Incremental Approach to Error Explanations in Ontologies

Petr Křemen and Zdeněk Kouba

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 their advantages w.r.t. 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 *SHIN* are used in our semantic annotation tool prototype to explain modeling errors.

1 Introduction

The problem of error explanations turned out to be of high importance in ontology editors and semantic annotation authoring tools. Users of such tools need to be informed both about inconsistencies in the modeled ontology and about reasons for these inconsistencies to occur. Rationale for this work was formulated during the implementation and evaluation of our narrative annotation tool [10] developed within the CiPHER project [4].

This work presents two novel incremental algorithms for error explanations in ontologies. These techniques can be regarded as a compromise between glass-box error explanation methods, that are fully integrated into the reasoning algorithms and thus strongly dependent on the expressivity of the chosen semantic web language and hardly reusable for other ones, and black-box techniques, that are fully reasoner-independent, but therefore quite inefficient, especially in combination with – already EXPTIME or worse – reasoning algorithms for expressive description logics. The proposed incremental algorithms are universal enough to be reused with wide variety of reasoners, which seems to be advantageous especially in the

Petr Křemen and Zdeněk Kouba
Czech Technical University in Prague, Faculty of Electrical Engineering,
Dept. of Cybernetics, Technická 2, 16627 Prague, Czech Republic
e-mail: {kremen,kouba}@labe.felk.cvut.cz

S. Schaffert et al. (Eds.): Networked Knowledge - Networked Media, SCI 221, pp. 171–185.
springerlink.com

dynamic field of semantic web languages, yet having quite tight interaction with the reasoner using the reasoner state. Proposed techniques were tested with our implementation of *SHIN* description logic tableau algorithm [1] with an incremental interface.

Section 2 surveys current black box and glass box techniques for error explanations in ontologies and shows advantages and disadvantages of these methods. Our incremental approach for error explanations is introduced in section 3 and evaluated in section 4. Section 5 briefly overviews our annotation tool prototype and this chapter is concluded by section 6.

2 Error Explanation Techniques – State of the Art

The mainstream of error explanations for description logic knowledge bases tries to pinpoint axioms in the knowledge base to localize errors. The notion of *minimal unsatisfiability preserving subterminology* (MUPS) has been introduced in [11], to describe minimal sets $\{S_i\}$ of axioms that cause a 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 [8] 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 use w.l.o.g. only the notion of MUPS and concept satisfiability.

At present, there are two general 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 a smaller number of satisfiability tests, but they heavily influence the reasoner internals, thus being hardly reusable with other reasoning algorithms.

In addition to these basic methods, [8] presents several practically interesting extensions. One of the techniques splits axioms into simpler ones (like $A \sqsubseteq B \sqcap C$ into $A \sqsubseteq B$ and $A \sqsubseteq C$) trading the error explanation granularity for the size of the knowledge base. Another technique tries to find concepts (called *root* concepts), unsatisfiability of which causes unsatisfiability of other concepts. Getting rid of unsatisfiability of these *root* concepts makes the other concepts satisfiable as well. The former technique can be used as a preprocessing and the latter as a postprocessing to all the methods described below.

2.1 Black-Box Techniques

There are plenty of black-box techniques that can be used for the purposes of error explanations. All of them have worst-case exponential time complexity in the number of axioms, as they search the power set of the axiom set – they differ in the search

strategies and pruning efficiency. For each candidate set of axioms a satisfiability check is necessary to determine, whether this axiom set causes the unsatisfiability of a given concept or not.

In [5], several simple methods based on *conflict set tree* (CS-tree) notion are shown (see Fig.1). CS-trees allow for efficient and 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 necessarily 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 [5] effectively searches the CS-tree in the depth-first manner, pruning necessarily satisfiable nodes. The CS-tree structure allows for various pruning methods, like constraint set partitioning and eliminating always satisfiable constraints, see [5] for more details. However, detailed evaluation of the feasibility of these methods and their optimizations for the axiom pinpointing problem is still an open issue.

