Pattern-based OWL Ontology Debugging Guidelines

Oscar Corcho[1], Catherine Roussey[23], Luis Manuel Vilches Blázquez[1], and Iván Pérez[4]

[1] Ontology Engineering Group, Departamento de Inteligencia Artificial, Universidad Politécnica de Madrid, Spain
[2] Université de Lyon, CNRS, Université Lyon 1, LIRIS UMR5205, Villeurbanne, France
[3] Cemagref, 24 Av. des Landais, BP 50085, 63172 Aubiére, France
[4] IMDEA Software, Madrid, Spain

Abstract. Debugging inconsistent OWL ontologies is a tedious and time-consuming task where a combination of ontology engineers and domain experts is often required to understand whether the changes to be performed are actually dealing with formalisation errors or changing the intended meaning of the original knowledge model. Debugging services from existing ontology engineering tools and debugging strategies available in the literature aid in this task. However, in complex cases they are still far from providing adequate support to ontology developers, due to their lack of efficiency or precision when explaining the main causes for unsatisfiable classes, together with little support for proposing solutions for them. We claim that it is possible to provide additional support to ontology developers, based on the identification of common antipatterns and a debugging strategy, which can be combined with the use of existing tools in order to make this task more effective.

1 Introduction

Ontology engineering methodologies describe the sets of activities to be carried out for ontology building, together with methods and techniques that can be applied to them. Among these activities, those of ontology formalisation and implementation appear in most methodologies, since the final objective is to obtain one or several ontologies that describe the domain according to the ontology requirement specifications provided in the early stages of development.

Formalisation and implementation activities have different degrees of difficulty, considering the knowledge representation formalism and ontology language selected, and the ontology requirements, among others. For example, implementing an RDF(S) ontology is less difficult than implementing an OWL ontology; and developing a small ontology where only primitive concepts are needed is much simpler than developing a network of ontologies where defined concepts are extensively used and complex inferences have to be drawn upon.

Similarly, there are different degrees of difficulty in ontology debugging. However, these are not deeply characterised in existing ontology engineering methodologies and methods (not even in effort estimation approaches like ONTOCOM [12]). Only some works (e.g., [18]) evaluate existing ontology debugging tools, showing the major issues in this task: run-time performance and robustness of the debugging tool result.

If we focus on DL formalisation and OWL implementation, several debugging tools exist (OWLDebugger [5] [4], SWOOP [8], [7], RepairTab [10]), which have proven their effectiveness in different domains, isolating the minimal set of axioms containing a conflict that leads to the unsatisfiability of a class (MUPS). Tools like RepairTab [10] also propose alternatives to resolve the identified conflicts, showing those entailments that would be lost if the proposed solution was applied. Nevertheless solutions are always limited to two choices: removing part of the existing axioms or replacing a class by one of its superclasses. And ontologies used in the experiments (e.g., the mad cow one) have been written by DL experts with the purpose of showing those conflicts.

Our focus is on real ontologies that have been developed by domain experts, who are not necessarily too familiar with DL, and hence can misuse DL constructors and misunderstand the semantics of some OWL expressions, leading to unwanted unsatisfiable classes. To illustrate this, we will use throughout the paper examples taken from a medium-sized OWL ontology (165 classes) developed by a domain expert in the area of hydrology [19]. The first version of this ontology had a total of 114 unsatisfiable classes. The information provided by the debugging systems used ([5], [8]) on (root) unsatisfiable classes was not easily understandable by domain experts to find the reasons for their unsatisfiability. And in several occasions during the debugging process the generation of justifications for unsatisfiability took several hours, what made these tools hard to use, confirming the results described in [18]. As a result, we found out that in several occasions domain experts were just changing axioms from the original ontology in a somehow random manner, even changing the intended meaning of the definitions instead of correcting errors in their formalisations.

Using this and several other real ontologies we have made an effort to identify common unsatisfiability-leading patterns used by domain experts when implementing OWL ontologies, together with common alternatives for providing solutions to them, so that they can be used by domain experts to debug their ontologies. Then we provide some hints about how to organise the iterative ontology debugging process using a combination of debugging tools and patterns.

2 Patterns and AntiPatterns

In software engineering, a design pattern can be defined as a general, proven and beneficial solution to a common re-occurring problem in software design [2]. Built upon similar experiences, design patterns represent best practices about how to build software. On the contrary, antipatterns are defined as patterns that

appear obvious but are ineffective or far from optimal in practice, representing worst practice about how to structure and build software [9].

In knowledge (and more specifically in ontology) engineering the concept of knowledge modelling (ontology design) pattern is used to refer to modelling solutions that allow solving recurrent knowledge modelling or ontology design problems [1][14][13][15]. A similar definition is given for knowledge modelling (or ontology design) antipatterns.

