Proceedings of I-KNOW '07
Graz, Austria, September 5-7, 2007

Incremental Approach to Error Explanations in Ontologies

Petr Křemen, Zdeněk Kouba
(Czech Technical University in Prague, Czech Republic
{kremen,kouba}@labe.felk.cvut.cz)

Abstract: Explanations of modeling errors in ontologies are of crucial importance both when creating and maintaining the ontology. This work presents two novel incremental methods for error explanations in semantic web ontologies and shows that they have significantly better performance than the state of the art black-box techniques. Both promising techniques together with our implementation of a tableau reasoner for an important OWL-DL subset $\mathcal{SHIN}$ are used in our semantic annotation tool prototype to explain modeling errors.

Key Words: ontologies, knowledge management, error explanations
Category: I.2.4, I.2.1, H.3.1, H.3.4, H.5.2

1 Introduction

The problem of error explanations turned out to be of high importance in ontology and semantic annotation authoring tools. Users of such tools need to be informed both about inconsistencies in the modeled ontology and about reasons for the inconsistency to occur. In this work we present two novel incremental algorithms for error explanations. These algorithms are universal enough to be reused with wide variety of reasoners, which seems to be of high importance especially in the dynamic field of semantic web languages.

We have tested the proposed algorithms with our implementation of $\mathcal{SHIN}$ description logics [Baader and Sattler 2001] (which is a significant subset of the OWL-DL language for semantic web) tableau algorithm. Both the explanation algorithms and the reasoner are used in the core of our novel annotation tool prototype to explain unsatisfiabity of concepts (ontology classes).

2 Error Explanation Techniques - State of the Art

The mainstream of error explanations for description logics knowledge bases tries to pinpoint axioms in the knowledge base to localize the error. The notion of *minimal unsatisfiability preserving subterminology* (MUPS) has been introduced in [Schlobach and Huang 2005], to describe minimal sets $\{S_i\}$ of axioms that cause given concept to be unsatisfiable. Removing a single axiom from each of these sets turns the concept satisfiable. Similarly to defining MUPSes for concept satisfiability, in [Kalyanpur 2006] the notion of *justification* for arbitrary axiom entailments has been presented. These justifications allow for explaining knowledge base inconsistencies, in our case annotation errors. Notions of MUPSes and justifications are dual, as for each concept an axiom can be found, for which the set of justifications corresponds to the set of MUPS of the concept. From now on, we will use w.l.o.g. only the notion of MUPS and concept satisfiability.

At present, there are two approaches for computing explanations of concept unsatisfiability: black-box (reasoner-independent) techniques and glass-box (reasoner-dependent) techniques. The former ones can be used directly with an existing reasoner, performing many satisfiability tests to obtain a set of MUPSes. The latter ones require smaller number of satisfiability tests, but they heavily influence the reasoner internals, thus being hardly reusable with other reasoning algorithms.

2.1 Black-box Techniques

There are plenty of black-box techniques that can be used for the purposes of error explanations. In [Banda et al. 2003], several simple methods based on *conflict set tree* (CS-tree) are shown. CS-trees allow for non-redundant searching in the power set of a given axiom set. Each node in a CS-tree is labeled with two sets, a set D of axioms that necessary belong to a MUPS and a set P of axioms that might belong to a MUPS. Each node represents the set $D \cup P$ and it has $|D \cup P|$ children, each one lacking an axiom from $D \cup P$. The method (denoted as *allMUPSbb*) introduced in [Banda et al. 2003] effectively searches the CS-tree in the depth-first manner, pruning necessarily satisfiable nodes.

An interesting black-box approach [Kalyanpur 2006] is based on a method for computing a single MUPS (denoted as *singleMUPSbb*) of a concept for given axiom set. In the first phase, this algorithm starts with an empty set K and fills it with all available axioms one by one until it becomes unsatisfiable. In the second phase, each axiom is conditionally removed from K. If the new K turns satisfiable, the axiom is put back. To obtain an algorithm for all MUPSes the general purpose Reiter's algorithm [Reiter 1987] for computing hitting sets of given conflict set is used. This algorithm generates a tree, each node of which is labeled with the knowledge base and a MUPS computed for this knowledge base using *singleMUPSbb*. Starting with an arbitrary root MUPS, each of its children is generated by removing one of the MUPS axioms from the knowledge base and computing a single MUPS for the new knowledge base. The search terminates when all leaves of the tree are satisfiable. The advantage of this approach is that it provides also repair solutions that are represented by axioms of minimal (w.r.t. set inclusion) paths starting in root. These paths correspond to hitting sets of the set of MUPSes. Due to the lack of space we refer to the works [Banda et al. 2003], [Kalyanpur 2006], [Schlobach and Huang 2005] for detailed algorithm descriptions.