Example 1 (Basic CS-tree algorithm). Consider a knowledge base consisting of three axioms

$$1 : C \sqsubseteq B \sqcap \exists R.A,$$
$$2 : B \sqsubseteq \forall R.\neg A,$$
$$3 : C \sqsubseteq D.$$

The concept C is unsatisfiable due to the single MUPS $\{1,2\}$. The run of the basic CS-tree algorithm presented in [5] is shown in Fig.1. The algorithm starts in the root $[],[1,2,3]$ and tries to find all MUPSes in the depth-first manner. All children for $[],[1,2,3]$ are generated and the left-most node $[],[2,3]$ is used for exploration. As this node is satisfiable, all of its children are pruned, the algorithm backtracks to the node $[1],[3]$, which is also satisfiable. After pruning its child and backtracking to the $[1,2],[]$ the searched MUPS is obtained. In this configuration the algorithm needs 4 satisfiability tests, while for the reversed axiom list 6 tests are needed.

An interesting black-box approach [8], [11] is based on a method for computing a single MUPS (denoted as *singleMUPSbb*) of a concept for a 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. An important observation is that *singleMUPSbb* algorithm is polynomial in the number of axioms. In the worst case we need $2n$ full consistency checks, where n is the number of axioms.

To obtain an algorithm for all MUPSes the general purpose Reiter's algorithm [13] for computing hitting sets of a given conflict set is used. This algorithm generates a tree (see Fig.2), where each node 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

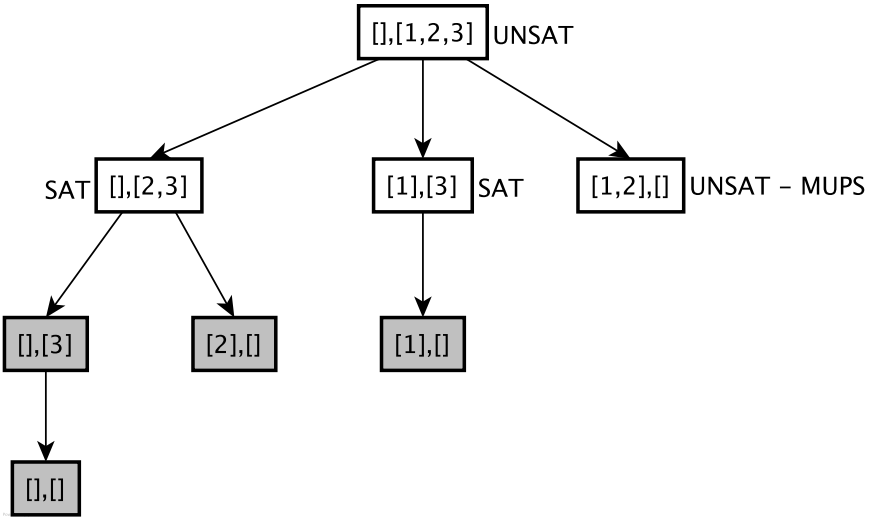


Fig. 1 An example of the basic CS-tree algorithm run searching MUPSes in a set of three axioms. Pruned nodes are darker.

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 [5], [8], [11] and [12] for detailed algorithm descriptions (see Fig.2).

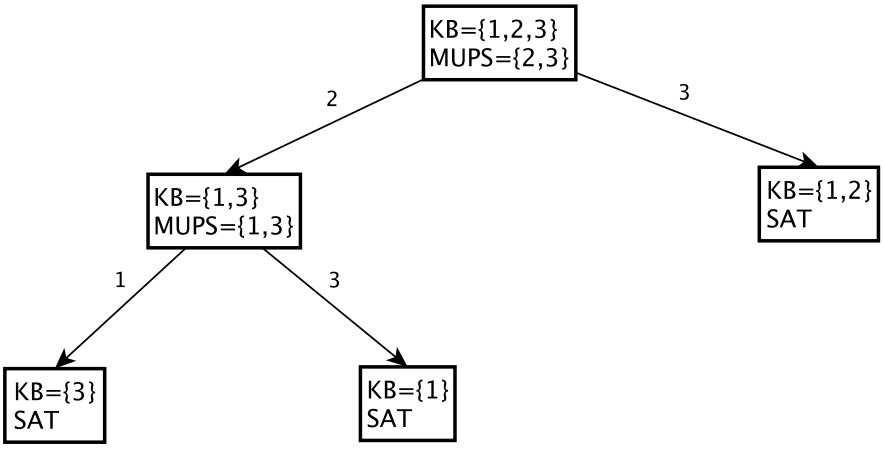


Fig. 2 A hitting set tree example

Example 2 (Single black box MUPS algorithm + Reiter's method). Let's have a knowledge base consisting of three axioms (A, B, C are concepts and R is a role.)