Different types of ontology design patterns are defined [14]:

- Logical Ontology Design Patterns (LP). They are independent from a specific domain of interest, but dependent on the expressivity of the logical formalism used for representation. For example, the n-ary relation pattern enables to model n-ary relations in OWL DL ontologies.
- Architectural Ontology Design Patterns (AP). They provide recommendations about the structure of an ontology. They are defined in terms of LPs or compositions of them. Examples are: taxonomy or lightweight ontology.
- Content Ontology Design Patterns (CP). They propose domain-dependent conceptual models to solve content design problems for the domain classes and properties that populate an ontology. They usually exemplify LPs, and they represent most of the work done on ontology design patterns.

In contrast to ontology design patterns, the work on antipatterns is less detailed ([11],[16],[17],[3]). [3] define a set of class metaproperties and associated patterns in order to check subsumption links and correct them. [17] proposed four patterns based on class names in order to detect possible errors in the taxonomic structure. Four logical antipatterns are presented in [11], all of them focused on property domains and ranges. [16] describes common difficulties for newcomers to DL in understanding the logical meaning of expressions. However, none of these contributions groups antipatterns in a common classification, nor provide a comprehensive set of hints to debug them.

2.1 A Classification of Ontology Design AntiPatterns

We have identified a set of patterns that are commonly used by domain experts in their DL formalisations and OWL implementations, and that normally result in unsatisfiable classes or modelling errors. As aforementioned all these antipatterns come from a misuse and misunderstanding of DL expressions by ontology developers. Thus they are all Logical AntiPatterns (LAP): they are independent from a specific domain of interest, but dependent on the expressivity of the logical formalism used for the representation. We have categorized them into three groups:

- Detectable Logical AntiPatterns (DLAP). They represent errors that DL reasoners and debugging tools normally detect.
- Cognitive Logical AntiPatterns (CLAP). They represent possible modelling errors that may be due to a misunderstanding of the logical consequences of the used expression.

- Guidelines (G). They represent complex expressions used in an ontology component definition that are correct from the logical and cognitive points of view, but for which the ontology developer could have used other simpler alternatives or more accurate ones for encoding the same knowledge.

In the rest of this section we describe the antipatterns identified in each group, providing their name and acronym, their template logical expressions and a brief explanation of why this antipattern can appear and how it should be checked by the ontology developer. It is important to note that DLAPs generate unsatisfiable classes that are normally identified by existing ontology debugging tools, although the information that is provided back to the user is not described according to such a pattern, what makes it difficult for ontology developers to find a good solution according to their domain formalisation. With respect to CLAP and G, they are not detected by these tools as such, although in some cases their combination may lead to unsatisfiable classes that are detected (although not appropriately explained) by tools. As we mention in our future work section, we think that tool support for them could be a major step forward in this task.

Finally, all these antipatterns should be seen as elementary units that cause ontology incoherence. That is, they can be combined into more complex ones. However, providing a solution for the individual ones is already a good advance to the current state of the art, and our future work will be also devoted to finding the most common combinations and providing recommendations for them.

2.2 Detectable Logical AntiPatterns (DLAP)

As aforementioned, these antipatterns represent errors that DL reasoners can detect. They can be classified into four main groups: those related to the misunderstanding of the logical conjunction, those related to the incorrect use of universal restrictions, those related to the incorrect use of the combination of universal/existential restrictions and those related to the incorrect representation of disjoint and equivalent knowledge. We now describe them in detail, with examples taken from earlier versions of HydrOntology [19] [5], and with proposed solutions for them, which should be always validated with the domain expert to make sure that the intended meaning of the represented knowledge model does not change.

AntiPattern AndIsOr (AIO) $C_1 \sqsubseteq \exists R.(C_2 \sqcap C_3); Disj(C_2, C_3);$ [6]

[5] All original examples presented in this paper are in Spanish, and we also provide their approximate translation in English for ease of understanding (given the specificity of the domain, not all terms can be translated directly into English). This ontology and several versions obtained throughout the debugging process are available at http://www.dia.fi.upm.es/ ocorcho/OWLDebugging/

[6] This does not mean that the ontology developer has explicitly expressed that C_2 and C_3 are disjoint, but that these two concepts are determined as disjoint from each other by a reasoner. We use this notation as a shorthand for $C_2 \sqcap C_3 \sqsubseteq \bot$.

This is a common modelling error that appears due to the fact that in common linguistic usage, "and" and "or" do not correspond consistently to logical conjunction and disjunction respectively [16]. For example, "I like cake with almond and with chocolate" is ambiguous. Does the cake contain?