2.1.1 Glass-box Techniques

A fully glass-box technique for axiom pinpointing in the description logic $\mathcal{ALC}$ [Baader and Sattler 2001] is introduced in [Schlobach and Huang 2005]. However, for more expressive languages, like OWL-DL, OWL-Lite or OWL 1.1, there is no full adaptation of this approach. A partially glass-box method for searching a single MUPS in OWL-DL is presented in [Kalyanpur 2006]. This method provides only supersets of MUPSes which justifies its usage as a preprocessing step in the first phase of the *singleMUPSbb* algorithm described above.

3 Incremental Approach to Error Explanations

High number of very expensive tableau algorithm runs required for black-box methods, as well as lack of glass-box methods, together with their poor reusability, has given rise to the idea of using incremental techniques for axiom pinpointing. These methods require the reasoner being able both to provide its current state, and to apply given axiom to given state. There is, however, no other interaction with the reasoner. These features place incremental methods somewhere between black-box and glass-box approach.

The following sections introduce two novel incremental methods for axiom pinpointing. Both of these methods need a reasoner state to be stored. Although the incremental approach can be used with a wide range reasoning algorithms, we have tested its feasibility with a tableau algorithm for $\mathcal{SHIN}$. The tableau reasoner state consists of two parts: a completion graph repository storing partially expanded completion graphs, and an axiom set used for expanding the completion graphs so far.

3.1 Computing a single MUPS

In this section, a novel incremental algorithm for computing a single MUPS is presented. This algorithm (see Algorithm 1) starts with an axiom list P and an empty state e of the reasoner. Axioms from P are tested one by one with the current reasoner state until the incremental test fails. The axiom $P(i)$ causing the unsatisfiability is put into the single MUPS core D, the rest of the axiom list is pruned and the direction of the search in the axiom list changes. The algorithm terminates, when all axioms are pruned.

Correctness.

Correctness of the algorithm is ensured by the following invariant. Before each direction change, D contains axioms that, together with some axioms in the non-pruned part of the axiom list, form a MUPS. Whenever an axiom i causes unsatisfiability, there must exist a MUPS that consists of all axioms in D, axiom

Algorithm 1 An Incremental Single MUPS Algorithm

```
1:  function SINGLEMUPSINC($P, e$)            ▷ $P$ ... initial axioms, $e$ ... initial state.
2:      $lower, i \leftarrow 0$
3:      $upper \leftarrow length(P) - 1$
4:      $D \leftarrow \emptyset$
5:      $sD, last \leftarrow e$
6:      $direction \leftarrow +1$
7:      while $lower \leq upper$ do
8:          if $i \geq length(P)$ then
9:              return $\emptyset$
10:         end if
11:         $(incState, result) \leftarrow test(P(i), last)$
12:         if $result$ then
13:             $last \leftarrow incState$
14:             $i = i + direction$
15:         else
16:             $D \leftarrow D \cup \{P(i)\}$
17:             $(sD, result) \leftarrow test(P(i), sD)$
18:             $last \leftarrow sD$
19:             if $direction = 1$ then
20:                 $upper \leftarrow i - 1$
21:             else
22:                 $lower \leftarrow i + 1$
23:             end if
24:             $direction \leftarrow -direction$            ▷ $+1$... right, $-1$... left
25:         end if
26:     end while
27:     return $D$
28: end function
```

i and some axioms in the previously searched part of the axiom list. This MUPS cannot be affected by pruning the axiom list tail that has not been explored in this iteration.

Example 1. Let's have an ontology containing six axioms numbered $1 \ldots 6$, where the unsatisfiability of a concept is caused by these axiom sets (MUPSes) $\{\{1,2,4\},\{2,4,5\},\{3,5\},6\}$. The *singleMUPSInc* algorithm works as follows :

direction	input list	mups core
	[1, 2, 3, 4, 5, 6]	$D = []$
$\longrightarrow$	[1, 2, 3, ~~**4**, 5, 6~~]	$D = [4]$
$\longleftarrow$	[~~**1**~~, 2, 3, ~~4, 5, 6~~]	$D = [4,1]$
$\longrightarrow$	[~~1~~, ~~**2**, 3, 4, 5, 6~~]	$D = [4,1,2]$

Each line corresponds to a direction change. Whenever an unsatisfiability is detected, the search direction is changed, "overlapping" axioms are pruned (emphasized by strikeout) and the last axiom that caused the unsatisfiability (in bold) is put into the MUPS core D.