$$1 : B \sqsubseteq \forall R^{-}.\neg A,$$
$$2 : A \sqsubseteq \forall R.\neg B,$$
$$3 : C \sqsubseteq A \sqcap \exists R.B.$$

The concept C is unsatisfiable due to the MUPS set $\{\{1,3\},\{2,3\}\}$. The algorithm first uses *singleMUPSbb*, with $\{1,2,3\}$ as its input to find a MUPS, corresponding to the root of the hitting set tree in Fig.2. The *singleMUPSbb* has to perform 3 tests in the first phase ($\{1\} = SAT$, $\{1,2\} = SAT$, $\{1,2,3\} = UNSAT$) to get an unsatisfiable set $\{1,2,3\}$ and 3 tests in the second phase ($\{2,3\} = UNSAT$, $\{2\}$, $\{3\}$).

Now, all children of the root are generated and the left-most child is being explored, calling the *singleMUPSbb* to obtain a MUPS in $\{1,2,3\} \setminus \{2\} = \{1,3\}$, which is $\{1,3\}$. As both sub-knowledge bases $\{1\},\{3\}$ of $\{1,3\}$ are satisfiable, the algorithm backtracks and tests the satisfiability of $\{1,2,3\} \setminus \{3\} = \{1,2\}$, which is satisfiable. Therefore two MUPSes $\{1,3\}$ and $\{2,3\}$ were found together with the hitting sets $\{3\}$ and $\{1,2\}$. If any of these sets is removed from the knowledge base, the concept C turns satisfiable.

As stated above all black box methods require, in general, time exponential to the number of axioms. Combining this with already (at least) exponential satisfiability checking for most description logic languages, we reach scalability problems for most real world ontologies.

2.2 Glass-Box Techniques

A fully glass-box technique for axiom pinpointing in the description logic *ALC* [1] is introduced in [11]. This method labels all concepts and roles in nodes of a completion tree with axioms they depend on. These labels are modified according to the applied expansion rules. Whenever no rule is applicable on any tableau, the union of labels of clashing concepts/roles builds up a superset of some MUPS. To obtain a MUPS, this set is minimized by backtracking the rule changes applied during expansions and constructing a boolean formula ϕ (so called *minimization function*) using another set of rules (see [11]). The searched MUPS is equivalent to the minimal set of axioms, conjunction of which implies ϕ. For a more formal and detailed description, see [11].

Example 3 (A glass-box technique for ALC). Let's have a knowledge base containing two axioms :

$$1 : A \sqsubseteq B \sqcup \exists R.A,$$
$$2 : A \sqsubseteq \neg A.$$

The only MUPS for the satisfiability of A is clearly $\{2\}$. The completion graphs evolve as depicted in fig.3. Inference rule applications are represented by double arrows, labeled with the type of the used rule. Right of each concept a set of axioms is shown, that is responsible for the concept appearing in the node label. The initial

