forall r.Y; \tag{15}$$

The main purpose of this transformation is to simulate reasoning so that all subclasses inherit all (asserted) constraints from their parents. It is applied at the end of the transformation process because we want to inherit all previously added (due to the application of other transformation rules) axioms as well.

4.2 SPARQL Query

According to Figure 1 SPARQL query, by using PatOMat detection tool, is performed after application of transformations. The following query retrieves OIL candidates defined by equation 6:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?c3 ?r ?c1 ?c2
WHERE
{ ?c3 rdfs:subClassOf _:restrictionA1.
_:restrictionA1 rdf:type owl:Restriction.
_:restrictionA1 owl:onProperty ?r.
_:restrictionA1 owl:allValuesFrom ?c1.
?c3 rdfs:subClassOf _:restrictionB1.
_:restrictionB1 rdf:type owl:Restriction.
_:restrictionB1 owl:onProperty ?r.
_:restrictionB1 owl:allValuesFrom ?c2.
FILTER
( isURI(?c1) && isURI(?c2) && isURI(?c3) && ?c1!= ?c2 )
}
ORDER BY ?c3  ?r ?c1
```

The query finds a class $?c3$ defined by two universal restrictions using the same property $?r$. The first restriction is linked to the $?c1$ class, the second one is linked to the $?c2$ class. As shown in the filter part of the query, all the class variables ($?c3$, $?c1$, $?c2$) should be named classes.

OIL candidates localize class definitions where OIL antipattern occurrences can appear. Thus, these candidates draw ontology developer's attention to avoid errors arised during long ontology development process.

5 Experimentation: Finding Antipatterns in Real-world Ontologies

In this section, we describe the results of our experiments with a corpus of ontologies downloaded directly from the Web and by the Watson semantic search engine. We will first describe how we have built the ontology corpus, and then we present the results of applying our detection method from Section 4 over this ontology corpus.

5.1 Building a Corpus of (Debuggable) OWL Ontologies

We have used the Watson API [3] to retrieve publicly available ontologies and we have always accessed these ontologies using the Watson cache, since there are sometimes mismatches between the stored URIs of those ontologies and the

actual files that can be obtained. We searched ontologies satisfying the following two constraints: they should be represented in OWL and they should have at least five classes. We collected 66 inconsistent ontologies. From these ontologies we have selected ontologies that cannot be classified by Pellet in a reasonable time. Using PatOMat, we queried and processed several sets of ontologies at the same time. We materialized the inferred axioms using Pellet and then we queried the resulting ontologies with a OIL query.[9] If Pellet could not classify the ontology or if the classifying process took more than 10 seconds, the ontology was used for our experimentation. [10]

Table 1 presents the ontologies used for our experimentation. For each ontology, this table indicates: its number of classes, information whether Pellet can classify it or not, the time which classification process by Pellet takes, its number of unsatisfiable classes. The last column indicates if the ontology was debugged or not. We use the Explanation Workbench tool [9] to debug these ontologies. This tool provides for each unsatisfiable class the minimal set of axioms in which the class is unsatisfiable. Regarding repairing process we do not simply remove problematic axioms. Our repairing process is rather collaborative task between a DL expert and a domain expert, i.e. the DL expert identifies what the domain expert wants to express and proposes the correct DL axioms capturing intended domain knowledge. When an ontology is debugged it means that we know the total number of OIL occurrences.

name	nr. of classes	Pellet	time execution	nr. of unsatisf. classes	debugged
HydrOntology	159	no		114	yes
Tambis	395	yes	27 s	112	yes
inconsistent 613	6	no		1	yes
inconsistent 623	11	no		1	yes
Open Cyc	2846	yes	43 s	1663	no
proteonic	401	yes	17 s	5	no
CSNCS	1274	yes	139 s	154	no

Table 1. the list of ontologies

5.2 Experiments

Evaluation of Antipattern Detection Precision The precision of the OIL candidate detection process was evaluated. One evaluator analyzed the SPARQL results and assigned to each OIL candidate one of the following values:

– *TI* (True positive Inconsistency): the evaluator is sure that the OIL candidate participates in the unsatisfiability of classes.

