Finding All Justifications of OWL DL Entailments

Aditya Kalyanpur[1], Bijan Parsia[2], Matthew Horridge[2], and Evren Sirin[3]

[1] IBM Watson Research Center, 19 Skyline Drive, Hawthorne NY 10532, USA
adityakal@us.ibm.com
[2] School of Computer Science, University of Manchester, UK
{bparsia,matthew.horridge}@cs.man.ac.uk
[3] Clark & Parsia LLC, Washington DC, USA
evren@clarkparsia.com

Abstract. Finding the justifications of an entailment (that is, all the minimal set of axioms sufficient to produce an entailment) has emerged as a key inference service for the Web Ontology Language (OWL). Justifications are essential for debugging unsatisfiable classes and contradictions. The availability of justifications as explanations of entailments improves the understandability of large and complex ontologies. In this paper, we present several algorithms for computing all the justifications of an entailment in an OWL-DL Ontology and show, by an empirical evaluation, that even a reasoner independent approach works well on real ontologies.

Keywords: OWL Ontology Explanation, Debugging, Justifications.

1 Introduction

Since OWL became a W3C standard, there has been a notable increase in the number of people that are attempting to build, extend and use ontologies. To some extent, the provision of editing environments, visualization tools and reasoners has helped catalyse this. In traditional ontology editing environments, users are typically able to create ontologies and use reasoners to compute unsatisfiable classes, subsumption hierarchies and types for individuals. However, as ontologies have begun to be used in real world applications, and a broader audience of users and developers have been introduced to Ontology Engineering, it has become evident that there is a significant demand for editing environments which provide more sophisticated services.

In particular, the generation of explanations, or *justifications*, for inferences computed by a reasoner is now recognized as highly desirable functionality for both ontology development and ontology reuse. A clear demonstration of the need for *practical* explanation services manifested itself in the observations of the switching of users from Protege 3.2 [1] to Swoop [2] purely for the benefits of automatic explanation facilities [3].

As an example of explanation of entailments using Swoop, see Figure 1. The left part of the Figure shows a justification for the entailed subsumption

K. Aberer et al. (Eds.): ISWC/ASWC 2007, LNCS 4825, pp. 267–280, 2007.
© Springer-Verlag Berlin Heidelberg 2007

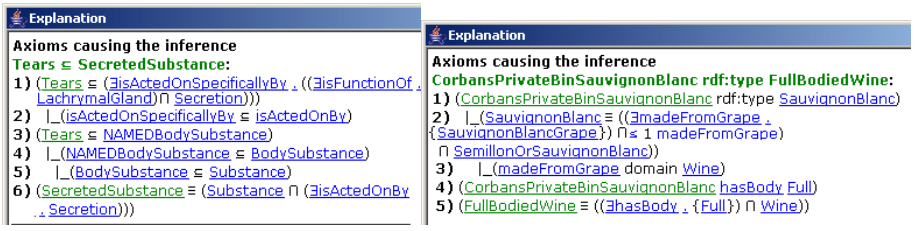

Fig. 1. Justifications in GALEN and Wine

Tears $\sqsubseteq$ *SecretedSubstance* in the medical ontology GALEN[1]. The right part of the Figure shows a justification for why a *CorbansPrivateBinSauvignonBlanc* is entailed to be an instance of the concept *FullBodiedWine* in the Wine[2] Ontology. Both GALEN and Wine are expressive OWL-DL ontologies with non-trivial entailments, and understanding how these entailments arise becomes much easier using Swoop's UI for displaying justifications (i.e., axioms responsible for the inference).

In general, the algorithms for finding justifications come in two flavors: *Black-box* and *Glass-box*:

- *Black-box (reasoner independent)* algorithms use the reasoner solely as a *subroutine* and the internals of the reasoner do not need to be modified. The reasoner behaves as a "Black-box" that typically accepts as input, an ontology and a specific entailment test, and returns an affirmative or a negative answer, depending on whether the entailment holds in the ontology. In order to obtain justifications, the algorithm selects the appropriate inputs to the reasoner and interprets its output accordingly. While Black-box algorithms typically require many satisfiability tests, they can be easily and robustly implemented – they only rely on the availability of a sound and complete reasoner for the logic in question, and thus can be implemented on reasoners based on techniques other than tableaux, such as resolution.
- *Glass-box (reasoner dependent)* algorithms are built on existing tableau-based decision procedures for expressive Description Logics. Their implementation requires a thorough and non-trivial modification of the internals of the reasoner. For a tableau based system, these involve some form of tracing through the tableau. Tracing techniques have been used to find all justifications for the description logic (DL) $\mathcal{ALC}$ [4] and also single justifications with rather low overhead for $\mathcal{SHIF}$ (in our previous work [5]).

In this paper, we present a *practical* two stage Black-box technique for effectively finding *all* justifications for an entailment, and demonstrate its significance on a set of expressive, realistic OWL Ontologies. In addition, we describe and evaluate a faster hybrid solution that combines a Glass-box and Black-box approach to computing all justifications.

[1] http://www.cs.man.ac.uk/~horrocks/OWL/Ontologies/galen.owl
[2] http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf

1.1 Justification of Entailments

Informally, a justification is simply the precise set of axioms in an ontology responsible for a particular entailment. For example, if an ontology $\mathcal{O}$ contains concepts A, B, and A is inferred to be a subclass of B, i.e., $\mathcal{O} \models A \sqsubseteq B$, then the justification for this concept subsumption entailment is simply the smallest set of axioms in $\mathcal{O}$ responsible for it. It is important to realize that there may be more that one justification for a given entailment in the ontology. If *at least one* of the axioms in *each* of the justifications for an entailment is removed from the ontology, then the corresponding entailment no longer holds.