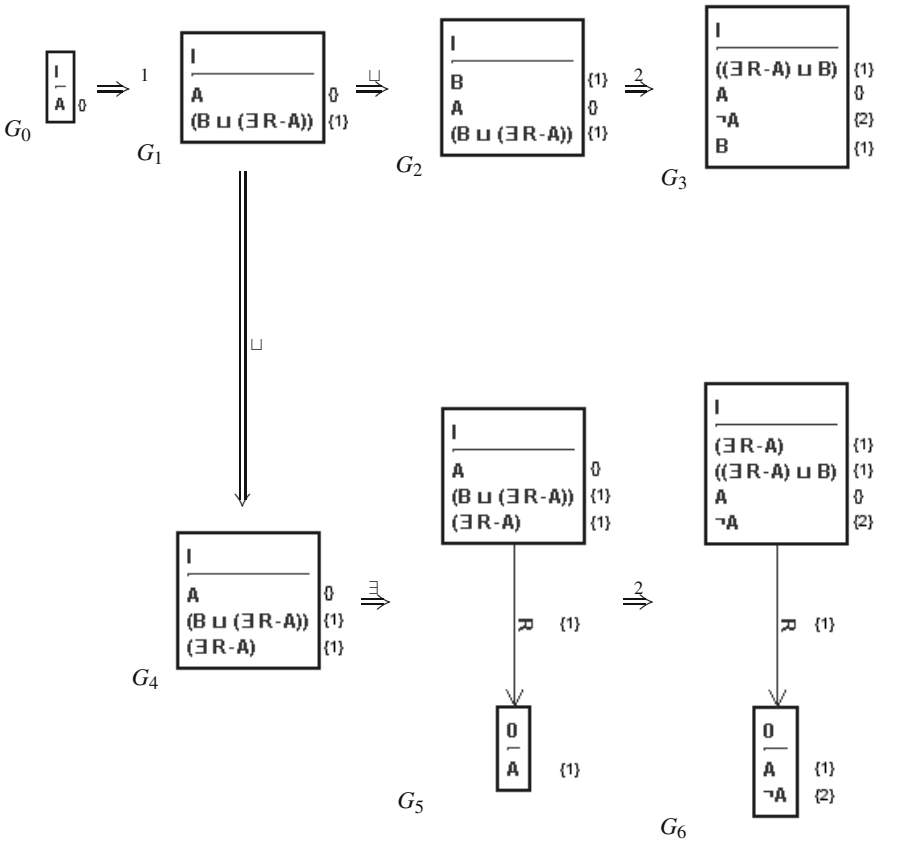


Fig. 3 Completion graph evolution for Example 3

completion graph G_0 contains a new individual I, asserted to belong to A. Rule applications result in two completion graphs G_3 and G_6, both containing a clash. The clash in G_3 is caused only by axiom 2, while the clash in G_6 is caused either by single axiom 2, or by axiom set $\{1,2\}$. Using the minimization rules introduced in [11], the following minimization function is obtained

$$2 \wedge (2 \vee (1 \wedge 2)). \tag{1}$$

The minimal set of axioms that makes this formula valid is $\{2\}$, which corresponds to the searched MUPS.

To the best of our knowledge there is no adaptation of this approach to more expressive languages, like *SHIN*, or OWL-DL. The problem lies in possibly complex interactions between completion rules for different language constructs that have to be traced back in the minimization function. However, [8] presents a partially glass-box method for searching a single MUPS in OWL-DL. The algorithm is an extension of

the first phase of the fully glass-box method described above. During completion graph expansion, concepts and roles are labeled with sets of axioms they depend on. Finding a clash in all branches, union of labels of clashing concepts/roles is a superset of a MUPS, usually much smaller than the initial axiom set. This method is then used as a preprocessing step in the first phase of the *singleMUPSbb* algorithm.

3 Incremental Approach to Error Explanations

A high number of very expensive tableau algorithm runs required for black-box methods, as well as a 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 to be able both to provide its current state, and to apply a given axiom to a given state. There is, however, no other interaction with the reasoner. These features place incremental methods somewhere between black-box and glass-box approach.

3.1 Incremental Tableau Reasoner

Incremental tableau reasoning has already been studied in [6], where additions and deletions of ABox (concept and role) assertions are considered. While additions can be implemented in a straightforward way due to the nature of tableau algorithms, to handle deletions all completion rules applications had to be tracked. When deleting an ABox axiom a rollback of the parts of completion graph dependent on this axiom had to be performed and completion rules reapplied.

For the purpose of incremental algorithms presented in the next section we just require the reasoner to support incremental additions of all axiom types (TBox, RBox, ABox). Thus, due to the monotonicity of considered description logics (like *SHIN*) we do not need making any changes to the implementation of the tableau reasoning strategy. We only need the reasoner to provide us with its current *state*. The tableau reasoner state consists of two parts: a set of completion graphs, and an axiom set used for expanding this completion graph so far. More formally, we represent the incremental reasoner as

$$(ns, r) \leftarrow test(a, s) \tag{2}$$

where s is the state before and ns the state after performing the incremental test, a is an axiom and r is a boolean result of the satisfiability test. Although we have tested its feasibility with a tableau algorithm for *SHIN* [7], our incremental approach can be used with a wide range of reasoning algorithms.

The following sections introduce two novel incremental methods for finding all MUPSes of a given concept. They use the above incremental reasoner interface as a black box. This makes them applicable to reasoning services (like DL tableau-algorithms) of monotonic logics without interaction with internals of these services.

3.2 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 changes, 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 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.

Algorithm 1. An Incremental Single MUPS Algorithm

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

Example 4. Let's have an ontology containing six axioms $1 \ldots 6$, where the unsatisfiability of some concept is caused by 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.

