Black Box Debugging of Unsatisfiable Classes

Aditya Kalyanpur, Bijan Parsia, and Evren Sirin
aditya@cs.umd.edu, bparsia@isr.umd.edu, evren@cs.umd.edu

Maryland Information and Network Dynamics Laboratory,
University of Maryland,
College Park, MD 20740, USA

Abstract. Determining the cause of concept unsatisfiability in an ontology axiomatized in an expressive description logic is a difficult task. As the ontology grows larger and more complex, modelers may be driven to underconstrain their concepts in order to avoid problems. Part of the difficulty in correcting satisfiability bugs lies in the poverty of tool support – typically, reasoners only report that a class is unsatisfiable, not why. The traditional, if underexplored, solution is to generate explanations of the problem using some features of the internal state of the reasoner, so called glass box approaches. This requires reasoners to be modified to support the explanation, which often adversely affects performance. In this paper, we explore black box techniques for debugging unsatisfiable concepts expressed in the Web Ontology Language (OWL), that is, techniques which use the reasoner solely to determine which concepts are unsatisfiable, but use the asserted structure of the ontology to help isolate the source of the problems. We limit ourselves to two key debugging tasks, detecting dependencies between unsatisfiable concepts and pinpointing and explaining problematic axioms in the root concepts.

1 Introduction

Now that OWL is a W3C Recommendation, one can expect that a much wider community of users and developers will be exposed to the expressive description logic SHIF(D) and SHOIN(D) which are the basis of OWL-DL. These users and developers are likely not to have a lot of experience with knowledge representation (KR), much less logic-based KR, much less description logic based KR. For such people, having excellent documentation, familiar techniques, and helpful tools is a fundamental requirement.

A ubiquitous activity in programming is debugging, that is, finding and fixing defects in a program. Ontologies too have defects, and a common activity is to find and repair these defects. Unfortunately, the tool and training support for debugging ontologies is fairly weak.¹ We have chosen to focus on debugging unsatisfiable concepts (and contradictory ABoxes) because contradictions,

¹ While, historically, good KR modeling practices have been developed and described, often with an emphasis on description logics[7], tool support for “good” modeling remains elusive, especially given the lack of consensus on practice and the strong dependence of goodness on application and domain specifics.
in general, seem analogous to fatal errors in programs. Debugging fatal errors in programs can be relatively straightforward: the program crashes, there is a stack trace or similar information, and (one measure of) success is a running program. For ontologies, current tools (reasoners) do support indicating the dramatic failure of a unsatisfiable class, and success is similarly clear, however, the diagnosis and resolution of the bug is not supported at all. For example, no sets of support (axioms responsible) for an unsatisfiable class is given; or detection of inter-dependencies between unsatisfiable classes, such as a class directly depending on another for its unsatisfiability (say by an existential property restriction on an unsatisfiable class). We argue that explanations of such forms are essential for the purpose of debugging ontologies: while the former can be used to understand and rectify problematic axioms and class expressions, the latter can help prune out dependency bugs and let the modeler focus on the root (source) of the problem alone.

In [6], we investigated better support for debugging unsatisfiable concepts using both glass box (wherein the reasoner is modified to return explanations of the unsatisfiability) and black box (wherein the only inference service is satisfiability checking) techniques. In this paper, we extend our investigation of black box techniques which have two advantages over glass box ones: reasoner independence (you do not need a specialized, explanation generating reasoner) and avoiding the performance penalty of glass box techniques.

Furthermore, we focus our attention on two key tasks: detecting dependencies between unsatisfiable classes and presenting problematic axioms for a set of core classes.

2 Dependency between Unsatisfiable Classes

We define dependency between a pair of unsatisfiable classes as follows:

**Definition 1** (Dependency): Given a TBox \( T \), and a pair of atomic unsatisfiable classes \( A, B \in T \), \( A \) depends on \( B \) for its unsatisfiability if there exists a sub-TBox \( T' \subseteq T \) such that:

- both \( A \) and \( B \) are satisfiable in \( T' \) and
- \( T' \) either entails the subsumption of \( A \) by \( B \), i.e. \( T' \models A \sqsubseteq B \), or entails that every instance of \( A \) is related to an instance of \( B \) by some property chain, i.e. \( T' \models A \sqsubseteq \exists (p_0 \ldots p_n).B \)

In this case, we refer to \( B \) as the parent dependency of \( A \). The intuition here is that if \( B \) is unsatisfiable, \( A \) has to be unsatisfiable as well.