[9] This query is looking for the RDF representation of the OIL antipattern expressed by the equation number 1 in section 3

[10] All of these ontologies are available from [4].

- *UI* (Unknown Inconsistency): the evaluator is not able to take a decision.
- *FI* (False positive Inconsistency): the evaluator is sure that the OIL candidate does not participate in the unsatisfiability of classes.

5.3 Results of OIL candidate detection

Due to the symmetric property of OIL candidate[11], the query identifies an OIL candidate twice. Table 2 presents the query results and the evaluation results. The first column indicates the total number of OIL candidates found by the query. The next columns indicate the number of TI, UI and FI from the OIL candidates decided by the evaluator.

All the FI we found in the query results come from the fact that: (1) $c1$ and $c2$ are linked by a subClassOf relationship; (2) the $c1$ class is built by the TR AC transformation rule, and $c2$ class is one of the named classes used in the $c1$ definition. For example, there is the following OIL candidate in the results from the Hydrontology: $c3 = Poza; r = parte_de; c1 = OR_Lago_Rio; c2 = Rio$.

set	nr. of results	nr. of TI	nr. of UI	nr. of FI
HydrOntology	84	44 (52 %)	0	41 (48 %)
Tambis	314	0	121 (39 %)	193 (61 %)
inconsistent 613	0	0	0	0
inconsistent 623	6	6 (100 %)	0	0
Open Cyc	0	0	0	0
Proteonic	610	0	65 (11 %)	545 (89 %)
CSNCS	21 914	0	0	21 914 (100 %)

Table 2. results of OIL candidate detection process

In the case of HydrOntology all the OIL antipattern occurrences were retrieved by our query. In the case of Tambis ontology, it does not contain any OIL antipattern occurrences. But thanks to the OIL candidates retrieved by the query, some dangerous classes definitions are detected because there exist two universal restrictions with sibling classes (these sibling classes are not inferred as disjoint classes so that there is no OIL antipattern). For example, we found: $c3 = gene-product; r = polymer-of; c1 = ribo-nucleotide; c2 = amino-acid$. Such results were tagged as UI by the evaluator.

The inconsistent 623 and 613 ontologies were debugged and they do not contain any OIL antipattern occurrences. These ontologies contain only one unsatisfiable class that is very difficult to debug due to the presence of transitive and inverse properties. In the case of the inconsistent 623 ontology, the detected OIL candidates identify one of the class involved in the class unsatisfiability.

[11] C_1 and C_2 can be switched in the equation 6.

Proteonic ontology was not yet debugged. The OIL candidates retrieved by the query identify some dangerous classes because there exist some OIL candidates, e.g. $c3 = collection; r = direct - part - of; c1 = collection; c2 = element$. These results were tagged as UI by the evaluator.

CSNCS ontology imports several ontologies. It imports the DUL ontology which describe classes using lots of universal restrictions. All the OIL candidates contains some DUL classes like Entity, Object, Social Object. This ontology also imports several large ontologies, e.g. the DOLCE Ultra Light ontology, which leads to many OIL candidates detected by the query.

Results from Table 2 are encouraging, because OIL candidates may be useful to detect error or dangerous class definitions. Using a list of transformations and a simple SPARQL query is sufficient for detecting complex antipattern like OIL one, even on large ontology that cannot be processed by a reasoner. Moreover during long development process it is useful to detect OIL candidates in order to localize dangerous class definitions where an error can often occur.

In this experimentation we have validated the fact that transformation processes and simple SPARQL queries are sufficient for detection some OIL antipattern without a reasoner. Note that in our previous work [18], the previous detection method based on reasoner output and using SPARQL queries were not able to detect any OIL occurrences at all. If we want to apply our method to other antipattern, we should define the adequate transformation rules and SPARQL queries.

Our immediate work will aim at presenting the SPARQL results so that the candidates where $c1$ and $c2$ classes are linked by a subClassOf relationship will be eliminated.