Let's have n axioms. All incremental consistency checks in a single run between two direction changes correspond approximately to one full consistency check performed for all axioms in the run. Thus, the incremental method requires in the worst case n full consistency tests ($n(n+1)/2$ incremental consistency tests), comparing to worst-case $2n$ full consistency tests for the *singleMUPSb* algorithm. In the example above, 9 incremental consistency tests (effectively 3 full consistency tests) are needed comparing to 8 full consistency tests needed by the *singleMUPSbb*.

3.3 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 [5]. This algorithm assumes that a state of the underlying reasoner depends on the order of axiom processing. However, tableau algorithms [1] 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

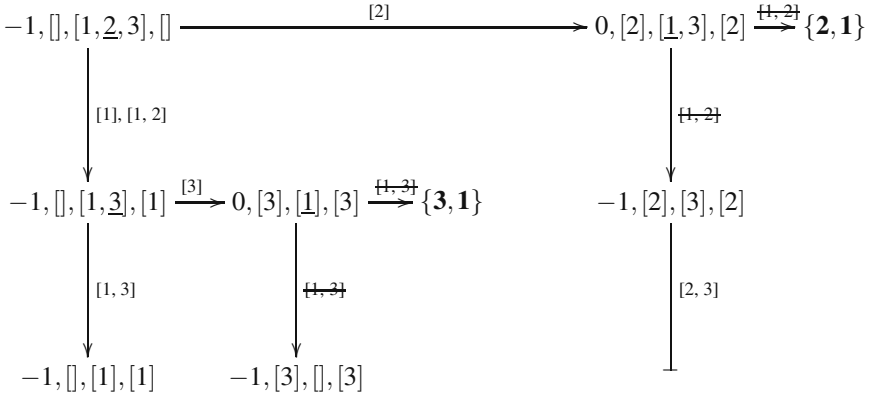
Algorithm 2. Modified version of *allMUPSInc1*

```
 1: function ALLMUPSINC2(D, sD, P, T, sT, A, cached)    ▷ sD (sT) is the state for D (T).
 2:     result ← true
 3:     i ← −1
 4:     while result ∧ ∃c ∈ P do
 5:         i ← i + 1
 6:         if c ∉ T then
 7:             T ← T ∪ {c}
 8:             lT ← sT
 9:             if i = cached then
10:                 result ← false
11:                 break
12:             else
13:                 (result, sT) ← test(c, sT)
14:             end if
15:         end if
16:     end while
17:     if result then
18:         return A
19:     end if
20:     A ← allMUPSInc2(D, sD, P \ {c}, T \ {c}, lT, A, −1)
21:     D ← D ∪ {c}
22:     if i = 0 ∧ d = t then
23:         result ← false
24:     else
25:         (result, sD) ← test(c, sD)
26:     end if
27:     if ¬result then
28:         if ¬∃a ∈ A such that a ⊂ D then
29:             A ← A ∪ {D}
30:         end if
31:         return A
32:     end if
33:     return allMUPSInc2(D, sD, P \ {c}, D, sD, A, i − 1)
34: end function
```

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.

Correctness. Correctness of the algorithm is ensured by the same invariant as for *allMUPSInc1* presented in [5] 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 5. 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 :



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 the tests: the miniTambis ontology (30 unsatisfiable concepts out of 182) and the miniEconomy ontology[1](51 unsatisfiable out of 338). As shown in Tab.1, 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 just a slight increase in the performance. To evaluate the difference in more detail, see Fig.4. 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. This is caused