Also, if a class has at least one parent dependency, we refer to it as a derived unsatisfiable class, whereas if a class has no parent dependency, we refer to it as a root unsatisfiable class.
3 Automating Dependency Detection

In this section, we present an algorithm to detect dependencies between unsatisfiable classes in an ontology as provided by a reasoner. The algorithm consists of two parts: asserted dependency detection and inferred dependency detection and we describe each in detail in the following subsections.

For each unsatisfiable class in the ontology, the algorithm returns all its parent dependency classes along with the corresponding axioms that link this class to the parent. More specifically, the data structure returned is a set of tuples, where each tuple $\tau$ is recursively defined as:

$$\tau = (\tau, \text{axiom})$$

and the fixed point of $\tau$ is:

$$\tau = (\text{dep}, \text{axiom})$$

where, $\text{dep}$ is a set of dependency sets $\text{dep}'$, such that each $\text{dep}'$ is a set of unsatisfiable classes that together cause the bug (could be a singleton set), and $\text{axiom} = \text{associated axiom linking current class to dependency set}$

For example,

$$\tau(A) = (\{\{D\}, =\}, \{\{C, E\}, \subseteq\})$$

implies the following:

- $A$ has an equivalent class axiom relating it to its parent dependency $D$. This means only $D$ being unsatisfiable causes $A$ to be unsatisfiable, e.g. $A = D \cap (\leq 1p)$
- $A$ has a subclass axiom relating it to parents $C$ and $E$. This means both $C$ and $E$ should be unsatisfiable to make $A$ unsatisfiable, e.g. $A \subseteq (C \cup E)$

3.1 Asserted Dependency Detection: Structural Tracing

This phase is used to detect dependencies between unsatisfiable classes by analyzing the asserted structure of the ontology. It is divided into three stages:

- **Stage 1: Pre-processing** - given a class definition (considering its equivalence and subclass axioms), we obtain a set of all property-value chains inherent in these axioms, which terminate in a universal value restriction ($\forall$) on an unsatisfiable class. For example, consider the class definition $A$:

  $$A = \exists p.\forall q.(B \cap (\exists r.C \cup \forall s.D))$$

  If classes $B, D$ are unsatisfiable, the following property-value chains ($allPC$) are obtained for $A$:

  $$allPC = (p.q., B), (p.q.s., D_o).$$

  Note that:
• The class $D$ has a subscript 'o' to denote that a union (or non-deterministic branch) exists in its property chain ($p.q.s$), making this value restriction an ‘optional’ requirement (to be handled later in the post-processing stage).

• We also consider role hierarchy in constructing property-value chains, i.e., in the example above, if the property $p$ has ancestor properties $p_1, p_2$, we expand the chains in our allPC set to:

$$((p/p_1/p_2).q., B), ((p/p_1/p_2).q.s., D_o)$$

Universal value restrictions on a property must be satisfied iff the property exists i.e. the class definition entails a $\geq 1$ cardinality restriction on the property. In Stage 2 (dependency-tracing of a particular class), each time we discover the existence of a property, we check the allPC chains to ensure that the associated universal restriction is satisfied. However, note that we need to determine the set of allPC chains beforehand, since the non-localization of the class definition makes it difficult to verify all universal restrictions during tracing directly.

– Stage 2: Dependency-tracing - this is the core stage in which a recursive set of methods are used to extract all parent dependency unsatisfiable classes and the adjoining axioms given the original class definition. The output contains a mixture of definite and optional dependency cases.

While a detailed description of the algorithm is beyond the scope of the paper, we present the basic cases of the tracing approach.