3.2 Computing all MUPSes

An incremental algorithm that can be used to search for all MUPSes (let's denote this algorithm as *allMUPSInc1*) is presented in [Banda et al. 2003]. This algorithm assumes, that a state of the underlying reasoner depends on the order of axiom processing. However, tableaux algorithms [Baader and Sattler 2001] are adopted in almost all current semantic web reasoners (for example Pellet). In case of tableau algorithms, two different permutations of an axiom set shall result in two equivalent states. Exploiting this fact, we modified the original algorithm to decrease the number of redundant calls to the testing procedure, resulting in Algorithm 2 (`allMUPSInc2`).

The original algorithm `allMUPSInc1` manages three axiom lists D, T and P. At the beginning of each recursive call, D contains axioms that must belong to all MUPSes searched in this recursive call, P represents possible axioms that might belong to some of these MUPSes and T represents a list of already tested axioms. The first while cycle adds axioms from P to T while T remains satisfiable. If an axiom that causes unsatisfiability is detected, the execution is branched. The first recursive call tries to remove this axiom and go on adding axioms from P to T, while the second branch tries to add the axiom to the MUPS core D found so far. If D turns unsatisfiable, a MUPS has been found. If A does not contain a subset of this MUPS, it is inserted into A, and A is returned.

Our modification of the original algorithm avoids executing some redundant tests – both in the while cycle and in testing whether D turns unsatisfiable. For this purpose, we store the position of the first unsatisfiability test in P in the parameter *cached*. Let's denote $T = \{t_1, \ldots, t_a\}$ and $P = \{p_1, \ldots, p_b\}$. Then *cached* is such an index to P, that $\{t_1, \ldots, t_a, p_1, \ldots, p_{cached}\}$ is unsatisfiable and each of its subsets is satisfiable.

Algorithm 2 Modified version of *allMUPSInc1*

```
function allMUPSInc2($D, sD, P, T, sT, A,$ cached)        ▷ $sD$ ($sT$) is the state for $D$ ($T$).
    $result \leftarrow true$
    $i \leftarrow -1$
    while $result \wedge \exists c \in P$ do
        $i \leftarrow i + 1$
        if $c \notin T$ then
            $T \leftarrow T \cup \{c\}$
            $lT \leftarrow sT$
            if $i = cached$ then
                $result \leftarrow false$
                break
            else
                $(result, sT) \leftarrow test(c, sT)$
            end if
        end if
    end while
    if $result$ then
        return $A$
    end if
    $A \leftarrow allMUPSInc2(D, sD, P \setminus \{c\}, T \setminus \{c\}, lT, A, -1)$
    $D \leftarrow D \cup \{c\}$
    if $i = 0 \wedge d = t$ then
        $result \leftarrow false$
    else
        $(result, sD) \leftarrow test(c, sD)$
    end if
    if $\neg result$ then
        if $\neg \exists a \in A$ such that $a \subset D$ then
            $A \leftarrow A \cup \{D\}$
        end if
        return $A$
    end if
    return $allMUPSInc2(D, sD, P \setminus \{c\}, D, sD, A, i - 1)$
end function
```

Correctness.

Correctness of the algorithm is ensured by the same invariant as for *allMUPSInc1* presented in [Banda et al. 2003] and the fact that, the variable *cached* uses the information of the last successful test performed on T only in the second recursive call, where $D = T$. In the first recursive call, the information cannot be used, as the sets D and T differ.

Example 2. To show how *allMUPSInc2* works, assume an axiom set $\{1, 2, 3\}$. MUPSes for unsatisfiability of a concept are $\{\{1, 2\}, \{1, 3\}\}$. The algorithm runs as follows :

$$\begin{tikzcd}
-1, [], [1, \underline{2}, 3], [] \arrow[rrr, "{[2]}"] \arrow[d, "{[1],\ [1,\ 2]}"] & & & 0, [2], [\underline{1}, 3], [2] \arrow[r, "\cancel{[1,\ 2]}"] \arrow[d, "\cancel{[1,\ 2]}"] & \text{found [2,1]} \\
-1, [], [1, \underline{3}], [1] \arrow[r, "{[3]}"] \arrow[d, "{[1,\ 3]}"] & 0, [3], [\underline{1}], [3] \arrow[r, "\cancel{[1,\ 3]}"] \arrow[d, "\cancel{[1,\ 3]}"] & \text{found [3,1]} & -1, [2], [3], [2] \arrow[d, -|, "{[2,\ 3]}"] & \\
-1, [], [1], [1] & -1, [3], [], [3] & & {} &
\end{tikzcd}$$

Each node in this graph represents a call to the procedure *allMUPSInc2*, with the signature $cached, D, P, T$. The search starts in the node $-1, [], [1, 2, 3], []$ and is performed in the depth first manner preferring up-down direction (first recursive call) to the horizontal one (second recursive call). Axioms that cause unsatisfiability in the given recursive call are underlined. Edges are labeled with the tests that have been done before the unsatisfiability is found and struck axiom sets represent the tests that are not performed, contrary to *allMUPSInc1*. In this example *allMUPSInc2* requires 6 tests contrary to 10 tests executed by *allMUPSInc1*.