- Some chocolate plus some almond? $Cake \sqsubseteq \exists contain.Chocolate \sqcap \exists contain.Almond$;
- Chocolate-flavoured almond? $Cake \sqsubseteq \exists contain.(Chocolate \sqcap Almond)$;
- Some chocolate or some almond? $Cake \sqsubseteq \exists contain.(Chocolate \sqcup Almond)$;

In the original version of HydrOntology this antipattern appeared twice. We present one instance of this antipattern with its approximate translation into English [7] .
$Caño \sqsubseteq \exists comunica.(Albufera \sqcap Mar \sqcap Marisma)$;
$Pipe \sqsubseteq \exists communicate.(Lagoon \sqcap Sea \sqcap Salt\_Marsh)$;

In order to solve this antipattern we propose replacing the logical conjunction by the logical disjunction, or by the conjunction of two existential restrictions.

~~$C_1 \sqsubseteq \exists R.(C_2 \sqcap C_3)$~~; $Disj(C_2, C_3); \Rightarrow C_1 \sqsubseteq \exists R.(C_2 \sqcup C_3)$; or
$C_1 \sqsubseteq \exists R.C_2$ *and* $\exists R.C_3$;

AntiPattern OnlynessIsLoneliness (OIL) $C_1 \sqsubseteq \forall R.(C_2); C_1 \sqsubseteq \forall R.(C_3); Disj(C_2, C_3)$; [8]

The ontology developer created a universal restriction to say that C_1 instances can only be linked with property R to C_2 instances. Next, a new universal restriction is added saying that C_1 instances can only be linked with R to C_3 instances, with C_2 and C_3 disjoint. In general, this is because the ontology developer forgot the previous axiom in the same class or in any of the parent classes.

The following is one of the two definitions of HydrOntology class where this antipattern can be found :
$Aguas\_de\_Transición \sqsubseteq \forall está\_próxima.Aguas\_Marinas \sqcap \forall está\_próxima.Desembocadura \sqcap = 1 está\_próxima.\top$;
$Transitional\_Water \sqsubseteq \forall is\_nearby.Sea\_Water \sqcap \forall is\_nearby.River\_Mouth \sqcap = 1 is\_nearby.\top$;

If it makes sense, we propose to the domain expert to transform the two universal restrictions into only one that refers to the logical disjunction of C_2 and C_3.

~~$C_1 \sqsubseteq \forall R.C_2$~~; ~~$C_1 \sqsubseteq \forall R.C_3$~~; $Disj(C_2, C_3); \Rightarrow C_1 \sqsubseteq \forall R.(C_2 \sqcup C_3)$;

AntiPatterns UniversalExistence (UE) $C_1 \sqsubseteq \exists R.C_2; C_1 \sqsubseteq \forall R.C_3; Disj(C_2, C_3)$;

[7] For better readability, we do not specify in these examples that the used classes are disjoint.

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

The ontology developer adds an existential/universal restriction to a class without remembering that there was already an inconsistency-leading universal/existential restriction in the same class or in a parent class, respectively.

The following is one of 3 examples of this antipattern in HydrOntology:
$Gola \sqsubseteq Canal\_Aguas\_Marinas; Gola \sqsubseteq \exists comunica.Ria;$
$Canal\_Aguas\_Marinas \sqsubseteq \forall comunica.Aguas\_Marinas;$
$Inlets \sqsubseteq Sea\_Waters\_Canal; Inlets \sqsubseteq \exists communicate.Rivers;$
$Sea\_Waters\_Canals \sqsubseteq \forall communicate.Sea\_Waters;$

These antipatterns are difficult to debug because ontology developers sometimes do not distinguish clearly between existential and universal restrictions. Our proposal is aimed at resolving the unsatisfiability of a class, but as usual it should be clearly analysed by the ontology developer.

$C_1 \sqsubseteq \exists R.C_2; \cancel{C_1 \sqsubseteq \forall R.C_3}; Disj(C_2, C_3) \Rightarrow C_1 \sqsubseteq \forall R.(C_2 \sqcup C_3);$

AntiPattern UniversalExistenceWithInverseProperty (UEWIP)
$C_2 \sqsubseteq \exists R^{-1}.C_1; C_1 \sqsubseteq \forall R.C_3; Disj(C_2, C_3);$

The ontology developer added restrictions about C_2 and C_1 using a property R but he didn't remember that he had already used its inverse property R^{-1}. The following is an example of this antipattern in HydrOntology:
$Aguas\_Marinas \sqsubseteq \exists es\_alimentada.Aguas\_Quietas\_Naturales;$
$Aguas\_Quietas\_Naturales \sqsubseteq \forall alimentada.Aguas\_Corrientes\_Naturales;$
$Sea\_Water \sqsubseteq \exists is\_fed\_by.Natural_Standing_Water;$
$Natural\_Standing\_Water \sqsubseteq \forall feed.Natural\_Watercourse;$

We propose to add the reverse axiom of the C_2 definition $C_1 \sqsubseteq \exists R.C_2$ and if it makes sense to add a class disjunction in the universal restriction.