Unsatisfiable class $A$ has a dependency if:

1. A (set of) equivalent/subClass axioms relate class $A$ directly to unsatisfiable class $B$ ($B$ becomes its parent)

2. A (set of) equivalent/subClass axioms relate class $A$ to an intersection set, any of whose elements are unsatisfiable, e.g. $A = (B \cap C \cap \ldots \cap D)$, and one of $B, C, \ldots, D$ is unsatisfiable (any such unsatisfiable class becomes its parent)

3. A (set of) equivalent/subClass axioms relate class $A$ to a union set, all of whose elements are unsatisfiable, e.g. $A = (B \sqcup C \sqcup \ldots \sqcup D)$, and all $B, C, \ldots, D$ are unsatisfiable (all such unsatisfiable classes become its parent)

4. A (set of) equivalent/subClass axioms entail that class $A$ has an existential ($\exists$) property restriction on an unsatisfiable class, e.g. $A = \exists(p, B)$ and $B$ is unsatisfiable ($B$ becomes its parent)

5. A (set of) equivalent/subClass axioms entail that class $A$ has a ($\geq 1$) cardinality restriction on a property-chain, and the universal ($\forall$) value restriction on that chain is not satisfied (object/value of property chain becomes its parent)

6. A (set of) equivalent/subClass axioms entail that class $A$ has a ($\geq 1$) cardinality restriction on a property, and the domain of the property is unsatisfiable, e.g. $A \subseteq (\geq 1p), domain(p) = B$, and $B$ is unsatisfiable, making it the parent of $A$ (similar domain check has to be made for every ancestor property of $p$)
7. A (set of) equivalent/subClass axioms entail that class $A$ has a ($\geq 1$) cardinality restriction on an object property, and the range of its inverse is unsatisfiable, e.g. $A \sqsubseteq (\geq 1p), \text{range}(p^-) = B$, and $B$ is unsatisfiable, making it the parent of $A$ (similar range check has to be made for every ancestor property of $p^-$).

Lemma 1 Parent dependencies detected in this stage satisfy Definition 1 of dependency

Proof 1 For an unsatisfiable class $A$, we find a sub-TBox, $T'$, (i.e. set of axioms in the KB) which either entails its subsumption by another unsatisfiable class $B$ or entails the existence of $B$ given $A$ via cardinality restrictions on some property(s). Also, based on the manner in which we find $T'$, both, $A$ and $B$ are clearly satisfiable in $T'$.

Note that the tracing approach described above does not consider nominals or transitive property relations. Hence, it can be considered as sound (i.e. it finds accurate dependencies between unsatisfiable classes), but not complete (does not find all dependencies).

Stage 3: Post-processing - if the final dependency set contains an optional unsatisfiable class $C_o$, we check if adjoining (siblings within the same nested set) dependencies are definite i.e. without an ’o’ tag. If they are, we simply remove the optional dependency $C_o$; else (if $C_o$ is the sole dependency) we transform it to a $C$ getting rid of the optional tag. We do this recursively, until all the optional unsatisfiable classes are either pruned out or transformed in the final dependency set.

For example, if the final dependency set for a class is:

$$\text{dep} = (\{ \{ C_o, D \}, = \}, \{ \{ E_o \} \sqsubseteq \})$$

we reduce it to:

$$\text{dep} = (\{ \{ D \}, = \}, \{ \{ E \} \sqsubseteq \})$$

Optional classes are unsatisfiable class dependencies occurring in a disjunction (union set). Thus, they are guaranteed to cause unsatisfiability if and only if they are the sole reason for it, else they need to be pruned out of the final dependency set.

For detailed examples of structural tracing, see Appendix A.

Drawbacks of Tracing One of the main drawbacks of the structural tracing algorithm is that it does not consider inferred equivalence or subsumption between classes. Consider two classes $A$ and $B$ that do not have an explicit subsumption relation between them but the reasoner can infer one, e.g. $A = (\geq 1p)$ and $B = (\geq 2p)$. Even though there is no subclass axiom relating the two classes,
a reasoner can infer that $B \sqsubseteq A$ (provided of course, that both are satisfiable). However, the tracing algorithm cannot find the hidden dependency of $B$ on $A$. In this case, even using a reasoner to infer the subsumption relation will not work as both classes are unsatisfiable and hence equivalent to the bottom concept (and each other). As a result, we need an alternate way to detect such hidden dependencies.