4 Experiments

First, the discussed methods have been compared with respect to the overall performance. Two ontologies have been used for tests: the miniTambis ontology (30 unsatisfiable concepts out of 182) and the miniEconomy ontology[1](51 unsatisfiable out of 338). As shown in Tab.4, the performance of incremental methods is significantly better than the fully black box approach. Furthermore, combination of Reiter's algorithm and *singleMUPSInc1* is typically 1-2 times worse than the fully incremental approaches. However, the main advantage of the *singleMUPSInc1* in comparison to the fully incremental approaches is that it allows direct computation of hitting sets of the set of MUPSes (i.e. generating repair diagnoses), which makes them more practical.

It can be seen that our modification *allMUPSesInc2* of *allMUPSesInc1* provides a slight increase in the performance. To evaluate the difference in more detail, see Fig.1. This figure shows the significance of caching for different MUPS configurations. The highest performance gain (over 30%) is obtained for ontologies containing a lot of MUPSes with approx. half size of the ontology size.

[1] To be found at http://www.mindswap.org/2005/debugging/ontologies

	miniTambis (time [ms])	miniEconomy (time [ms])
allMUPSbb	$> 15min.$	$> 15min.$
Reiter + singleMUPSbb	67481	$> 15min.$
Reiter + singleMUPSinc	19875	19796
allMUPSInc1	8655	14110
allMUPSInc2	7879	12970

Table 1: Comparison of incremental and black-box algorithms.

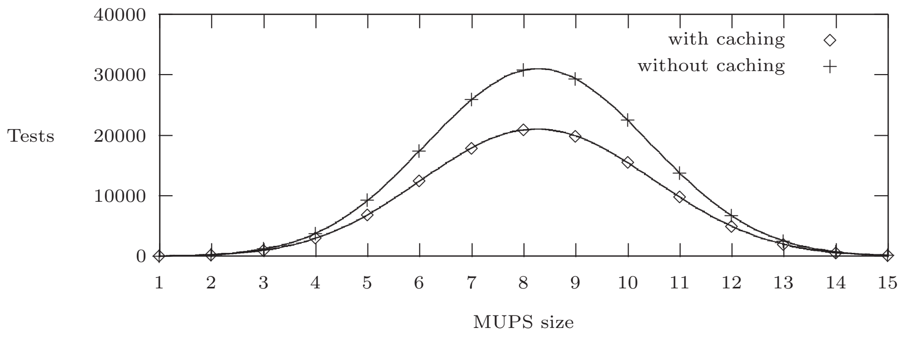


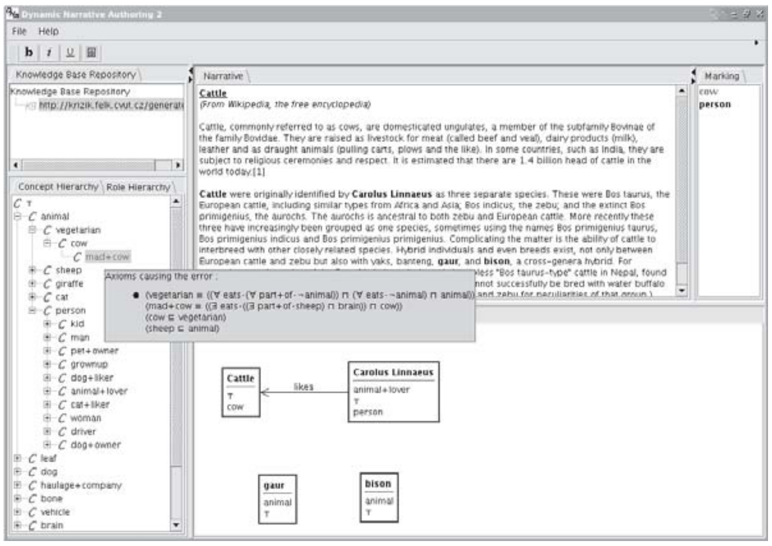
Figure 1: Comparison of incremental algorithm with caching and without it. Different configuration of MUPSes of 15 axioms, say $\{1, \ldots, 15\}$, were tested. For each $1 \leq k \leq 15$ (the x-axis), the set of all MUPSes is generated, so that it contains all axiom combinations of size k, thus containing $\frac{15!}{k!(15-k)!}$ MUPSes. For example, for $k = 2$, the set of MUPSes is $\{\{1,2\},\{1,3\},\ldots,\{1,15\},\{2,3\},\ldots\}$.