$C_2 \sqsubseteq \exists R^{-1}.C_1; \cancel{C_1 \sqsubseteq \forall R.C_3}; Disj(C_2, C_3); \Rightarrow C_1 \sqsubseteq \forall R.(C_2 \sqcup C_3);$

AntiPattern EquivalenceIsDifference (EID) $C_1 \equiv C_2; Disj(C_1, C_2);$

This pattern, which is only common for ontology developers with no previous training in OWL modelling, comes from the fact that the ontology developer wants to say that C_1 is a subclass of C_2, or viceversa, but at the same time it is different from C_2 since he has more information. After a short training session the developer would discover that he really wants to express $C_1 \sqsubseteq C_2$
The following is an example of this antipattern in HydrOntology:
$Cascada \equiv Catarata; Disj(Cascada, Catarata);$
$Cascade \equiv Waterfall; Disj(Cascade, Waterfall);$

We propose to ask the ontology developer whether he really wants to define a synonym or a subclass-of relation. Depending on the ontology developerś answer, the equivalent axiom should be transformed into a subclass-of one or the less used concept should be suppressed according to the SOE recommendations.

$\cancel{C_1 \equiv C_2}; Disj(C_1, C_2) \Rightarrow C_1 \sqsubseteq C_2$ or C_2 is a label of C_1;

2.3 Cognitive Logical AntiPatterns (CLAP)

As aforementioned, these antipatterns are not necessarily errors, but describe common templates that ontology developers use erroneously trying to represent a different piece of knowledge.

AntiPattern SynonymOrEquivalence (SOE) $C_1 \equiv C_2$;

The ontology developer wants to express that two classes C_1 and C_2 are identical. This is not very useful in a single ontology that does not import others. Indeed, what the ontology developer generally wants to represent is a terminological synonymy relation: the class C_1 has two labels: C_1 and C_2. Usually one of the classes is not used anywhere else in the axioms defined in the ontology.

The following is an example of this antipattern in HydrOntology:
$Corriente\_Subterránea \equiv Rio\_Subterráneo$;
$Subterranean\_Watercourse \equiv Subterranean\_River$;

The proposal for avoiding this antipattern is the following (if C_2 is the less used term in the ontology) add all the comments and labels of C_2 into C_1 and remove C_2.

$\cancel{C_1 \equiv C_2} \Rightarrow C_1.[RDFS:label|comment] = C_2.[RDFS:label|comment]$;

2.4 Guidelines

In contrast to the antipatterns already described, guidelines represent complex expressions used in an ontology component definition that are correct from a logical point of view, but in which the ontology developer could have used other simpler alternatives for encoding the same knowledge. The recommendations provided for Guidelines mainly focus on making the ontology easier to understand by ontology developers, and do not make any change with respect to the semantics or intended meaning of the ontology. We have determined that the proposed changes make the ontology easier to understand by asking a good range of ontology developers about their preferences when analysing ontologies that were not developed by them.

Guideline DisjointnessOfComplement (DOC) $C_1 \equiv not\ C_2$;

During the development process of a new ontology, it is hard to know that C_1 is the logical negation of C_2. Maybe the ontology developper will define later C_3 as a sister class of C_1 and C_2. Thus we recommend to say that C_1 and C_2 cannot share instances first, and change the definition of C_1 as a negation of C_2 if necessary at the end of the development. The following is an example of this antipattern in HydrOntology:
$Laguna\_Salada \equiv not\ Aguas\_Dulces$;
$Salt\_Lagoon \equiv not\ Fresh\_Water$;

We propose: $\cancel{C_1 \equiv not\ C_2} \Rightarrow Disj(C_1, C_2)$;

Guideline Domain&CardinalityConstraints (DCC) $C_1 \sqsubseteq \exists R.C_2$; $C_1 \sqsubseteq (\geq 2R.\top)$; *(for example)*

Ontology developers with little background in formal logic find difficult to understand that "only" does not imply "some" [16]. This antipattern is a counterpart of that fact. Developers may forget that existential restrictions contain a cardinality constraint: $C_1 \sqsubseteq \exists R.C_2 \vDash C_1 \sqsubseteq (\geq 1R.C_2)$. Thus, when they combine existential and cardinality restrictions, they may be actually thinking about universal restrictions with those cardinality constraints.