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

Table 1 Comparison of incremental and black-box algorithms

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

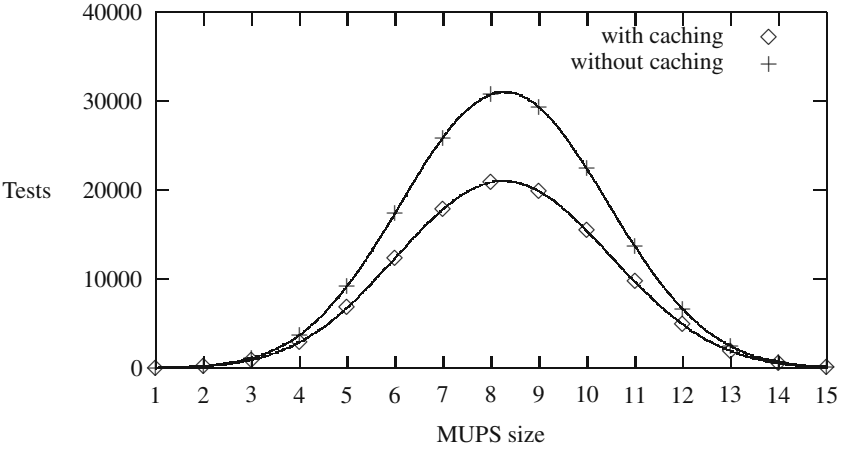


Fig. 4 Comparison of incremental algorithm with caching and without it. Different configuration of MUPSes of 15 axioms, say $\{1,\ldots,15\}$, were tested. For each $1 \leq k \leq 15$ (the x-axis), the set of all MUPSes is generated, so that it contains all axiom combinations of size k, thus containing $\frac{15!}{k!(15-k)!}$ MUPSes. For example, for $k = 2$, the set of MUPSes is $\{\{1,2\},\{1,3\},\ldots,\{1,15\},\{2,3\},\ldots\}$.

by the fact, that the broader the search tree of the *allMUPSesInc2* algorithm (see 5) is, the more applications of the caching occur.

Second, the performance and robustness of the incremental methods with respect to the axiom ordering were tested. As all permutations of a given axiom set are required, two small ontologies have been chosen. *TambisP* is a subset of Tambis ontology [2], restricted to the definitions of unsatisfiable concepts *metal*, *nonmetal* and *metalloid* (6 axioms). *MadCowP* is the restriction of Mad cow ontology [3] to the 7 axioms causing unsatisfiability of the concept *madCow*. The best results were obtained with *allMUPSInc2*, which is most efficient (measured by the count of IT) and robust enough to the axiom ordering (measured by test count variance). The results also show, like above, that the performance of *Reiter + singleMUPSinc* strongly depends on the axiom ordering.

[2] http://protege.stanford.edu/plugins/owl/owl-library/tambis-full

[3] http://www.mindswap.org/2005/debugging/ontologies/madcow.owl

Table 2 Comparison of discussed incremental methods. For each ontology and each unsatisfiable concept, the tests are performed for all permutations of the input axiom set. '#', 'avg' and 'var' stands for number of, average and variance of incremental tests.

tambisP	# of inc. tests	avg	var
R. + singleMUPSinc	268362	124.29	206.81
allMUPSInc1	75696	35.04	36.44
allMUPSInc2	61590	28.51	16.76
madCowP	**# of inc. tests**	**avg**	**var**
R. + singleMUPSinc	277200	55.00	8.00
allMUPSInc1	131040	26.00	0.00
allMUPSInc2	119520	23.04	0.50

5 Annotation Tool Prototype

Exploiting our experience with the annotation prototype based on conceptual graphs [10], we are developing an annotation tool, see Fig.5, that will integrate described reasoning services to support detecting modeling errors. The tool is aimed at authors of semantic annotations of narratives and other natural language documents.

The annotation tool prototype consists of several modules. The *ontology module* is the core of the system. Its internal model corresponds to the description