### 3.2 Inferred Dependency Detection

The problem with detecting hidden dependencies in an incoherent KB (TBox) is that the unsatisfiability masks some useful subsumption relationships in the TBox. Hence, given a TBox with unsatisfiable concepts, we consider a subsumption safe transformation as one which modifies the TBox by trying to get rid of all inconsistencies while attempting to preserve the intended subsumption hierarchy as much as possible. Here, we present only a heuristic approach that seems to work well in practice.

The heuristic consists of two steps:

1. For every axiom in the TBox that refers to a class of the form $\neg C$, where $C$ could be atomic or complex, we replace $\neg C$ by a new atomic class $C'$.
2. Similarly, for every axiom in the TBox that refers to a class of the form $\leq n.p$, where $p$ is an OWL property, we replace $\leq n.p$ by a new atomic class $P'$.

There is an important aspect of the algorithm above which attempts to preserve subsumption relations in the underspecified KB, that is: every substitution is stored in a cache, and each time a new one is made, we check for subsumption (using a reasoner) between satisfiable terms in the original KB, and if found, add corresponding relations between concepts in the transformed KB. For example, consider a TBox with the following 3 axioms:

\[
(ax_1) \quad A = \neg C \sqcap \leq 1.p \\
(ax_2) \quad B = \neg D \sqcap \leq 2.p \\
(ax_3) \quad D \sqsubseteq C
\]

Here, we substitute $\neg C$ by $C'$ and $\neg D$ by $D'$. Now, because of the subsumption of $D$ by $C$ in the original KB, we add the axiom $C' \sqsubseteq D'$ to the transformed KB. Similarly, after substituting $\leq 1.p$ by $P_1$ and $\leq 2.p$ by $P_2$, due to the subsumption relation $\leq 1.p \sqsubseteq \leq 2.p$, we add the axiom $P_1 \sqsubseteq P_2$ to the transformed KB.

The motivation for the above approach is to remove well-known causes for concept unsatisfiability, i.e. class complements and max cardinality violations on a property as discussed in [6]. In fact, since both steps are independent, we perform any one step first and test subsumption before moving to the other.

Note that the above procedure is always sound since the monotonic nature of the logic (OWL-DL) implies that removing an axiom from the KB (or underspecifying it in the manner in which we have by reducing expressivity) cannot add a
new subsumption relation. Though the procedure is incomplete because safety cannot be guaranteed and the original subsumption relation may get destroyed.

After applying the above transformation, we can use the reasoner to classify the (relevant part of) KB to detect hidden dependencies between concepts not caught in structural tracing. Moreover, if the concepts turn out to be satisfiable, and hidden dependencies are revealed, we can use the transformations performed to pinpoint axioms which cause incoherence in the core (parent) unsatisfiable classes.

To elaborate, consider a simple TBox with two unsatisfiable classes $A, B$, and the following axioms:

$$(a_1) \ A = D \cap \exists p.D$$
$$(a_2) \ A \sqsubseteq \neg D$$
$$(a_3) \ B = C \cap \exists p.C$$
$$(a_4) \ C \sqsubseteq D$$

In this TBox, the unsatisfiability masks the subsumption of $B$ by $A$ (i.e. $B \sqsubseteq A$). If, however, we apply step 1 of the transformation above and reduce the TBox to a consistent form by modifying axiom $a_2$ to $A \sqsubseteq D'$, the new TBox $T' \models B \sqsubseteq A$, and the hidden dependency is revealed. Moreover, we also know that axiom $a_2$ is an integral part of the problem.