The following is an example of this antipattern in HydrOntology:
$Aguas\_de\_Transición \sqsubseteq \exists sometida\_a\_influencia.Aguas\_Dulces \sqcap$
$\exists sometida\_a\_influencia.Aguas\_Saladas \sqcap$
$\forall sometida\_a\_influencia.(Aguas\_Dulces \sqcup Aguas\_Saladas) \sqcap$
$= 1 sometida\_a\_influencia.\top$;
$Transitional\_Water \sqsubseteq \exists is\_influenced\_by.Fresh\_Water \sqcap$
$\exists is\_influenced\_by.Salt\_Water \sqcap$
$\forall is\_influenced\_by.(Fresh\_Water \sqcup Salt\_Water) \sqcap$
$= 1 is\_influenced\_by.\top$;

We propose to transform the existential restriction into a universal one when a cardinality restriction exists.

~~$C_1 \sqsubseteq \exists R.C_2$~~; $C_1 \sqsubseteq (\geq 2R.\top); \Rightarrow C_1 \sqsubseteq \forall R.C_2$;

Guideline GroupAxioms (GA) $C_1 \sqsubseteq \forall R.C_2; C_1 \sqsubseteq (\geq 2R.\top)$; *(for example)*

For visualisation purposes, we recommend grouping all the restrictions of a class that use the same property R in a single restriction. This recommendation is to facilitate the visualisation of complex class definitions.

~~$C_1 \sqsubseteq \forall R.C_2$~~; ~~$C_1 \sqsubseteq (\geq 2R.\top)$~~; $\Rightarrow C_1 \sqsubseteq \forall R.C_2 \sqcup (\geq 2R.\top)$;

Guideline MinIsZero (MIZ) $C_1 \sqsubseteq (\geq 0R.\top); \Rightarrow$ ~~$C_1 \sqsubseteq (\geq 0R.\top)$~~;

The ontology developer wants to remember that C_1 is the domain of the R property. This restriction has no impact on the logical model being defined and can be removed at the end of the development process. This antipattern appeared once in the HydrOntology debugging process. $Laguna\_Salada \sqsubseteq (\geq 0 es\_alimentada.\top)$;
$Salted\_Lagoon \sqsubseteq (\geq 0 fedBy.\top)$;

3 Ontology Debugging Strategy

As mentioned in the introduction, OWL ontology debugging features have been proposed in the literature with different degrees of formality ([5], [8], [20]). They allow identifying the main root for unsatisfiable classes and superfluous axioms and restrictions, and in some cases they explain them with different degrees of detail, so that the debugging process can be guided by them and can be made more efficient. However, in general these features are mainly focused on the explanations of logical entailments and are not so focused on the ontology

engineering side. Hence explanations are still difficult to understand for ontology developers. Furthermore, there are no clear strategies about how an ontology developer should debug incoherent ontologies, in terms of steps to be followed in this process. Consequently, we think that there is a need to complement both types of suggestions in order to make the ontology debugging process more efficient.

Figure 1 shows graphically a usual ontology debugging lifecycle, with the roles of knowledge engineer and domain expert identified. As it happens in other disciplines (e.g., software development), the first step is to locate where the problems are, using a reasoner directly or an ontology debugging tool, which may also identify root unsatisfiable classes, one of which can be chosen. Otherwise any of the unsatisfiable classes that are in the top of the class hierarchy can be selected. Then antipatterns have to be identified for the selected class. Based on the antipattern identification, the knowledge engineer proposes recommendations for corrections, such as the ones presented in the section 2. Recommendations cannot always be automated, since they may change the intended meaning of the ontology, and should be documented. Once a change is done, new unsatisfiability checks should be performed. This is an iterative process to be followed until there are no more unsatisfiable classes in the ontology.

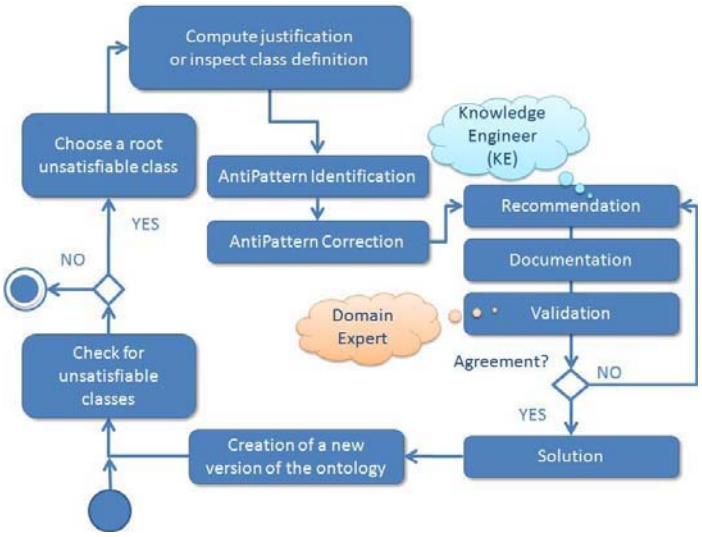

Fig. 1. Global strategy for ontology debugging.

