Black Box Techniques for Debugging Unsatisfiable Concepts

Aditya Kalyanpur, Bijan Parsia, Evren Sirin
University of Maryland
College Park, USA
`aditya@cs.umd.edu`, `bparsia@isr.umd.edu`, `evren@cs.umd.edu`

June 22, 2005

1 Motivation

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, 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. Current tools do support indicating the dramatic failure of a unsatisfiable class, and success is similarly clear, however, supporting the diagnosis and resolution of the bug is not supported at all. In [2], we investigated better support for debugging unsatisfiable concepts using both glass box (wherein the reasoner is

[1]While, historically, good KR modeling practices have been developed and described, often with an emphasis on description logics[3], 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.

modified 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.

2 Root and Derived Unsatisfiabilities

We categorize unsatisfiable classes into two types:

1. *Root Class* - this is an unsatisfiable class in which a clash or contradiction found in the class definition (axioms) does not **depend on the unsatisfiability** of another class in the ontology. More specifically, the unsatisfiability bug for a root class cannot be fixed by *simply* correcting the unsatisfiability bug in some other class, instead, it requires fixing some contradiction stemming from its own definition. Root classes can be further subdivided into two categories:

   (a) *Pure Root*: This is a (root) class whose unsatisfiability depends strictly on its own definition (axioms), and not on the unsatisfiability of any other (root) class in the ontology. For example: $Student \sqsubseteq (\geq 2) hasAdvisor \cap (\leq 1) hasAdvisor$

   (b) *Mutually Dependent Root*: This is a (root) class whose unsatisfiability depends on one or more distinct (root) classes in the ontology. For example: if you have two classes $Male$ and $Female$, which are asserted to be disjoint from one other, but also incorrectly defined as equivalents. In this case, both classes are mutually dependent roots with the problem lying in the equivalence and disjoint axioms.

2. *Derived Class* - this is an unsatisfiable class in which a clash or contradiction found in a class definition either directly (via explicit assertions) or indirectly (via inferences) **depends on the unsatisfiability** of another class (we refer to it as the *parent* unsatisfiable class). Hence, this is a less critical bug in the sense that (in most cases) it can be resolved by fixing the unsatisfiability of this parent dependency. Note that there may be cases in which fixing the dependency bug reveals yet another unsatisfiability bug in the class, which needs to be resolved separately (making the derived class necessarily, but not sufficiently, dependent on the parent). Example of a derived class is: Class $GraduateStudent \sqsubseteq Student$, where $Student$ is an unsatisfiable class itself, in this case, its parent.

3 Automating root class discovery

In this section, we present an algorithm to separate the root unsatisfiable classes from the derived ones given the total set of unsatisfiable classes in an ontology as provided by a reasoner. The algorithm consists of two parts: **Structural Tracing** and **Root Pruning** and we describe each in detail in the following subsections.

For each unsatisfiable class in the ontology, the algorithm obtains its dependencies as follows:

- For a *root* class, it returns itself.

- For a *derived* class, it returns all the parent dependency classes along with the corresponding axioms that link the derived class to the parent. More specifically, the data structure returned is a set of tuples, where each tuple τ is recursively defined as:
  τ = (τ, *axiom*), and the fixed point of τ is:
  τ = (*dep*, *axiom*), where,
  dep is a set of dependency sets *dep′*, such that each *dep′* is a set of parent unsatisfiable classes that together cause the bug (could be a singleton set), and
  axiom = associated axiom linking current class to dependency set

  For example,
  $\tau(A) = (\{\{D\}, equClaAxiom\}, \{\{C, E\}, subClaAxiom\})$
  implies the following:
  A has an equivalent class axiom relating it to parent dependency D i.e. (*solely* D) is unsatisfiable makes A unsatisfiable. E.g., $A = D \cap (\leq 1p)$
  A has a subclass axiom relating it to parents (C and E) i.e. (*both* C and E) are unsatisfiable makes A unsatisfiable. E.g., $A \sqsubseteq (C \cup E)$

3.1 Structural Tracing

The tracing algorithm is used to determine structural dependencies between unsatisfiable classes. The details of the algorithm are given in [1]. For now, we briefly enumerate its 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. The intuition for this is as follows: universal value restrictions on a property must be satisfied iff the property exists i.e. the class definition entails a ≥ 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 - a recursive set of methods are used to extract all dependency unsatisfiable classes and the adjoining axioms given the original class definition. The output contains a mixture of definite and optional dependency cases. The basic tenets of the tracing approach are as follows:
Class A is a derived unsatisfiability if:

1. A is equivalentTo/subClassOf an intersection set, *any* of whose elements are unsatisfiable, i.e., $A = (B \cap C.. \cap D)$, and one of B,C..D is unsatisfiable (any such unsatisfiable class becomes its parent)

2. A is equivalentTo/subClassOf a union set, *all* of whose elements are unsatisfiable, i.e., $A = (B \cup C.. \cup D)$, and all B,C..D are unsatisfiable (all such unsatisfiable classes become its parents)

3. A has an existential ($\exists$) property restriction on an unsatisfiable class, i.e., $A = \exists(p, B)$ and B is unsatisfiable (B becomes its parent)

4. A entails a (≥ 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)

5. A entails a (≥ 1) cardinality restriction on a property, and the domain of the property is unsatisfiable, i.e., $A \sqsubseteq (\geq 1p), domain(p) = B$, and B is unsatisfiable, making it the parent of A (similar *domain* check has to be made for every ancestor property of p)

6. A entails a (≥ 1) cardinality restriction on an object property, and the range of its inverse is unsatisfiable, i.e., $A \sqsubseteq (\geq 1p), range(p^{-}) = B$, and B is unsatisfiable, making it the parent of A (similar *range* check has to be made for every ancestor property of p^{-})

Stage 3: Post-processing - In this stage, all the optional unsatisfiable classes or parent dependencies that may have been introduced in the previous stages are either pruned out or transformed (to necessary dependencies) in the final dependency set.

Detailed examples of the structural tracing algorithm are given in [1].

3.1.1 Drawbacks of Tracing

One of the main drawbacks of the structural tracing algorithm is that it does not consider *inferred* equivalence or subsumption between root 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, if the tracing algorithm returns both classes as root, it 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 effectively equivalent to the bottom concept. As a result, we need an alternate way to prune the root classes further by finding these hidden dependencies.

3.2 Pruning potential roots

We are working on a simple, but optimized, brute force approach to detect hidden dependency (equivalence and/or subsumption) between potential root classes ($root_p$) found at the end of the structural tracing. It involves removing a set of $roots_p$ from the ontology and testing the satisfiability of the remaining $roots_p$. Before we present the details of this approach, we need to specify what we mean by *removing* a class from the ontology: here, removal of class C implies getting rid of all definitions of the class from the ontology, i.e., axioms in which C appears solely on the LHS.

3.2.1 Brute-force Pruning

Considering the following example. Suppose structural tracing identifies the following potential root classes : $\{R_1, R_2, R_3, C_1, C_2, C_3, C_4\}$ such that $\{R_1, R_2, R_3\}$ are the actual roots (each mutually dependent) and $\{C_1, C_2, C_3, C_4\}$ are derived from the roots based on the following (non-asserted) dependencies:
$C_3 \sqsubseteq C_1 \sqsubseteq C_4 \sqsubseteq R_1$
$C_3 \sqsubseteq R_2$
$C_2 \sqsubseteq R_2$
$C_4 \sqsubseteq R_3$

The detailed algorithm for iteratively pruning the actual roots is given in [1]. Here, we briefly explain how it works: One iteration of the algorithm takes a set of unsatisfiable classes and identifies a single root in it by gradually reducing the set to one in which all classes are satisfiable.

Thus, we have the following condition: given a set of unsatisfiable classes L, we split it into two parts, L_1 and L_2, such that:

1. removing L_1 from the ontology makes L_2 satisfiable, and
2. L_1 is minimal, i.e., there exists no other partitioning of the list $L = \{L_1', L_2'\}$ satisfying (1) such that $L_1' \leq L_1$

Now, if R is the class at the boundary of the partition, i.e., R is the last class to be added to L_1 to satisfy the above two conditions, then we can infer the following:

1. R is a root class since removing it (along with the previously removed classes in L_1) from the ontology makes L_2 completely satisfiable. Note: L_1 (which contains R) should be minimal for this to hold.
2. Conversely, each class in L_2 is a derived class which has atleast one root dependency: R (there may be additional dependencies on other classes in L_1)

Since the above root identification process is generic for any list of unsatisfiable classes, we can remove the root R from the ontology and repeat the above process iteratively to prune out all the roots.