## 4 Evaluation: The Tambis Ontology

For the purpose of evaluation, we needed an expressive, moderately-sized OWL ontology that had a large number of unsatisfiable classes. The Tambis ontology [1] fit our needs well - its an OWL DL ontology containing 395 Classes and its expressivity is SHIN. Moreover, the OWL version 2 was generated by a conversion script and a number of errors crept in during that process – 144 unsatisfiable classes in all. Many of the unsatisfiable classes depend in simple ways on other unsatisfiable classes, so that a brute force going down the list correcting each class in turn is unlikely to produce correct results, or, at best, will be pointlessly exhausting. In one case, three changes repaired over seventy other unsatisfiable classes. Given the highly non-local effects of assertions in a logic like OWL, it is not sufficient to take on defects in isolation.

We implemented the black-box techniques in the Swoop OWL ontology editor [3], used the default DL Tableaux Reasoner, Pellet [5], and carried out the analysis and debugging of Tambis. Running the structural tracing algorithm on Tambis identified 111 *derived* classes with at least one parent dependency, and 33 classes with no parent dependencies (potential *root* classes). This was a significant result, the problem space was pruned by more than 75% enabling us to direct our attention on a narrow set of unsatisfiable classes, and moreover, for each unsatisfiable class, we obtained the dependency relation (via axioms) leading to its corresponding parents, which were presented in the Swoop UI.

---

2 http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/tambis-full.owl
Out of the remaining 33 potential root classes, we applied the inferred dependency detection algorithm and uncovered equivalence between all of them. This was both, a surprising and interesting result, and due to the fact that all 33 classes shared the same structure (defined equivalent to the same intersection set) out of which 3 were asserted as mutually disjoint in the ontology thus causing the contradiction; while the remaining 30 classes were inferred to be equivalent to the above 3 classes making them unsatisfiable as well. In this case, not only were hidden dependencies detected, but the disjoint axioms causing the incoherence were obtained as well, exposing the classes whose definition contained the axioms (in this case, the classes metal, non-metal, metalloid).

As a result, we now have an efficient iterative procedure to remove all the unsatisfiability bugs in the ontology: at each point, we focus solely on fixing all the root classes (if any), or top-level parent classes identified by the pinpointed axioms, which effectively fixes a large set of directly derived class bugs. However, doing so might reveal additional contradictions and a new set of unsatisfiable classes. We then use the dependency detecting algorithm again to obtain new potential roots (with problematic axioms) and repeat the fixing process iteratively till no unsatisfiable classes are left in the ontology.

5 Explaining Sets of Support for Root Classes

Having identified the core set of unsatisfiable classes in the ontology, we now focus on pinpointing the axioms leading to the contradiction of such classes.
Moreover, since our emphasis is on explaining the circumstances leading to the contradiction, we present the axioms in a particular order to aid understanding of the problem better.

We illustrate this method with an example. Consider the following TBox containing 8 axioms:

1. \( A \sqsubseteq B \cap D \)
2. \( B \sqsubseteq \exists p.\neg H \)
3. \( C \sqsubseteq \exists p.\neg F \)
4. \( D \sqsubseteq (\leq 1)p \)
5. \( E \sqsubseteq \forall p.G \cap H \)
6. \( G \sqsubseteq F \)
7. \( \text{domain}(p) = E \)
8. \( \text{range}(p) = C \)

The TBox \( (T) \) has two interesting properties:

1. Only class \( A \) is unsatisfiable
2. \( A \) is satisfiable in any sub-TBox, \( T' \subset T \). Thus, the above set of axioms is the minimal TBox that captures the contradiction in \( A \), or in other words, \( MUPS(A) \) as defined in [8].

The above example shows that even identifying the \( MUPS \) of a class, which itself is a non-trivial task, does not make it obvious why the class is unsatisfiable. Thus we must take into account the interaction among the axioms, which in turn can be used to explain the cause for the unsatisfiability.