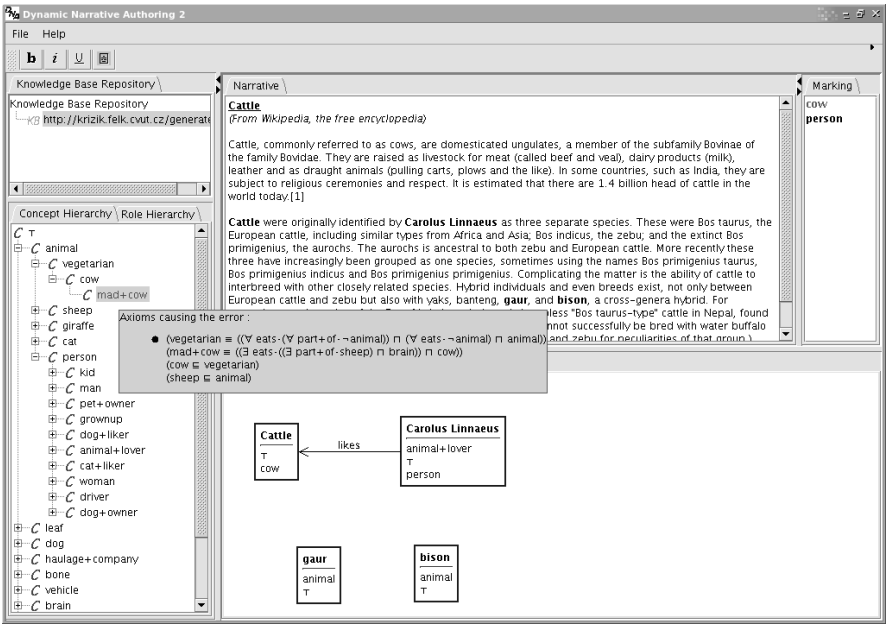


Fig. 5 Annotation Tool Prototype

logic *SHIN*. This model is connected to our implementation of a tableau reasoner for *SHIN*, allowing for concept satisfiability, subsumption, disjointness testing and knowledge base consistency checking. Furthermore, the reasoner could be run in server mode, using the DIG [3] interface for communication. The reasoner is equipped with the above introduced concept satisfiability explanation functionality.

The *annotation module* will serve to create annotations using annotation graphs. Basic form of these graphs allows for creating ABOX assertions. Further refinements are needed to provide inequality assertions, equality assertions, n-ary relations, and other.

The *document module* manages the documents to be annotated. It provides a simple text editor, that allows for visualizing the annotated parts of the document directly in the texts. Finally, the *marking module* allows for color highlighting of annotations according to the classes they belong to.

6 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 a novel incremental algorithm that searches for one such minimal axiom set (MUPS). The second one is an extension of the fully incremental algorithm presented in [5] used for searching all minimal axiom sets.

The introduced incremental methods seem promising and our experiment proved that they are also 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. Efficient generation of diagnoses by the fully incremental methods is an open issue.

While it does not seem feasible to invent a sound and complete fully glass-box method that might be reused in a wide range of description logics formalisms, it seems promising to use an incomplete glass-box approach (like the one discussed in sec. 2.2) as the preprocessing step for the incremental methods discussed above. Furthermore, we would like to test several optimizations of the introduced methods, like partitioning of the axiom set.

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

1. Baader, F., Sattler, U.: An overview of tableau algorithms for description logics. Studia Logica 69, 5–40 (2001)
2. Baader, F., Calvanese, D., McGuiness, D., Nardi, D., Patel-Schneider, P.(eds.): The Description Logic Handbook, Theory, Implementation and Applications. Cambridge (2003)

3. Bechhofer, S., Moller, R., Crowther, P.: The DIG Description Logic Interface. In: Proc. of International Workshop on Description Logics, DL 2003 (2003)
4. CIPHER project homepage, `http://cipherweb.open.ac.uk` (cited December 2007)
5. De la Banda, M.G., Stuckey, P.J., Wazny, J.: Finding All Minimal Unsatisfiable Subsets. In: PPDP 2003C (2003)
6. Halaschek-Wiener, C., Parsia, B., Sirin, E.: Description logic reasoning with syntactic updates. In: Proc. of the 5th International Conference on Ontologies, Databases, and Applications of Semantics, ODBASE 2006 (2006)
7. Horrocks, I., Sattler, U., Tobies, S.: Practical Reasoning for Expressive Description Logics. In: Ganzinger, H., McAllester, D., Voronkov, A. (eds.) LPAR 1999. LNCS, vol. 1705. Springer, Heidelberg (1999)
8. Kalyanpur, A.: Debugging and Repair of OWL Ontologies. PhD thesis, University of Maryland (2006)
9. Křemen, P.: Inference Support for Creating Semantic Annotations. Technical Report GL 190/07, CTU FEE in Prague, Dept. of Cybernetics (2007)
10. Uhlíř, J., Kouba, Z., Křemen, P.: Graphical Interface to Semantic Annotations. In: Znalosti 2005, pp. 129–132. Ostrava (2005)
11. Schlobach, S., Huang, Z.: Inconsistent Ontology Diagnosis: Framework and Prototype. Technical Report, Vrije Universiteit Amsterdam (2005)
12. Schlobach, S., Huang, Z., Cornet, R., Van Harmelen, F.: Debugging Incoherent Terminologies. Technical Report, Vrije Universiteit Amsterdam (2006)
13. Reiter, R.: A Theory of Diagnosis from First Principles. Artificial Intelligence 32(1), 57–96 (1987)