A Heuristic Approach to Explain the Inconsistency in
OWL Ontologies

Hai Wang  Matthew Horridge  Alan Rector
Nick Drummond  Julian Seidenberg
Department of Computer Science,
The University of Manchester,
Manchester M13 9PL, UK
{hwang,mhorridge,rector,ndrummond,jms}@cs.man.ac.uk

1 Introduction

Most modern OWL-DL reasoners can only provide lists of unsatisfiable classes without
offering any explanation as to why those classes are inconsistent. The process of deter-
mining the cause of inconsistent classes, what we refer to as debugging the ontology,
is a task that is left for the user. Even expert ontology engineers can find it difficult
to work out why a class has been marked as inconsistent. When faced with several in-
consistent classes in a moderately large ontology, the task of debugging can become
onerous indeed.

In this paper we present a heuristic approach for debugging OWL ontologies, which
can be utilised by tools and therefore go some way to automating the debugging pro-
cess, thereby alleviating the user from this troublesome task. We are are focusing about
debugging OWL, however the approach can be applied to general description logics.
Since the debugger is based on heuristic, it is not a complete solution. However it do
provide the explanation to a majority of inconsistent cases.

The heuristic we used is from the experience of a series of tutorials, workshops
and post-graduate modules we have presented over the past five years, teaching people
to use OWL-DL and its predecessors effectively. Based on our experience, a set of
common made mistakes have been identified. These common have been used as the
basic for developing a set of ontology debugging heuristics. These heuristics have been
used in an implementation of an ontology debugger for the Prot´eg´e-OWL [1] ontology
development environment, which with the assistance of a DL-reasoner, can generate
descriptions that explain the reasons for inconsistent OWL classes.

The debugger that has been developed treats the DL-reasoner as a ‘black box’. In
other words, the debugger does not know the details of any reasoning algorithms that
are used by the DL-reasoner. This ‘black box’ approach obviously has many benefits –
in particular it is reasoner independent, meaning that the debugger does not need any
modification, tuning or parameter setting in order to work with a specific reasoner. This
means that the end user is able to select any DIG compliant reasoner that is appro-
priate for their needs, without worrying about loosing any of the functionality offered
by the debugger, or being tied to a specific reasoner implementation. For example, the
debugger can still be used in conjunction with a reasoner that offers non-standard rea-
soning services, or a reasoner that has been optimised to work with a specific style of ontology\(^1\).

2 Debugging process

Figure 1 illustrates the main steps of the debugging process. The user selects an OWL class for debugging, which is checked to ensure it is indeed inconsistent, and that the user is making a valid request to the debugger. The debugger then attempts to identify the unsatisfiable core for the input class in order to minimise the search space. The unsatisfiable core is the smallest set of local conditions (direct super classes) that leads to the class in question being inconsistent. Having determined the unsatisfiable core, the debugger attempts to generate the debugging super conditions, which are the conditions that are implied the conditions in the unsatisfiable core. Figure 3 presents the rules that are used in generating the debugging super conditions. The debugger then examines the debugging super conditions in order to identify the most general conflicting class set, which is analysed to produce an explanation as to why the class in question is inconsistent.

There are many different ways in which the axioms in an ontology can lead to an inconsistency. However, in general, we have found that most inconsistencies can be boiled down into a small number of ‘error patterns’. In summary the ‘error patterns’ for class inconsistency may be boiled down to following reasons:

The inconsistence is from some local definition.

1. Having both a class and its complement class as super conditions.
2. Having both universal and existential restrictions that act along the same property, whilst the filler classes are disjoint.
3. Having a super condition that is asserted to be disjoint with owl:Thing.
4. Having a super condition that is an existential restriction that has a filler which is disjoint with the range of the restricted property.

\(^1\) For example, a large ontology which is conceptually simple, or a small ontology which is very complex.
5. Having super conditions of $n$ existential restrictions that act along a given property with disjoint fillers, whilst there is a super condition that imposes a maximum cardinality restriction or equality cardinality restriction along the property whose cardinality is less than $n$.

6. Having super conditions containing conflicting cardinality restrictions.

The inconsistence is propagated from other source.

1. Having a super condition that is an existential restriction that has an inconsistent filler.
2. Having a super condition that is a hasValue restriction that has an individual that is asserted to be a member of an inconsistent class.

The debugger determines which of the above cases led to an inconsistency, and then uses provenance information that describes how the debugging super conditions were generated in order to determine the ‘root’ cause of the inconsistence.