Justifications are formally defined as follows:

Definition 1. *(JUSTIFICATION)*
Let $\mathcal{O} \models \alpha$ where α is an axiom and $\mathcal{O}$ is a consistent ontology. A fragment $\mathcal{O}' \subseteq \mathcal{O}$ is a justification for α in $\mathcal{O}$, denoted by $\mathsf{JUST}(\alpha, \mathcal{O})$, if $\mathcal{O}' \models \alpha$, and $\mathcal{O}'' \not\models \alpha$ for every $\mathcal{O}'' \subset \mathcal{O}'$.

We denote by $\mathsf{ALL\_JUST}(\alpha, \mathcal{O})$ the set of all the justifications for α in $\mathcal{O}$.

While our goal, in general, is to compute all justifications for any arbitrary entailment in an OWL-DL ontology, we specifically focus on a particular type of entailment – concept unsatisfiability. This is because an OWL-DL ontology corresponds to a $\mathcal{SHOIN}(\mathcal{D})$ knowledge base, and [6] shows that for every sentence (axiom or assertion) α entailed by a $\mathcal{SHOIN}(\mathcal{D})$ KB $\mathcal{O}$, there is always a class C_α that is unsatisfiable w.r.t $\mathcal{O}$. Conversely, given any class C that is unsatisfiable w.r.t. $\mathcal{O}$, there is always a sentence α_C that is entailed by $\mathcal{O}$.

As an example, $\mathcal{O} \models A \sqsubseteq B$ iff the class $A \sqcap \neg B$ is unsatisfiable in $\mathcal{O}$. Thus, the axioms responsible for the subsumption entailment $\mathcal{O} \models A \sqsubseteq B$ are precisely the same set of axioms responsible for the concept unsatisfiability entailment $\mathcal{O} \models A \sqcap \neg B \sqsubseteq \bot$. Consequently, given an OWL-DL Ontology ($\mathcal{SHOIN}(\mathcal{D})$ KB), the problem of finding all justifications for an arbitrary entailment reduces to the problem of finding all justifications for *some* unsatisfiable concept[3].

Therefore, in the remainder of this paper, we shall restrict our attention, without loss of generality, to the problem of finding all justifications for an unsatisfiable concept w.r.t to a consistent $\mathcal{SHOIN}(\mathcal{D})$ KB.[4]

2 Finding Justifications

In this section, we investigate the problem of computing justifications. First, we focus on the problem of finding *one* justification and then we discuss how to compute *all* of them. For such a purpose, we explore and discuss different techniques to tackle the problem. In any case, the process of finding justifications

[3] [6] describes the translation of a particular $\mathcal{SHOIN}(\mathcal{D})$ entailment test to a concept satisfiability check.

[4] It should be emphasized that this reduction, in logics with nominals, such as $\mathcal{SHOIN}$, holds for all standard entailments, including ABox statements. Thus our techniques extend to explaining inconsistent ontologies as well.

for an entailment consists of transforming the entailment to an unsatisfiable class (as described in the previous section) and then computing the justifications for that unsatisfiable class.

2.1 Computing a Single Justification

A Black-box Technique. The intuition behind our black box approach is simple: given a concept C unsatisfiable relative to $\mathcal{O}$, add axioms from $\mathcal{O}$ to a freshly generated ontology $\mathcal{O}'$ until C is found unsatisfiable with respect to $\mathcal{O}'$. We then prune extraneous axioms in $\mathcal{O}'$ until we arrive at a single minimal justification. Thus, the algorithm consists of two stages: (i) "expand" $\mathcal{O}'$ to find a superset of a justification and (ii)"shrink" to find the final justification. Each stage involves various satisfiability-check calls to the reasoner and the main aim of optimizations should be minimizing the number of satisfiability tests.

Table 1. Black Box Algorithm to find a Single Justification

Algorithm: SINGLE_JUST_ALG$_{Black-Box}$ **Input**: Ontology $\mathcal{O}$, Unsatisfiable concept C **Output**: Ontology $\mathcal{O}$'
(1) $\mathcal{O}' \leftarrow \emptyset$ (2)**while** (C is satisfiable w.r.t $\mathcal{O}$') (3) select a set of axioms $s \subseteq \mathcal{O}/\mathcal{O}'$ (4) $\mathcal{O}' \leftarrow \mathcal{O}' \cup s$ (5)perform fast pruning of $\mathcal{O}$' using a sliding window technique (6)**for each** axiom $k' \in \mathcal{O}'$ (7) $\mathcal{O}' \leftarrow \mathcal{O}' - \{k'\}$ (8) **if** (C is satisfiable w.r.t. $\mathcal{O}'$) (9) $\mathcal{O}' \leftarrow \mathcal{O}' \cup \{k'\}$

This algorithm, which we refer to as SINGLE_JUST_ALG$_{Black-Box}$(C, $\mathcal{O}$), shown in Table 1, is composed of two main parts: in the first loop, the algorithm generates an empty ontology $\mathcal{O}'$ and inserts into it axioms from $\mathcal{O}$ in each iteration, until the input concept C becomes unsatisfiable w.r.t $\mathcal{O}'$. In the second loop, the algorithm removes an axiom from $\mathcal{O}'$ in each iteration and checks whether the concept C turns satisfiable w.r.t. $\mathcal{O}'$, in which case the axiom is reinserted into $\mathcal{O}'$. The process continues until all axioms in $\mathcal{O}'$ have been tested.