Identifying antipatterns at each step in the process may be a hard task, even for experienced ontology developers. Thus we also propose a more detailed strategy, based on the catalogue of antipatterns described in section 2. We propose to follow a specific order, based on our experience, as summarised in Figure 2.

First, we recommend solving terminological problems (SOE), checking the use of equivalence and disjoint constructors between classes (EID, DOC) and applying the guideline GA in order to make formal definitions easier to understand, grouping in the same definition all the axioms dealing with the same role. These are the easiest antipatterns to detect and they are useful to clean other ontology definitions.

Then we propose checking the root unsatisfiable classes in the ontology, using any debugging tool. At that point we can check the use of conjunction (AIO), the use of universal restrictions (OIL), and combinations of universal and existential restrictions (UE, DCC). Sometimes, unsatisfiability arises from a combination of several antipatterns, so that is the reason why there are loops in the figure.

After solving problems in root unsatisfiable classes, the branch of the class hierarchy should be checked manually from the leaf to the root to detect if the same antipattern is present in any class of the branch.

Finally, we recommend removing superfluous axioms (MIZ) to improve the clarity of the ontology. However, this could be really done at any point in time throughout the ontology debugging process.

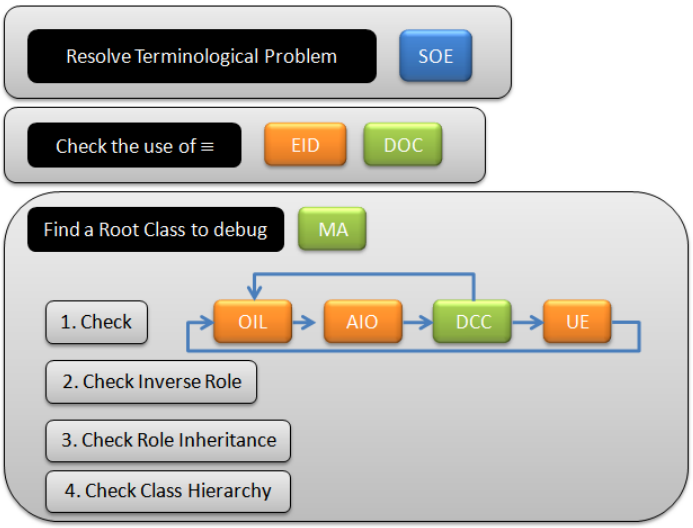

Fig. 2. Detailed debugging strategy based on antipatterns.

4 Evaluation

In order to evaluate our debugging strategy, we conducted a user study with two groups of subjects, using Protégé OWL v4 and its associated explanation

workbench [5]. The ontology tested was the first version of HydrOntology, which has been used to provide examples in the previous sections. Table 1 summarises some of its characteristics.

Number of classes	Number of unsatisfiable classes	Number of object properties	Number of datatype properties	Number of class axioms	Average number of axioms per class
165	114	47	64	625	4

Table 1. The characteristics of Hydrontology

4.1 User Study

Fourteen volunteer subjects, who where postgraduate students in the Computing Science Department at the Universidad Politécnica de Madrid, were chosen for the evaluation. Most of these subjects had the same profile: they were experienced in software debugging, they had basic knowledge in description logics and OWL, and they had no knowledge about the Hydrology field.

Our initial hypotheses, which we wanted to test with this study, were:

Hypothesis 1: The subjects using our debugging strategy and our set of antipatterns will take less time to debug the ontology.

Hypothesis 2: The subjects using our set of antipatterns will take less time to find the problematic parts of axioms.

Hypothesis 3: The subjects using our set of antipatterns and their associate recommendations will provide a better solution according to the domain expert than just remove the problematic part of the axioms.

The study was conducted as follows. Each subject was given a tutorial on the debugging of DL axioms and another tutorial on using Protégé v4 with the explanation workbench [4]. None of the subjects had seen the ontology before. They were divided into two similar groups, with similar profiles and similar size. Group 2 was given an additional third tutorial about all the antipatterns, with general examples, and about our proposed debugging strategy.

The subjects in the two groups were asked to answer two surveys , where they were provided with a set of fixed questions to answer (dealing with specific classes), and in all cases they had to specify the time needed to answer each query.

The first survey was about the perceived quality of the ontology. For this survey both groups could only use Protégé to browse the ontology, without the use of any reasoner or the Protégé explanation workbench. Besides, subjects in Group 2 could also use the documentation on antipatterns that we had provided them. The questions were about which parts of the axioms of specific classes would not be taken in account by a reasoner when checking satisfiability, about existing discrepancies between the formal and the natural language definitions of classes, about how to rewrite axioms to make them more understandable, and

about the existence of duplicate classes. The second survey was focused on how they would be able to solve existing unsatisfiable classes in the original ontology. They were asked to find the problematic axioms that lead to unsatisfiability in specific classes and provide some solutions. They could use reasoners Fact++ or Pellet1.5 and the Protégé explanation workbench, plus the set of antipatterns in the case of Group 2.