Inconsistencies have ‘mechanisms’ of propagating throughout an ontology. For example, any concept that is inconsistent and is used in an existential restriction, causes that existential restriction to be inconsistent. This in turn causes the named class that the existential is a super class of to become inconsistent, which causes all subclasses of that named class to become inconsistent. These propagation mechanism can cause a single inconsistency in a densely interconnected ontology to make large swathes of the ontology inconsistent. The process of tracking down the root cause of such inconsistencies then becomes incredibly time consuming and frustrating. The debugger therefore has the functionality to trace an ontology subsumption hierarchy and existential graph in order to discover the root causes of unsatisfiable classes.

Figure 2 shows the debugging tool.

![Fig. 2. Debugging DeepPanBase class](image)

References

Rule 1: Named class rule
(a) IF $C_1 \in DSC(C) \land C_1 \subseteq C_2$, where $C_1$ is a named OWL class
    THEN $C_2 \in DSC(C)$
(b) IF $C_1 \in DSC(C)$ and $Disj(C_1, C_2)$, where $C_1$ and $C_2$ are named OWL classes
    THEN $\neg C_2 \in DSC(C)$

Rule 2: Complement class rule
(a) IF $\neg C_1 \in DSC(C)$, where $C_1$ is a named OWL class
    THEN IF $C_2 \subseteq C_1$, THEN $\neg C_2 \in DSC(C)$
        IF $C_1 \equiv C_2$, THEN $\neg C_2 \in DSC(C)$
(b) IF $\neg C_1 \in DSC(C)$, where $C_1$ is an anonymous OWL class
    THEN $\text{NORM}(C_1) \in DSC(C)$

Rule 3: Domain/Range rule
(a) IF $\exists S.C_1 \in DSC(C) \lor \geq n S \in DSC(C) \lor = n S \in DSC(C)$, where $n > 0$, and $\text{DOM}(S) = C_2$
    THEN $C_2 \in DSC(C)$
(b) IF $\exists S.C_1 \in DSC(C) \lor \geq n S \in DSC(C) \lor = n S \in DSC(C)$, and where $n > 0$, $\text{INV}(S) = S_1$ and $\text{RAN}(S_1) = C_2$
    THEN $C_2 \in DSC(C)$
(c) IF $\exists S.C_1 \in DSC(C) \lor \geq n S \in DSC(C) \lor = n S \in DSC(C)$, where $n > 0$, and $\text{RAN}(S) = C_2$
    THEN $\forall S.C_2 \in DSC(C)$

Rule 4: Functional/Inverse functional property
(a) IF $\exists S.C_1 \in DSC(C)$ or $\geq n S \in DSC(C)$ or $= n S \in DSC(C)$, where $n > 0$ and $S$ is functional
    THEN $\leq 1 S \in DSC(C)$
(b) IF $\exists S.C_1 \in DSC(C)$ or $\geq n S \in DSC(C)$ or $= n S \in DSC(C)$, where $n > 0$ and $\text{INV}(S) = S_1$, $S_1$ is inverse functional
    THEN $\leq 1 S \in DSC(C)$

Rule 5: Inverse Rule
IF $\exists S.C_1 \in DSC(C)$ and $\text{INV}(S) = S_1$, and $C_2 \sqsupseteq C_1$ and $C_2 \sqsubseteq \forall S_1 C_3$
THEN $C_3 \in DSC(C)$

Rule 6: Symmetric Rule
IF $\exists S.C_1 \in DSC(C)$ and $S$ is a symmetric property, and $C_2 \sqsupseteq C_1$ and $C_2 \sqsubseteq \forall S.C_3$
THEN $C_3 \in DSC(C)$

Rule 7: Transitive Rule
IF $\forall S.C_1 \in DSC(C)$ and $S$ is a transitive property,
THEN $\forall S \forall S.C_1 \in DSC(C)$

Rule 8: Intersection Rule
IF $C \land C_1 \in DSC(C)$,
THEN $C \in DSC(C)$ and $C_1 \in DSC(C)$

Rule 9: Subproperty Rule
(a) IF $\forall S.C_1 \in DSC(C)$ and $S_1 \sqsubseteq S$, THEN $\forall S_1 C_1 \in DSC(C)$
(b) IF $\leq n S \in DSC(C)$ and $S_1 \sqsubseteq S$, THEN $\leq n S_1 C_1 \in DSC(C)$
(c) IF $\exists S.C_1 \in DSC(C)$ and $S_1 \sqsupseteq S$, THEN $\exists S_1 C_1 \in DSC(C)$
(d) IF $\geq n S \in DSC(C)$ and $S_1 \sqsubseteq S$, THEN $\geq n S_1 \in DSC(C)$

Rule 10: Other inference Rule
IF $C_1$ can be inferred by any subset of $UC(C)$, where $C$ is a named class
THEN $C_1 \in DSC(C)$

Fig. 3. Rules for the membership of Debugging Super Conditions (DSC).