In between the two loops, lies a key optimization stage of fast pruning of $\mathcal{O}$' that is output at the end of the first loop. The idea here is to use a window of n axioms, slide this window across the axioms in $\mathcal{O}'$, remove axioms from $\mathcal{O}'$ that lie within the window and determine if the concept is still unsatisfiable in the new $\mathcal{O}'$. If the concept turns satisfiable, we can conclude that at least one of the n axioms removed from $\mathcal{O}'$ is responsible for the unsatisfiability and hence we insert the n axioms back into $\mathcal{O}'$. However, if the concept still remains unsatisfiable, we can conclude that all n axioms are irrelevant and we remove them from $\mathcal{O}'$.

A Glass-box Technique. In our previous work [5], it was shown that for the logic $\mathcal{SHIF}(\mathcal{D})$, the computational overhead of tracking axioms internally within the tableau reasoner as they contribute to an inconsistency (a technique known as 'tableau tracing') is negligible, especially for time. That is, a satisfiability test while computing the justification (if the test is negative) is pragmatically as easy as performing the satisfiability test alone. Thus, if you have tracing implemented in your reasoner, it makes sense to use it, indeed, to leave it on in all but the most resource intensive situations.

We have improved on our tracing solution described in [5] by extending it to approximately cover $\mathcal{SHOIN}(D)$. We say "approximately" since the final output of the tracing algorithm is not ensured to be the justification itself, but may include a few extraneous axioms due to non-deterministic merge operations caused by max-cardinality restrictions. These extraneous axioms are pruned out using the same algorithm which is used in the final stage of the Black-Box approach. In the remainder of this paper, we refer to our $\mathcal{SHOIN}(\mathcal{D})$-extended Glass-box tracing algorithm as $\mathsf{SINGLE\_JUST\_ALG}_{\mathsf{Glass-Box}}$.

2.2 Computing All Justifications

Given an initial justification, we can use other techniques to compute the remaining ones. A plausible one is to employ a variation of the classical Hitting Set Tree (HST) algorithm [7]. This technique is both reasoner independent (Black-box) and, perhaps surprisingly, practically effective.

A Black-box Approach Using Reiter's HST. We first provide a short background on Reiter's HST Algorithm and then describe how it can be used to compute all justifications.

Reiter's general theory of diagnosis considers a system of components described by a *universal set* U, and a set $S \subseteq \mathcal{P}U$ of *conflict sets* (each conflict set is a subset of the system components responsible for the error), where $\mathcal{P}$ denotes the powerset operator. The set $T \subseteq U$ is a *hitting set* for S if each $s_i \in S$ contains at least one element of T, i.e. if $s_i \cap T \neq \emptyset$ for all $1 \leq i \leq n$ (in other words, T 'hits' or intersects each set in S).[5]. We say that T is a *minimal hitting set* for S if T is a hitting set for S no $T' \subset T$ is a hitting set for S. The *Hitting Set Problem* with input S, U is to compute all the minimal hitting sets for S. The problem is of interest to many kinds of *diagnosis* tasks and has found numerous applications.

Given a collection S of conflict sets, Reiter's algorithm constructs a labeled tree called *Hitting Set Tree* (HST). Nodes in an HST are labeled with a set $s \in S$, and edges are labeled with elements $\sigma \in \bigcup_{s \in S} s$. If $H(v)$ is the set of edge labels on the path from the root of the HST to the node v, then the label for v

[5] The significance of a hitting set for the collection of conflict sets is that in order to repair the system fully, at least one element in each of the conflict sets, i.e., its hitting set, needs to be removed from the ontology. Moreover, to keep system changes to a minimum, hitting sets should be as small as possible.

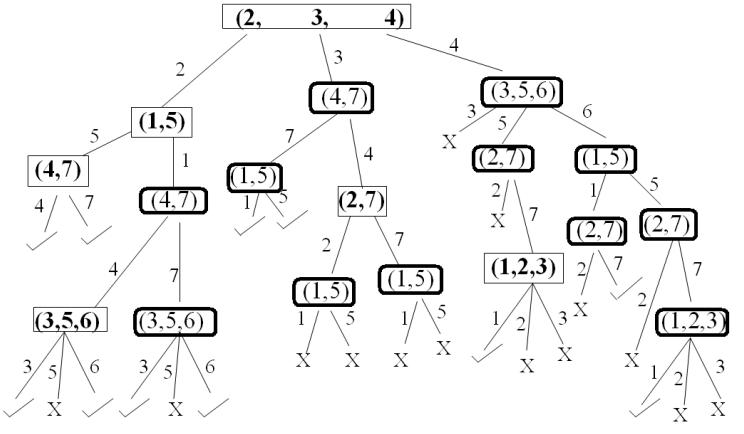

Fig. 2. Finding all Justifications using HST: Each distinct node is outlined in a rectangular box and represents a set in ALL_JUST$(\mathsf{C}, \mathcal{O})$. Total number of satisfiability tests is no. of distinct nodes (6) + nodes marked with $'\surd'$ (11) = 17.

is any $s \in S$ such that $s \cap H(v) = \emptyset$, if such a set exists. If s is the label of v, then for each element $\sigma \in s$, v has a successor w connected to v by an edge with σ in its label. If the label of v is the empty set, then $H(v)$ is a hitting set for S.