6 Conclusions and Future Work

In this paper we have shown how complex antipatterns such as OIL can be detected. Our detection method can work without a reasoner. It is based on ontology transformations and one SPARQL query. These transformations have several goals: harmonizing ontology developer implementation style, simulating reasoner inference and simplifying class definition axioms. Our method detects OIL antipattern candidates. These candidates localize class definition where OIL occurrences can appear. Thus, they draw ontology developer's attention to avoid errors during long ontology development process. We conduct some experiments to detect OIL antipattern in an OWL ontology corpus obtained from the Watson ontology search engine. Our future work will focus on the definition of new transformations to detect other complex antipatterns from our antipattern catalogue.

References

1. Apache jena. `http://jena.apache.org/` (2012)
2. Pellet: Owl 2 reasoner for java. `http://clarkparsia.com/pellet/` (2012)

3. Watson: Exploring the semantic web. http://watson.kmi.open.ac.uk/WS_and_API.html (2012)
4. web site related to our ontology antipattern detection methods. https://sites.google.com/site/ontologyantipattern (2012)
5. Clark, K.: Pellint: An ontology repair tool. http://weblog.clarkparsia.com/2008/07/02/pellint-an-ontology-repair-tool/ (2008)
6. Corcho, O., Roussey, C., Vilches Blázquez, L.M., Pérez, I.: Pattern-based OWL ontology debugging guidelines. In: Proceedings of WOP. pp. 68–82. CEUR-WS.org (2009)
7. Gangemi, A., Presutti, V.: Ontology design patterns. Handbook on Ontologies pp. 221–243 (2009)
8. Guarino, N., Welty, C.A.: Evaluating ontological decisions with OntoClean. Commun. ACM 45(2), 61–65 (2002)
9. Horridge, M., Parsia, B., Sattler, U.: Laconic and precise justifications in OWL. In: Proceedings of ISWC. pp. 323–338 (2008)
10. Iannone, L., Rector, A.L., Stevens, R.: Embedding knowledge patterns into OWL. In: Proceedings of ESWC. pp. 218–232 (2009)
11. Kalyanpur, A., Parsia, B., Sirin, E., Cuenca Grau, B.: Repairing unsatisfiable concepts in OWL ontologies. In: Proceedings of ESWC. pp. 170–184 (2006)
12. Kalyanpur, A., Parsia, B., Sirin, E., Hendler, J.: Debugging unsatisfiable classes in OWL ontologies. Journal of Web Semantics 3(4), 268–293 (2005)
13. Lehmann, J., Bhmann, L.: ORE - a tool for repairing and enriching knowledge bases. In: proceedings of ISWC. LNCS, vol. 6497- part 2, pp. 177–193. Springer, Shanghai, China (2010)
14. Mortensen, J., Horridge, M., Musen, M., Noy, N.: Modest use of ontology design patterns in a repository of biomedical ontologies. In: Proceedings of WOP. vol. 929. CEUR-WS.org, Boston, USA (2012)
15. Poveda-Villalon, M., Suarez-Figueroa, M.C., Gomez-Perez, A.: Validating ontologies with OOPS! In: proceedings of EKAW. LNCS, vol. 7603, pp. 267–281. Springer Berlin Heidelberg, Ireland (2012)
16. Presutti, V., Blomqvist, E., Daga, E., Gangemi, A.: Pattern-based ontology design. Ontology Engineering in a Networked World pp. 35–64 (2012)
17. Rector, A.L., Drummond, N., Horridge, M., Rogers, J., Knublauch, H., Stevens, R., Wang, H., Wroe, C.: OWL pizzas: Practical experience of teaching OWL-DL: common errors & common patterns. In: Proceedings of EKAW. pp. 63–81 (2004)
18. Roussey, C., Corcho, O., Svab-Zamazal, O., Scharffe, F., Bernard, S.: SPARQL-DL queries for antipattern detection. In: Proceedings of WOP. vol. 929. CEUR-WS.org, Boston, USA (2012)
19. Šváb-Zamazal, O., Svátek, V.: Analysing ontological structures through name pattern tracking. In: Proceedings of EKAW. pp. 213–228 (2008)