An important optimization in the algorithm above is to track *dependent* sets dynamically, i.e., if during an intermediate partitioning stage, we find that a class becomes consistent, we tag it as a dependent class and skip it during later partitioning stages, thereby preventing numerous unnecessary satisfiability checks.

To see how the algorithm works with the above case, consider an initial (random) ordering of potential roots: $L = \{C_4, R_3, C_1, R_1, C_3, R_2, C_2\}$. The iterations of the root pruning process are shown in the table below. The total no. of satisfiability tests needed to identify all three roots $\{R_1, R_2, R_3\}$ is 11.

Unsat. Class List / Partitions	**Roots**	**Dependent**
$L = \{C_4, R_3, C_1, R_1, C_3, R_2, C_2\}$ $L_1 = \{C_4, R_3, C_1, R_1, C_3\} + \{R_2\}$ $L_2 = \{C_2\}$ No. of Sat Tests = 6	R_2	C_1, C_2
$L = \{C_4, R_3, R_1, C_3\}$ $L_1 = \{C_4, R_3\} + \{R_1\}$ $L_2 = \{C_3\}$ No. of Sat Tests = 3	R_1	C_3
$L = \{C_4, R_3\}$ $L_1 = \{C_4\} + \{R_3\}$ $L_2 = \{\}$ No. of Sat Tests = 1	R_3	-
$L = \{C_4\}$ No. of Sat Tests = 1	-	C_4

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 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, and carried out the analysis and debugging of Tambis. Running the structural tracing algorithm on the unsatisfiable classes in Tambis identified 111 derived classes and 33 potential roots in $\approx 20ms$. 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 unsatisfiabile classes, and moreover, for each derived unsatisfiabile class, we obtained the dependency relation (via axioms) leading to its corresponding roots, which were presented in the Swoop UI.

Out of the remaining 33 potential roots, we applied the brute-force root pruning technique (which performed an additional 51 satisfiability checks in $\approx 11s$) and reduced the root set to just 2! This was both, a surprising and interesting result, and due in fact to the inferred equivalence between a large number of potential roots. The 33 classes all shared the same structure (defined equivalent to the same intersection set) out of which 2 were actual roots, (any pair from the set {`metal`, `nonmetal`, `metalloid`} [4]), each asserted as mutually disjoint in the ontology thus causing the contradiction; while the remaining 30 classes were all inferred to be equivalent to the above 3 classes making them unsatisfiable as well. Thus, root pruning was able to reduce the problem space of the ontology even further, demonstrating the effectiveness of black-box techniques for debugging ontologies.

The next step we intend working on is pinpointing problematic axioms in the ontology, a problem which is simplified now that we have identified the small set of root unsatisfiable classes.

[2]http://www.cs.man.ac.uk/ horrocks/OWL/Ontologies/tambis-full.owl

[3]http://www.mindswap.org/2004/SWOOP

[4]Fixing any pair of unsat. classes from the set makes the third class satisfiable

5 Related Work and Conclusion

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 [4], 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 related to our work, though they rely on glass box techniques as well.

The black box debugging techniques described in this paper focus on separating the root from the derived unsatisfiable classes allowing the modeler to focus solely on the problematic parts of the ontology. Evaluation performed on the Tambis ontology 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 root classes; while further root pruning (30/33) helped reveal hidden equivalence/subsumption between potential roots. The Tambis use case scenario is quite reasonable, given that automated scripts for converting ontologies or schemas 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. The debugging techniques are also useful to reveal dependencies during iterative building of an ontology when highly non-local interactions in the ontology cause sudden and unexpected erroneous results, e.g., when adding a new axiom makes some atomic class equivalent to `owl:Thing` causing numerous classes to become unsatisfiable due to disjointness.

References

[1] A. Kalyanpur, Parsia B, and E. Sirin. Detecting dependencies between unsatisfiable classes. http://www.mindswap.org/papers/depunsat.pdf. 2005.

[2] Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Debugging owl ontologies. In *The 14th International World Wide Web Conference (WWW2005)*, Chiba, Japan, May 2005.

[3] A. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe. Owl pizzas: Common errors & common patterns from practical experience of teaching owl-dl. *In European Knowledge Acquisition Workshop (EKAW)*, 2004.

[4] S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. *Proceedings of IJCAI, 2003*, 2003.