In our case, the universal set describing the system corresponds to the total set of axioms in the ontology, and a justification (for a particular concept unsatisfiability) corresponds to a single conflict set. While Reiter's algorithm is typically used to find all minimal hitting sets given a collection of conflict sets, it can also be used to dynamically find all conflict sets (justifications in our case), given the duality of the algorithm.

The idea is that given an algorithm to find a single justification for concept unsatisfiability, call it SINGLE_JUST_ALG (it could be either Black-box or Glass-box based as described in the previous section), we find any one justification and set it as the root node of the HST. We then remove each of the axioms in the justification individually, thereby creating new branches of the HST, and find new justifications along these branches on the fly (using SINGLE_JUST_ALG) in the modified ontology. This process needs to be exhaustively done in order to compute all justifications. The advantage of drawing parallels to Reiter's algorithm is that we can make use of all the optimizations presented in the latter to speed up the search.

The following example illustrates the approach.

Consider an ontology $\mathcal{O}$ with ten axioms and some unsatisfiable concept C. For the purpose of this example, we denote the axioms in $\mathcal{O}$ as natural numbers. We now show how to combine HST and SINGLE_JUST_ALG to compute ALL_JUST$(\mathsf{C} \sqsubseteq \bot, \mathcal{O})$. Figure 2 illustrates the whole process for our example. We anticipate that the expected outcome is the following:

$$\text{ALL\_JUST}(\mathsf{C} \sqsubseteq \bot, \mathcal{O}) = \{\{1,2,3\}, \{1,5\}, \{2,3,4\}, \{4,7\}, \{3,5,6\}, \{2,7\}\}.$$

The algorithm starts by executing SINGLE_JUST_ALG(C,$\mathcal{O}$) and let us assume that we obtain the set $S = \{2, 3, 4\}$ as an output. The next step is to initialize a Hitting Set Tree $\mathbf{T} = (V, E, \mathcal{L})$ with S in the label of its root, i.e. $V = \{v_0\}, E = \emptyset, \mathcal{L}(v_0) = S$. Then, it selects an arbitrary axiom in S, say 2, generates a new node w with an empty label in the tree and a new edge $\langle v_0, w\rangle$ with axiom 2 in its label. Then, the algorithm invokes SINGLE_JUST_ALG with arguments C and $\mathcal{O} - \{2\}$. In this case, it obtains a new justification for $C \models \bot$ w.r.t. $\mathcal{O} - \{2\}$, say $\{1, 5\}$. We add this set to S and also insert it in the label of the new node w.

The algorithm repeats this process, namely removing an axiom and executing the SINGLE_JUST_ALG algorithm to add a new node, until the concept turns satisfiable, in which case we mark the new node with a checkmark '$\surd$'.

There are two critical optimizations in Reiter's HST algorithm that help reduce the number of calls to SINGLE_JUST_ALG:

- **Early path termination:** Once a hitting set path is found, any superset of that path is guaranteed to be a hitting set as well, and thus no additional satisfiability tests are needed for that path, as indicated by a 'X' in the label of the node. Moreover, if all possible paths starting with the current edge path have been considered in a *previous* branch of the HST, the current path can be terminated early. For example, in Figure 2, the first path in the right-most branch of the root node is 4,3 and is terminated early since the algorithm has already considered all possible paths (hitting sets) containing axioms {3,4} in an earlier branch.
- **Justification reuse:** If the current edge path in any branch of the HST does not intersect with a previously found justification, then that justification is directly added as the new node to the branch. This is because the edge path represents axioms removed from the ontology, and if none of these removed axioms are present in a particular justification, that justification is guaranteed to exist in the ontology. Thus, we do not need to call SINGLE_JUST_ALG again to re-compute this justification. In Figure 2, *oval-bordered* nodes of the HST represent reused justifications.

Using the above optimizations, the total no. of calls to SINGLE_JUST_ALG, as shown in Figure 2, is reduced from 47 (in the exhaustive case) to only 17.

When the HST is fully built, the distinct nodes of the tree collectively represent the complete set of justifications of the unsatisfiable concept.

Definition of the Algorithm. The main component of the algorithm, which we refer to as ALL_JUST_ALG, is the recursive procedure SEARCH_HST, that effectively traverses a hitting set tree. The algorithm proceeds using a *seed* justification which is obtained by the SINGLE_JUST_ALGalgorithm. This seed is then used to construct a Hitting Set Tree (HST), which ultimately yields all justifications.

The correctness and completeness of the algorithm is given by Theorem 1.

Theorem 1. *Let C be unsatisfiable concept w.r.t $\mathcal{O}$. Then,* ALL_JUST_ALG(C, $\mathcal{O}$) = ALL_JUST(C, $\mathcal{O}$).