4.2 Analysis of the results

Survey ID	Number of queries	Number of classes involved in the survey	Number of subjects who completed the survey	Average time (in minutes) to complete the survey
S1	44	36	G1: 6 G2: 8	G1: 167 (+31) G2: 136
S2	18	25	G1: 4 G2: 8	G1: 187 G2: 225 (+38)

Table 2. Survey results and characteristics

Hypothesis 1: Concerning the first hypothesis about time, our debugging strategy doesn't seem to reduce the time needed for debugging. As shown in Table 2 the subjects using our strategy (Group 2) completed the first survey in less time than Group 1, but needed more time to complete survey 2. In any case, the differences in time were not very relevant.

However, it is important to notice that most of the subjects complained about the fact that the reasoner crashed regularly while performing debugging activities, and in those cases they found out that the availability of a catalogue of antipatterns was useful to go on working in the meantime.

Hypothesis 2: Concerning the hypothesis related to finding more quickly the errors, the results are mixed. It seems that it depends on the complexity of the antipatterns. The subjects using our set of antipatterns found more quickly the errors related to the SOE, EID, DOC and MIZ antipatterns. The explanation is that these antipatterns are easier to find in an ontology development tool user interface. However, when an unsatisfiable class contains an error which is the concatenation of several antipatterns, subjects in group 2 were not able to perform better than those in group 1. Finally, when an unsatisfiable class contained too many axioms almost none of the subjects managed to find the error. For example the class Río contains 15 axioms and only one subject per group found the error.

Hypothesis 3: Concerning the solution of errors, our recommendations helped in the debugging process. The quality of a solution is evaluated by comparing the result axiom with the one belonging to the final Hydrontology version debugged by a knowledge engineer and the domain expert. When they manage to identify an antipattern, the subjects in group 2 provided a more accurate and precise

solution than those in group 1, who were mainly removing axioms randomly in order to make the class satisfiable. That is, without our recommendations the main resolution strategy is still to remove completely the problematic axioms.

5 Conclusions and future work

In this paper we have described a strategy for OWL ontology debugging that can be used in combination with existing OWL ontology debugging services in order to improve the efficiency of the debugging process by having a predefined set of suggested actions to be performed by ontology developers. We have obtained this strategy taking into account our experience in the development of DL-based ontologies and a careful analysis about how ontology developers and ontology engineers debug their ontologies nowadays.

As part of the work that we had to do in order to come up with this strategy, we have collected a list of common antipatterns that can be found in domain-expert-developed ontologies and that cause a large percentage of the unsatisfiability of classes. Besides, we have listed some antipatterns that do not have an impact on the logical consequences of the ontology being developed, but that are of importance in order to reduce the number of errors in the intended meaning of ontologies or to improve their understandability.

For the time being, our strategy is mainly manual, where debugging tools are used to detect some unsatisfiable classes or propose sets of axioms containing antipatterns, although it remains to the user to find out exactly where the antipattern is and which antipattern is applied.

We have evaluated our strategy and compared it with current practice in ontology debugging by using two groups of volunteers that have worked with an incoherent ontology in the geographical domain. As a result, we can confirm that our strategy does not reduce the debugging time but it improves the quality of debugging, that is, our proposed recommendations help finding a more appropriate solution to an error. The other conclusion of our evaluation is that users have some difficulty to find the antipatterns among all the axioms defining an unsatisfiable class.

Hence as part of our future work we are aiming at implementing additional tools that can be used in combination with existing debugging tools (e.g., the Protégé explanation workbench) to help in the identification of antipatterns. For the time being we have started applying the OPPL language [6] for this task, with promising results.

Another part of future work will be related to applying this strategy for the debugging of well-known inconsistent ontologies (e.g., TAMBIS). In this case we would be extending our work to that of experts debugging ontologies that have not been written by them. And we will also focus on how anti-patterns are usually combined together and how more complex ones can be found that can speed up even more the debugging process.

Finally, some of the explanations that we have provided for the appearance of antipatterns are related to the order in which some of the restrictions and axioms

have been added to the ontology. Hence keeping a record of the changes that have been made to the ontology following well-known ontology change management systems could be useful in order to incorporate this into the antipattern detection phases and providing possible ranked solutions to them.

Acknowledgements This work is a result of collaboration between OEG, LIRIS lab and IMDEA Software. It has been done under the context of the project GeoBuddies, funded by the Spanish Ministry of Science and Technology and it was also partially funded by the COST Action C21 sponsored by the European Commission under the grant number STSM-C21-04241. The work of IMDEA Software on this paper has been partially funded by the Spanish Ministry of Industry, Tourism and Trade under the grant FIT-340503-2007-2, as part of the Morfeo EzWeb strategic singular project.