At this point, we revert back to a glass box technique described in our earlier work [6] in which we obtain the sets of support for an unsatisfiable class (above 8 axioms in this case) and also display \textit{clash information} of the form: ‘Class \( A \) is unsatisfiable because it is a subclass of both, classes \( X \) and \( Y \), which in the above case amounts to the following explanation: \( A \sqsubseteq \exists p.\neg F \cap \forall p.F \). Again, while this is a step in the right direction from a debugging point of view, it still does not tell us \textit{how} the specific subsumption relation shown above came about. Now using this feature as a starting point, we present a black box heuristic approach to explain the cause of the contradiction in two steps:

- **Step 1.** For each satisfiable element of the pair \( (X,Y) \) that is responsible for the clash, we obtain a set of all descendant subclasses (using the reasoner). In this case, we get the following two subsumption paths:
  
  \[ B \sqsubseteq C \sqsubseteq \exists p.\neg F \]
  
  \[ D \sqsubseteq E \sqsubseteq \forall p.F \]

  We can then prefix the relation \( A \sqsubseteq .. \) before each path to complete the trace diagram as follows:

  \[ A \sqsubseteq B \sqsubseteq C \sqsubseteq \exists p.\neg F \]
  
  \[ A \sqsubseteq D \sqsubseteq E \sqsubseteq \forall p.F \]

  Already, such a trace is useful for leading the ontology debugger to the contradiction.
Step 2. We overlay the subsumption trace diagram with axioms, i.e. for each subsumption link between a pair of classes, we display the axioms that entail the subsumption relation. For this purpose, we define the notion of MSPS:

**Definition 2** (MSPS($C \sqsubseteq D$)): Given an ordered pair of atomic classes, $C$ and $D$ in a TBox, $T$, the Minimal Subsumption Preserving Sub-TBox written as MSPS($C \sqsubseteq D$) is the smallest sub-TBox $T'$ of $T$ ($T' \subseteq T$), which entails the subsumption of $C$ by $D$, i.e., $T' \models C \sqsubseteq D$.

Note that we can easily relate the MUPS of a class to the MSPS for a pair of classes in the TBox.

**Lemma 2** Given a pair of satisfiable atomic classes in a TBox: $C$, $D$, $\text{MSPS}(C \sqsubseteq D) = \text{MUPS}(C \sqcap \neg D)$.

**Proof 2** By definition of MUPS [8], $\text{MUPS}(C \sqcap \neg D)$ is the smallest sub-TBox, $T'$ which entails the contradiction of the concept $C \sqcap \neg D$. A TBox entails the subsumption of $C$ by $D$ iff it entails the contradiction of $C \sqcap \neg D$. Thus the sub-TBox, $T'$, necessarily entails $C \sqsubseteq D$. Moreover, since MUPS is minimal, $T'$ is the smallest sub-TBox which captures this subsumption, satisfying the definition of MSPS($C \sqsubseteq D$). Hence, $\text{MSPS}(C \sqsubseteq D) = \text{MUPS}(C \sqcap \neg D)$.

Using the above technique, we can find the MSPS for a pair of satisfiable classes in the subsumption trace, provided we have the MUPS. For now though, we use our earlier glass box approach for sets of support which is simply an approximation of the MUPS.

Note that for the unsatisfiable class, we need to look at the asserted structure of the ontology alone. Now, we could either use the top-down approach of inferred dependency detection (Section 3.2) to capture the axiom set in a consistent KB, or a bottom-up approach as follows: we obtain the asserted super classes of the unsatisfiable class and find subsumption (and thus sets of support) between a satisfiable superclass (atomic or complex) and the low-hanging classes above using the reasoner. In this example, the solution is direct because of the axiom $A \sqsubseteq B \sqcap D$, which makes $A$ a direct subclass of both $B$ and $D$. We are currently working on heuristics to solve this efficiently, but in most cases that we have seen, the link is direct making the solution easy to obtain.