ALL_JUST_ALG(C, $\mathcal{O}$)
Input: Concept C and ontology $\mathcal{O}$
Output: Set S of justifications
(1) **Globals:** S $\leftarrow$ HS $\leftarrow \emptyset$
(2) just $\leftarrow$ SINGLE_JUST_ALG($C, \mathcal{O}$)
(3) S $\leftarrow$ S $\cup$ {just}
(4) $\alpha \leftarrow$ select some $i \in$ just
(5) path $\leftarrow \emptyset$
(6) SEARCH-HST(C, $\mathcal{O} \setminus \{\alpha\}, \alpha$, path)
(7) **return** S

SEARCH-HST(C, $\mathcal{O}, \alpha$, path)
Input: C and $\mathcal{O}$
α is the axiom that was removed
path is a set of axioms
Output: *none* — modifies globals S, HS
(1)
if path $\cup \{\alpha\} \subseteq h$ for some $h \in$ HS
(2) **or** there exists a prefix-path p for some $h \in$ HS s.t. p=path
(3) **return** (i.e., early path termination)
(4) **if** there exists *just* $\in$ S s.t. path $\cap$ *just* $= \emptyset$
(5) new-just $\leftarrow$ *just* (i.e., justifications reuse)
(6) **else**
(7) new-just $\leftarrow$ SINGLE_JUST_ALG($C, \mathcal{O}$)
(8) **if** new-just $\neq \emptyset$ (i.e., C is satisfiable relative to $\mathcal{O}$)
(9) S $\leftarrow$ S $\cup$ {new-just}
(10) new-path $\leftarrow$ path $\cup \{\alpha\}$
(11) **foreach** $\beta \in$ new-just
(12) SEARCH-HST($C, \mathcal{O} \setminus \{\beta\}, \beta$, new-path)
(13) **else**
(14) HS $\leftarrow$ HS $\cup$ path

Proof 1. ($\subseteq$)*: Let* $S \in$ ALL_JUST_ALG*(C, $\mathcal{O}$), then S belongs to the label of some non-leaf node w in the HST generated by the algorithm. In this case,* $\mathcal{L}(w) \in$ ALL_JUST(C, $\mathcal{O}'$)*, for some* $\mathcal{O}' \subseteq \mathcal{O}$*. Therefore,* $S \in$ ALL_JUST(C, $\mathcal{O}$)*.*

($\supseteq$)*: We prove by contradiction. Suppose there exists a set* $M \in$ ALL_JUST(C, $\mathcal{O}$)*, but* $M \notin$ ALL_JUST_ALG*(C, $\mathcal{O}$). In this case, M does not coincide with the label of any node in the HST. Let v_0 be the root of the tree, with* $\mathcal{L}(v_o) = \{\alpha_1, ..., \alpha_n\}$*. As a direct consequence of the completeness of Reiter's search strategy, the algorithm generates all the minimal Hitting Sets containing* α_i *for each* $i \in \{1, .., n\}$*. Every minimal hitting set U is s.t.* $U \cap M \neq \emptyset$*. This follows from the fact that in order to get rid of an entailment, at least one element from each of its justifications must be removed from the ontology, and hence the hitting set must intersect each justification. This implies that* $\alpha_i \in M$ *for* $1 \leq i \leq n$*, and therefore,* $\mathcal{L}(v_0) \subseteq M$*. However, since* $\mathcal{L}(v_0) \in$ ALL_JUST(C, $\mathcal{O}$) *and* $\mathcal{L}(v_0) \subseteq M$*, then* $M \notin$ ALL_JUST(C, $\mathcal{O}$)*, as it is a superset of an existing justification.*

Alternate Glass-box Approach to Computing All Justifications. While the Glass-box approach (tableau tracing) mentioned in Section 2.1 is used to find a *single* justification of an unsatisfiable concept, extending it to compute all the justifications is *not* straightforward. This is because computing all justifications amounts to saturating the completion graph generated by the DL reasoner (when testing the concept satisfiability) in order to explore *all* possible clashes. This, in effect, requires us to "turn off" many of the key optimizations in the reasoner. Since the excellent performance of current OWL reasoners critically depends on these optimization techniques, having to disable them renders this technique (currently) impractical. The optimizations (such as early clash detection or back-jumping) need to be reworked (if possible) to handle the fact that finding a single clash is no longer useful (in that it stops the search). For this reason our approach to finding all justifications uses the SEARCH_HSTalgorithm in combination with the SINGLE_JUST_ALG$_{Black-Box}$ or SINGLE_JUST_ALG$_{Glass-Box}$ algorithm.

3 Implementation and Evaluation

3.1 Implementation Details

We implemented SINGLE_JUST_ALG$_{Black-Box}$ and ALL_JUST_ALG using the latest version of the OWL API.[6] This version has excellent support for manipulating axioms and has fairly direct, low level wrappers for Pellet 1.4[8], and FaCT++ 1.1.7[9]. Such access is important for our experiments since the overhead of a remote access protocol can easily dominate processing time. [7]

Implementation of SINGLE_JUST_ALG$_{Black-Box}$. A critical piece in the "expand" stage of the SINGLE_JUST_ALG$_{Black-Box}$ algorithm is selecting *which* axioms to copy over from $\mathcal{O}$ into $\mathcal{O}'$. In our implementation, we run a loop that starts by inserting the concept definition axioms into $\mathcal{O}'$ and slowly expands $\mathcal{O}'$ to include axioms of structurally connected concepts, roles, and individuals (i.e., axioms which share terms in their signature). We vary the pace with which the fragment $\mathcal{O}'$ is expanded, initially considering few axioms to keep the size of $\mathcal{O}'$ bounded, and later allowing a large number of axioms into $\mathcal{O}'$ (at each iteration of the loop) if the concept continues to remain satisfiable in $\mathcal{O}'$.