References

1. Clark P, Thompson J, Porter B.: Knowledge patterns. In Proceedings of 7th International Conference Principles of Knowledge Representation and Reasoning (KR), Breckenridge, Colorado, USA: 591-600. (2000)
2. Gamma E, Helm R, Johnson R, Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2. (1995)
3. Guarino N and Welty C: Evaluating Ontological Decisions with OntoClean. Communications of the ACM. 45(2):61-65. New York:ACM Press, (2002).
4. Horridge M.: Understanding and Repairing Inferences. Tutorial in the 11th Intl. Protégé Conference - June 23-26, 2009 - Amsterdam, Netherlands.
5. Horridge M, Parsia B, Sattler U.: Laconic and Precise Justifications in OWL. In Proceedings of the 7th International Semantic Web Conference (ISWC), Karlsruhe, Germany; LNCS 5318: 323-338. (2008).
6. Iannone L, Rector A, Stevens R.: Embedding Knowledge Patterns into OWL. In proceedings of the 6th European Semantic Web Conference (ESWC2009), Crete, Greece. The Semantic Web: Research and Applications (2009), pp. 218-232
7. Kalyanpur A, Parsia B, Cuenca-Grau B.: Beyond Asserted Axioms: Fine-Grain Justifications for OWL-DL Entailments. Description Logics 2006
8. Kalyanpur A, Parsia B, Sirin E, Cuenca-Grau B.: Repairing Unsatisfiable Classes in OWL Ontologies. In Proceedings of the 3rd European Semantic Web Conference (ESWC), Budva, Montenegro; LNCS 4011: 170-184 (2006)
9. Koenig A.: Patterns and Antipatterns. Journal of Object-Oriented Programming 8(1):46-48. (1995)
10. Lam J, Pan JZ, Sleeman D, Vasconcelos W. A Fine-Grained Approach to Resolving Unsatisfiable Ontologies. Journal of Data Semantics (JoDS) 10:62-95. 2008
11. Laboratory of Applied Ontology: Collection of antipatterns from http://wiki.loa-cnr.it/index.php/LoaWiki:MixedDomains
12. Paslaru E, Simperl B, Popov I O, Bürger T.: ONTOCOM Revisited: Towards Accurate Cost Predictions for Ontology Development Projects. In Procs. of the 6th European Semantic Web Conference, ESWC 2009, Heraklion, Greece, June 2009, LNCS 5554: 248-262 (2009)

13. Presutti V., Gangemi A.: Content Ontology Design Patterns as practical building blocks for web ontologies. In Proceedings of the 27th International Conference on Conceptual Modeling (ER), Barcelona, Spain, LNCS 5231: 128-141(2008).
14. Presutti V, Gangemi A, David S, Aguado G, Suarez-Figueroa MC, Montiel E, Poveda M.: Neon Deliverable D2.5.1: A Library of Ontology Design Patterns available at <http://www.neon-project.org>
15. Rech J, Feldmann R L, Ras E.: Knowledge Patterns. In M. E. Jennex (Ed.), Encyclopedia of Knowledge Management (2nd Edition), IGI Global, USA, (2009).
16. Rector AL, Drummond N, Horridge M, Rogers L, Knublauch H, Stevens R, Wang H, Wroe C.: OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns. In Proceedings of the 14th International Conference Knowledge Acquisition, Modeling and Management (EKAW), Whittlebury Hall, UK. LNCS 3257: 63-81 (2004)
17. Sváb-Zamazal O, Svátek V: Analysing Ontological Structures through Name Pattern Tracking. In Proceedings of the 16th International Conference, EKAW 2008, Acitrezza, Italy, September 29 - October 2, 2008. Lecture Notes in Computer Science 5268 Springer 2008, ISBN 978-3-540-87695-3: 213-228 (2008)
18. Stuckenschmidt H.: Debugging OWL Ontologies - a Reality Check. In Proceedings of the 6th International Workshop on Evaluation of Ontology-based Tools and the Semantic Web Service Challenge (EON-SWSC-2008), Tenerife, Spain. (2008).
19. Vilches-Blázquez LM, Bernabé-Poveda MA, Suárez-Figueroa MC, Gómez-Pérez A, Rodríguez-Pascual AF: Towntology & hydrOntology: Relationship between Urban and Hydrographic Features in the Geographic Information Domain. In Ontologies for Urban Development. Studies in Computational Intelligence, vol. 61, Springer: 73-84. (2007)
20. Wang, H., Horridge M, Rector A, Drummond N, Seidenberg J.: Debugging OWL-DL Ontologies: A heuristic approach. In Proceedings of the 4th International Semantic Web Conference (ISWC), Galway, Ireland; LNCS 3729: 745-757(2005)