The final subsumption trace with axiom sets overlaid looks like this (→ represent subclass relationships):

1. $A \rightarrow B$ (2,8)
2. $C \rightarrow \exists p. \neg F$
3. $A \rightarrow D$ (4,7)
4. $E \rightarrow \forall p. F$

Such a trace makes the cause for the contradiction in the unsatisfiable class clearer by systematically leading the ontology debugger to the source of the problem, at each point revealing only the relevant axioms which further it along the trace (see Figure 2 for a Tambis example).
6 Related Work

To our knowledge, black-box debugging to find and explain dependencies between unsatisfiable classes is a largely unexplored topic. The closest work we know of is [8], who propose non-standard reasoning algorithms (for ALC TBoxes) based on minimization of axioms using Boolean methods, and demonstrate promising results on the DICE terminology. Their approach which deals with axiom and concept pinpointing is directly related to our work, though they restrict their scope to unfoldable ALC TBoxes and rely on glass box techniques as well. On the other hand, we develop black box approaches which are independent of the logical complexity of the model (only caveat is that a DL reasoner has to be able to process the model). Moreover, while their focus is axiom-driven, we take a class-driven approach and define and detect dependencies between unsatisfiable classes.

The second technique we describe, i.e. explaining unsatisfiability of a class from the sets of support axioms, inherently involves explaining subsumption. There is plenty of literature on subsumption explanation ([4], [2]). However, we differ in three respects: first our context and hence methodology for capturing axioms that cause subsumption is different since we rely on identifying the MUPS of an unsatisfiable class; second, we attempt to capture subsumption in a narrow context for a class that may be unsatisfiable in the broader context (heuristic

---

### Fig. 2. Explaining Unsatisfiability

Presenting axioms leading to contradiction for the unsatisfiable class Protein in the Tambis Ontology. Protein becomes a top-level parent class in the second debugging iteration.

<table>
<thead>
<tr>
<th>OWL Class: protein</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsatisfiable concept</td>
</tr>
<tr>
<td>Reason: An individual belongs to a type and its complement</td>
</tr>
<tr>
<td>Reason: Individual auto-generated individual for protein</td>
</tr>
<tr>
<td>Reason: Details: Individual contains both nucleic acid and its complement</td>
</tr>
</tbody>
</table>

**Axioms causing the problem:**

1. nucleic $\neq$ nucleic acid
2. protein $\neq$ ca
3. ca $\neq$ (Nucleoside.of, nucleoside) $\land$ (Pyrimidine.of, nucleoside) $\land$ macromolecular compound
4. nucleic $\neq$ (Nucleoside.of, nucleoside) $\land$ (Pyrimidine.of, nucleoside) $\land$ macromolecular compound
5. de-nucleotide $\neq$ nucleic acid

Intersection of: (add)
- Nucleoside.of, amino-acid (Class)
- Pyrimidine.of, amino-acid (Class)
- Nucleotide.of, macromolecular (Class)

Equivalent to: (add)
- ca (Class)
- Nucleic acid (Class)

Disjoint with: (add)
- nucleic acid (Class)

Annotations: (add)
- Union of: (add)
approach); and third, the primary goal driving our presentation is explaining the contradiction.

7 Conclusion & Future Work

In the black box debugging techniques described in this paper, we primarily focus on separating the root from the derived unsatisfiable classes allowing the modeler to focus solely on the problematic parts of the ontology. As a subsequent step, we are working on explaining unsatisfiability for the root classes from the sets of support axioms.

Evaluation performed on the Tambis OWL ontology, which contains 144 unsatisfiable classes, has given us promising preliminary results. The structural tracing algorithm helped prune out a large chunk of unsatisfiable classes (111/144) based on simple (direct), as well non-local (indirect) dependencies on parent classes; while inferred dependency detection helped reveal equivalence between all 33 potential roots. Additionally, the sets of support explanations for the core classes helped pinpoint relevant axioms leading to the contradiction, making it easier to understand and debug the problem.

The Tambis use case scenario is quite reasonable, given that automated scripts for converting ontologies or schema to OWL are likely to introduce errors, and modelers keen on using the rich expressivity of OWL-DL require good debugging support to fix them.

As future work, we plan on extending structural tracing to capture more derived dependencies, and on optimizing the inferred dependency detection algorithm. Finally, we are working on heuristics for black box determination of MUPS and MSPS as noted in Section 5.

References


Appendix A

We consider a few cases to explain the working of the entire structural tracing algorithm (especially highlighting the significance of the pre- and post-processing stages). In all cases below, A, B, C, D are named OWL classes (as opposed to complex class expressions) and p,q are OWL properties.