In the fast pruning stage of SINGLE_JUST_ALG$_{Black-Box}$, we start with window size n being either one tenth of the size of the number of axioms to prune or just ten (whichever is greater). As pruning is repeated, we shrink the window size by some factor (currently, by 0.5). Pruning continues until the window size

[6] http://sourceforge.net/projects/owlapi

[7] We did not test with Racer Pro (http://www.racer-systems.com/) since, as of this writing, we only had DIG access to Racer Pro from the new OWL API and the overhead of DIG was proving unworkable. Given that Racer Pro often does better than the other reasoners and is usually in the rough ball park for normal reasoning, it is safe to extrapolate our results.

Table 2. Sample OWL Data used in our experiments. C=Classes, P=Properties, and I=Individuals in the ontology. Entailed are all the non-explicit subsumptions, including unsatisfiabilities, found during classification. These ontologies are available upon request.

Ontology	Expressivity	Axioms	C/P/I	Entailed	Domain
1. Generations	$\mathcal{ALCIF}$	335	22/4/0	24	Family tree
2. DOLCE-Lite	$\mathcal{SHOIN}(\mathcal{D})$	1417	200/299/39	3	Foundational
3. Economy	$\mathcal{ALH}(\mathcal{D})$	1704	338/53/481	51	Mid-level
4. MadCow	$\mathcal{ALCHOIN}(\mathcal{D})$	105	54/17/13	32	Tutorial
5. Tambis	$\mathcal{SHIN}$	800	395/ 100/ 0	65	Biological science
6. Sweet-JPL	$\mathcal{ALCHO}(\mathcal{D})$	3833	1537/ 121/ 150	183	Earthscience
7. Chemical	$\mathcal{ALCH}(\mathcal{D})$	254	48/20/0	43	Chemical elements
8. Transport	$\mathcal{ALH}(\mathcal{D})$	2051	444/93/183	52	Mid-level
9. MyGrid	$\mathcal{SHOIN}$	8179	550/69/13	297	Bioinformatics services
10. University	$\mathcal{SIOF}(\mathcal{D})$	169	30/12/4	23	Training
11. AminoAcids	$\mathcal{ALCF}$	2077	47/5/3	64	Classifies proteins
12. Sequence Ontology	$\mathcal{ALEHI}+$	1754	1248/17/9	179	The OBO (xp) sequence
13. Gene Ontology	$\mathcal{ALEHI}+$	1759	759/16/0	100	The OBO (xp) gene
14. MGED Ontology	$\mathcal{ALEF}(\mathcal{D})$	236	236/88/0	100	Microarray experiment

is 1 (i.e. equivalent to slow pruning), or until the size of the candidate justification stays constant between each round of pruning. Thereafter, slow pruning is performed until the candidate justification is verified as minimal.

Implementation of ALL_JUST_ALG. We implemented one additional optimization in the SEARCH_HST procedure to speed up the search. Since early path termination in the algorithm relies on the presence of previously detected hitting sets, it makes sense to find hitting sets as soon as possible during the Reiter search. Since a hitting set is basically a set that intersects all justifications, chances of getting a hitting set earlier in the search are higher if you give a higher priority to exploring axioms that are common to many justifications. For this purpose, we order axioms in a justification based on the commonality (or frequency) of the axiom across all the justifications currently found.

Our test data consists of 14 publicly available OWL ontologies that varied greatly in size, complexity, number of interesting entailments, and expressivity. See Table 2 for details. We classified each ontology to determine the unsatisfiable classes and the inferred atomic subsumptions. Since unsatisfiable classes and atomic subsumptions are the standard entailments exposed by ontology development environments, these are the natural explananda of interest. Note that 185 of the entailments are detecting unsatisfiable classes, whereas the remaining 587 represent "coherent" subsumptions. This is a bit distorted as several of the ontologies have "deliberate" bugs for tutorial purposes. But Chemical, e.g., is a production ontology, wherein (in this version) there are 37 unsatisfiable classes.

Since our sample reasoners do not handle precisely the same logic (FaCT++ has difficulty with certain datatypes), for the cross reasoner test we stripped out the problematic axioms. Since these constructs were not particularly heavily or complexly used, this seems harmless for the purpose of establishing the feasibility of these techniques.

All experiments have been performed on a MacBook Pro (Intel) 2.16 GHz Intel Core Due, with 2GB RAM, and 1.5GB (max) memory allotted to Java.

3.2 Experimental Results

First, we recorded the base classification time for each reasoner on all the ontologies (**Fig. 3a**). Then, for each entailment, we compared the performance of generating a *single* justification (**Fig. 3b**), and then *all* justifications (**Fig. 3c**). In both cases, Pellet and FaCT++ were used in the SINGLE_JUST_ALG$_{Black-Box}$ algorithm and Pellet with tableau tracing enabled for SINGLE_JUST_ALG$_{Glass-Box}$.

The first striking thing to notice is that the time to compute a justification for each entailment is in the sub second range on our setup. The second thing to notice is the excellent performance of SINGLE_JUST_ALG$_{Glass-Box}$ – in many cases (ontologies 4, 5, 6 for example) the time to compute a justification was so small that it was difficult to measure (around 1 ms). It should be noted that the times in **Fig. 3b** do not include the time to classify in the first place, which is, of course, needed to find the entailments. But, it does show that there is relatively little overhead for finding a single justification even with SINGLE_JUST_ALG$_{Black-Box}$. We believe that these times are perfectly acceptable for generating explanations on a practical basis in ontology development environments. In such situations, ontologies have typically been classified and users generate justifications on demand as they need them.