5 Conclusions and Future Work

Two novel incremental algorithms for finding minimal sets of axioms responsible for given modeling error in an ontology have been introduced. The first one is an incremental algorithm that searches for one such minimal axiom set (MUPS). The second one is an extension of the fully incremental algorithm presented in [Banda et al. 2003] used for searching all minimal axiom sets.

The introduced incremental methods seem promising and more efficient than the fully black box approaches in the context of error explanations. Although the fully incremental approaches are more efficient than the combination of single MUPS testers and Reiter's algorithm, they do not allow to compute diagnoses directly. This justifies our focus on both approaches. A problem of generating diagnoses by the fully incremental methods efficiently is still an open issue.

The explanation service for concept satisfiability based on the discussed incremental methods is implemented in our new annotation tool prototype (see Fig.2), providing the user with concept unsatisfiability explanations [Křemen 2007].



Cattle
(From Wikipedia, the free encyclopedia)

Cattle, commonly referred to as cows, are domesticated ungulates, a member of the subfamily Bovinae of the family Bovidae. They are raised as livestock for meat (called beef and veal), dairy products (milk), leather and as draught animals (pulling carts, plows and the like). In some countries, such as India, they are subject to religious ceremonies and respect. It is estimated that there are 1.4 billion head of cattle in the world today.[1]

Cattle were originally identified by **Carolus Linnaeus** as three separate species. These were Bos taurus, the European cattle, including similar types from Africa and Asia; Bos indicus, the zebu; and the extinct Bos primigenius, the aurochs. The aurochs is ancestral to both zebu and European cattle. More recently these three have increasingly been grouped as one species, sometimes using the names Bos primigenius taurus, Bos primigenius indicus and Bos primigenius primigenius. Complicating the matter is the ability of cattle to interbreed with other closely related species. Hybrid individuals and even breeds exist, not only between European cattle and zebu but also with yaks, banteng, **gaur**, and **bison**, a cross-genera hybrid. For
less "Bos taurus-type" cattle in Nepal, found
nnot successfully be bred with water buffalo
and zebu for peculiarities of that group.)

Axioms causing the error :

- (vegetarian ≡ ((∀ eats-(∀ part+of-¬animal)) ⊓ (∀ eats-¬animal) ⊓ animal))
  (mad+cow ≡ ((∃ eats-((∃ part+of-sheep) ⊓ brain)) ⊓ cow))
  (cow ⊑ vegetarian)
  (sheep ⊑ animal)

Figure 2: Semantic Annotation Tool Prototype. To detect and explain modeling errors our implementation of the $\mathcal{SHIN}$ reasoner and presented incremental algorithms are used. The annotation graphs (bottom pane) represent annotations created from the current document (top pane) and the ontology (left pane).

Acknowledgements

This work has been supported by the grant No. MSM 6840770038 “Decision Making and Control for Manufacturing III”' of the Ministry of Education, Youth and Sports of the Czech Republic.

References

[Baader and Sattler 2001] Baader, F., Sattler U.: “An overview of tableau algorithms for description logics”; Studia Logica, 69:5-40, 2001.

[Baader et al. 2001] Baader, F., Calvanese D., McGuiness D., Nardi D., Patel-Schneider P. (eds.): “The Description Logic Handbook, Theory, Implementation and Applications”; Cambridge, 2003.

[Banda et al. 2003] De la Banda M. G., Stuckey P. J., and Wazny J.: “Finding All Minimal Unsatisfiable Subsets”; In PPDP’03C, 2003.

[Kalyanpur 2006] Kalyanpur A.. ”Debugging and Repair of OWL Ontologies”; PhD thesis, University of Maryland, 2006.

[Křemen 2007] Křemen P. ”Inference Support for Creating Semantic Annotations”; Technical Report GL 190/07, CTU FEE in Prague, Dept. of Cybernetics, 2007.

[Schlobach and Huang 2005] Schlobach S. and Huang Z. ”Inconsistent Ontology Diagnosis: Framework and Prototype”; Technical Report, Vrije Universiteit Amsterdam.

[Reiter 1987] Reiter R.: “A Theory of Diagnosis from First Principles”; Artificial Intelligence, 32(1):57-96, April 1987.