- **Case 1**: Given two axioms defining class A: (suppose A, B, C, D all unsatisfiable)
  
  \( (i) A \sqsubseteq ((\forall p.C) \cap B \cap (\exists q.D)) \)
  
  \( (ii) A \sqsubseteq ((\geq 1 p) \cup D) \)

  After pre-processing, we get the following \( \forall \) property-value chains:
  
  \( allPC = \{ (p,C) \} \)

  During structural tracing of axiom \((i)\), we obtain the following dependencies
  
  \( \{\{B\}, \{D\}, (i)\} \) (last element being the axiom pointer).

  B appears independently as its unsatisfiable and part of the intersection set, and D appears independently because the existential property restriction on q forces the unsatisfiable class D to be A’s successor. Note that all \( \forall p.C \) restrictions are ignored in this stage (read below to see how they are accounted for).

  Structural tracing of axiom \((ii)\) returns \( \{\{C, D\}, (ii)\} \). The \{C\} appears when the \( \forall \) restriction on p (as stored in \( allPC \)) is checked because of the \( (\geq 1 p) \) restriction, and D appears since its part of the union set. Note here that both C and D appear together in one set (as opposed to two independent elements as in the previous axiom) because union semantics of axiom \((ii)\) imply that A is unsatisfiable if both C and D are unsatisfiable.

  In the post-processing stage, no additional transformations take place on the dependency sets because there are no optional dependencies to consider.

  Thus the final dependency sets for A are \( \{\{B\}, \{D\}, (i)\} \) and \( \{\{C, D\}, (ii)\} \), which imply that fixing A requires correcting the unsatisfiability bugs in B and D. You don’t need to correct C since its part of the union set with D.

- **Case 2**: Given two axioms defining class A: (suppose A, B, C, D all unsatisfiable)
  
  \( (i) A \sqsubseteq ((\geq 1 p) \cap B) \)
  
  \( (ii) A \sqsubseteq ((\forall p.C) \cup (\forall q.D)) \)
After pre-processing, we get the following ∀ property-value chains:

\[ allPC = \{(p.C_o), (q.D_o)\} \] (the 'o' subscript denoting that the chain is optional as it terminates inside a union)

During structural tracing of axiom (i), we obtain the following dependencies \{\{C_o\}, \{B\}, (i)\}. The \{C_o\} appears when the ∀ restriction on \(p\) (as stored in allPC) is checked because of the \(\geq 1p\) restriction, and \(B\) appears directly as its part of the intersection set. Structural tracing of axiom (ii) returns nothing since ∀ restrictions are basically ignored in this stage.

In the final post-processing stage, since \(C_o\) has a sibling \(B\) which is a definite root, we remove \(C_o\) from the dependency set giving us just \{\{B\}, (i)\}. This is consistent with the model, since fixing the unsatisfiability of \(B\) will make class \(A\) satisfiable as well.

- **Case 3**: Given two axioms defining class \(A\): (suppose \(A, C, D\) are unsatisfiable)
  
  \[(i) A \sqsubseteq ((\geq 1p) \cap (\geq 1q))\]
  
  \[(ii) A \sqsubseteq ((\forall p.C) \sqcup (\forall q.D))\]

  After pre-processing, we get the same ∀ property-value chains as in Case 2:
  
  \[ allPC = \{(p.C_o), (q.D_o)\} \]

  During structural tracing of axiom (i) we obtain the following dependencies \{\{C_o\}, \{D_o\}, (i)\} for similar reasons as explained in Case 2. Again structural tracing of axiom (ii) returns nothing since ∀ restrictions are basically ignored in this stage.

  In the final post-processing stage, since neither \(C_o\) nor \(D_o\) have definite root siblings we randomly remove the optional tag on any root, say \(C\), and now since \(D_o\) has a definite root sibling, we remove it from the dependency set giving us \{\{C\}, (i)\}. This is consistent with the model, since fixing the unsatisfiability of either \(C\) (or \(D\) as the case maybe) will make class \(A\) satisfiable as well.