Similarly, as seen in **Fig. 3c**, the time to compute all justifications for all entailments in these ontologies using both blackbox and glassbox implementations of SINGLE_JUST_ALG as input to ALL_JUST is impressive. We are no longer solidly in second and sub-second range across the board (ontologies 7, 9 and 14 have dramatically lengthened experiment times), but for a wide range of purposes it is quite acceptable. Again, as would be expected from the results presented in **Fig. 3b**, ALL_JUST with SINGLE_JUST_ALG$_{Glass-Box}$ generally beat the black box technique (in certain cases by several orders of magnitude).

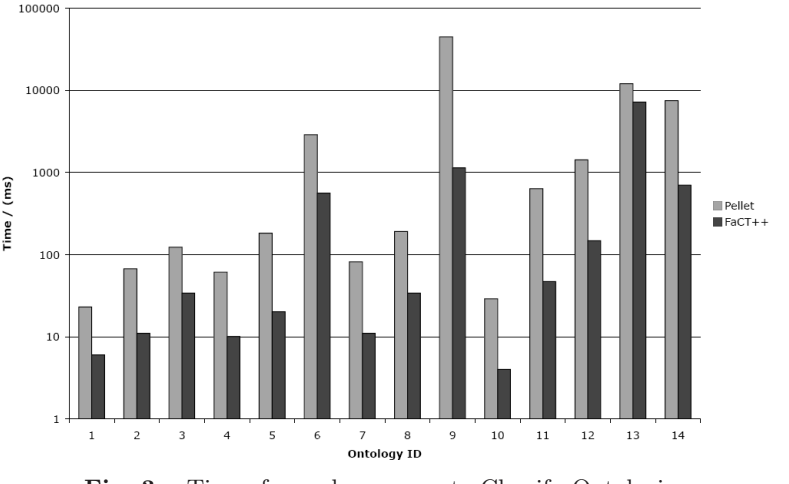

Fig. 3a. Times for each reasoner to Classify Ontologies

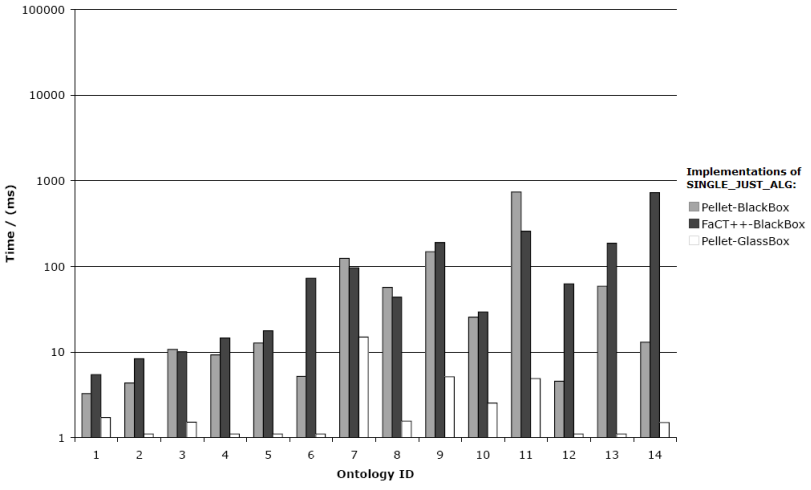

Fig. 3b. Times to Compute Single Justifications

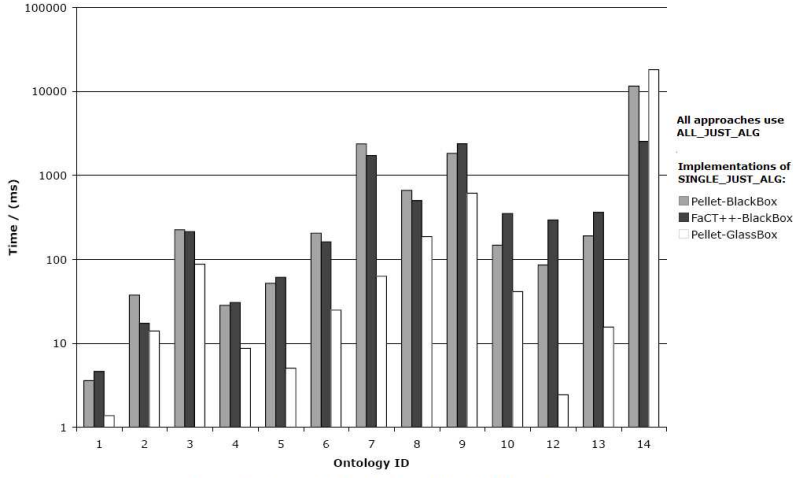

Fig. 3c. Times to Compute All Justifications

However, for ontology 14 (MGED), the reverse is true – ALL_JUST fared worse with $\mathsf{SINGLE\_JUST\_ALG}_{Glass-Box}$ than with $\mathsf{SINGLE\_JUST\_ALG}_{Black-Box}$, and, in any case, the time required is much greater than the corresponding time require to compute a single justification. The reason for this discrepancy is that in general, when computing a single justification using the blackbox technique, the reasoner only has to operate on a small fragment of the ontology. However, in the case of the glassbox approach a satisfiability check has to be performed on the whole ontology – when axioms are added or removed from the ontology this means that the satisfiability check must be rerun. While this does not pose

Table 3. Found justification statistics. Notice some very large justifications and some entailments with high numbers of justifications. In contrast, many ontologies had relatively small and few justifications.

	No. of Justifications		Justification Size	
Onts	Mean	Max	Mean	Max
1	1.00	1	1.79	2
2	1.00	1	1.00	1
3	1.29	2	3.61	6
4	1.09	2	2.33	6
5	1.00	1	4.12	13
6	1.12	3	2.04	7
7	9.86	26	7.99	12
8	2.33	5	5.53	9
9	1.39	8	5.39	40
10	1.65	5	5.04	10
12	1.00	1	1.50	7
13	1.04	3	1.24	7
14	1.00	1	2.00	2

a problem for many of the ontologies used in these tests, the time for Pellet to perform a satisfiability check on the MGED ontology was around 5 seconds – with multiple satisfiability checks the overall effect was noticeable.

In summary, the rather high performance of the black box case is notable. While typically beat by glass box justification generation, the total time generally stayed in the sub second range – it is arguable this performance is more than acceptable for use in ontology browsers and editors. From a pure performance perspective, it suggests that glass box techniques might not be necessary even as an optimization, at least in a wide range of cases. It should be noted that these constant factors can significantly add up if one is finding justifications for a large number of entailments (e.g., when caching a justification for each of the entailments found during classification). However, for debugging and on demand explanation it is clear that black box is sufficient for current ontologies, and has the great advantage of reasoner independence.

4 Related Work

There has been a lot of recent work done in capturing justifications for inconsistent ontologies or unsatisfiable concepts in relatively inexpressive description logic KBs using reasoner-dependent approaches. [10] describes a technique to find minimal sets of axioms responsible for an entailment (in this case, minimal inconsistent ABoxes) by labeling assertions, tracking labels through the tableau expansion process and using the labels of the clashes to arrive at a solution. The technique is limited to the logic $\mathcal{ALCF}$. Similar ideas can be seen in [11], where the motivation is debugging unsatisfiable concepts in the DICE terminology. [11] formalizes the problem including the specification of terms such as the MUPS (Minimal Unsatisfiability Preserving sub-TBox), which is the justification for the unsatisfiability entailment. [11] also describes an algorithm, restricted to $\mathcal{ALC}$ without general TBoxes, that relies on tableau saturation in order to find all

the MUPS for an unsatisfiable $\mathcal{ALC}$ concept and later compares this approach with a black-box implementation (in their system DION) not guaranteed to find all MUPS in [4]. On the other hand, we focus on a much more expressive logic OWL-DL (or $\mathcal{SHOIN}(\mathcal{D})$), provide a general definition of justification for any arbitrary OWL-DL entailment, present a sound, complete and highly optimized Glass-box and Black-box solution to finding all justifications, and demonstrate its feasibility on realistic expressive OWL-DL ontologies.

5 Conclusion

There is clearly room for further optimizations of all the algorithms presented in this paper, e.g., using more sophisticated analytical techniques for axiom selection in the 'expand' stage of the algorithm $\mathsf{SINGLE\_JUST\_ALG}_{\mathsf{Black-Box}}$. However, the overall message is clear: *reasoner independent techniques for finding all justifications for an OWL-DL entailment are practical.* We believe our implementation could be easily deployed in current OWL editors to offer explanation services that have been demonstrated to be useful [5] and are in high demand. In our own experience, we find having justifications completely change the way we work with ontologies for the enormously better. Indeed, we believe that no respectable OWL environment need lack for explanation support.

References

1. Noy, N., Sintek, M., Decker, S., Crubezy, M., Fergerson, R., Musen, M.: Creating semantic web contents with Protégé-2000. IEEE Intelligent Systems (2001)
2. Kalyanpur, A., Parsia, B., Sirin, E., Cuenca-Grau, B., Hendler, J.: Swoop: A web ontology editing browser. Journal of Web Semantics 4(2) (2006)
3. Kalyanpur, A., Parsia, B., Sirin, E., Grau, B.: Repairing unsatisfiable concepts in owl ontologies. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, Springer, Heidelberg (2006)
4. Schlobach, S., Huang, C.R., Van-Harmelen, F.: Debugging incoherent terminologies. In: Journal of Automated Reasoning (JAL) (in press, 2007)
5. Kalyanpur, A., Parsia, B., Sirin, E., Hendler, J.: Debugging unsatisfiable classes in OWL ontologies. Journal of Web Semantics 3(4) (2005)
6. Horrocks, I., Patel-Schneider, P.: Reducing OWL entailment to description logic satisfiability. In: International Semantic Web Conference (2003)
7. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32, 57–95 (1987)
8. Sirin, E., Parsia, B.: Pellet system description. In: Description Logics (DL) (2004)
9. Horrocks, I.: FaCT and iFaCT. In: Description Logics (1999)
10. Baader, F., Hollunder, B.: Embedding defaults into terminological knowledge representation formalisms (Technical Report RR-93-20)
11. Schlobach, S., Cornet, R.: Non-standard reasoning services for the debugging of description logic terminologies. In: Proc. of IJCAI 2003 (2003)