

A Tableaux-based Decision Procedure for Explanations in *SHOIN*

Aditya Kalyanpur¹, Bijan Parsia², Bernardo Cuenca Grau³, Evren Sirin⁴

Maryland Information and Network Dynamics Lab.

8400 Baltimore Av.

College Park, MD, 20740 USA

{aditya¹, evren⁴}@cs.umd.edu,

bparsia@isr.umd.edu², bernardo@mindswap.org³

Abstract

With the advent of OWL-DL, the expressive description logic *SHOIN* is exposed to a wider audience of ontology users and developers. In order to aid the usability of an OWL-DL based KR system, various non-standard reasoning services such as *inference explanation* need to be developed for *SHOIN*.

In this paper, we present a sound and complete tableaux-based decision procedure for identifying minimal fragments of a *SHOIN* knowledge base (KB) responsible for making a concept in it unsatisfiable. Based on this procedure, we develop a service that explains causes of contradictions in unsatisfiable concepts, and then extend it to explain arbitrary entailments of the KB.

Keywords: Description Logics, Ontology, Explanations

1 Introduction and Motivation

OWL-DL is a World Wide Web Consortium standard for representing ontologies on the Semantic Web [10]. It is a syntactic variant of the Description Logic $\mathcal{SHOIN}(\mathcal{D})$ [4] with an OWL-DL ontology corresponding to a $\mathcal{SHOIN}(\mathcal{D})$ knowledge base [9].

DL systems typically offer a set of standard inference services, such as concept classification, concept satisfiability and realization, knowledge base consistency checking, among others. These services are inter-definable, but it is standard to reduce all of them to KB consistency checking. Modern DL reasoners, such as FaCT, RACER and Pellet tackle the consistency problem using highly optimized tableaux-based decision procedures.

However, in order to be useful for real-world applications, a DL-based KR system must expose to the user a number of additional, non-standard services. A typical example is the generation of *explanations* for the inferences performed by the reasoner, such as inconsistencies in the knowledge base and entailed subsumption relations in the concept hierarchy. These services are critical, especially with the advent of the Semantic Web, which has exposed Ontology Engineering to a broader audience of users and developers.

A natural question is whether these services can be formalized as *reasoning* services in a way that is both useful and understandable for modelers. The first steps in this direction were taken in the early and mid nineties by the developers of the CLASSIC system [1]. CLASSIC incorporated a structural subsumption algorithm and was able to generate explanations for concept subsumption [8]. In the late nineties, the authors of [2] used a modified sequent calculus to explain tableaux-based subsumption proofs for the logic \mathcal{ALC} .

A different view was recently adopted in [11], where the trace of a tableaux reasoner was exploited to maintain a dependency relation between axioms in the KB and the inferences drawn from it. The motivation was debugging unsatisfiable concepts in the DICE terminology and the work focused on axiom and concept pinpointing in unfoldable \mathcal{ALC} TBoxes. The paper provided a formalization of the problem based on the notion of *Minimal Unsatisfiability Preserving Sub-TBoxes* (MUPS). Roughly, a MUPS for an atomic concept A is a minimal fragment of the KB in which A is unsatisfiable. Obviously, a concept may have several different MUPS within an ontology.

In our previous work [7], we extended this technique to the more expressive Description Logic \mathcal{SHIF} , provided an optimized implementation in the reasoner Pellet [12] and a UI in the ontology editor Swoop [6]. We have shown through a user study that these techniques are effective for debugging unsatisfiable classes and proposed various enhancements in the UI to improve

the explanation.

However, in [7], we did not modify the termination condition of the consistency checking tableaux decision procedure, i.e., the algorithm reports an inconsistency and stops whenever a contradiction is detected and no non-deterministic choices remain to be explored. Such an approach, though useful in practice, is limited to finding one MUPS for a given concept.

Finding *all* the MUPS of an unsatisfiable concept is, however, a critical task from a debugging point of view, since invariably, one needs to remove at least one axiom from each set in its MUPS in order to make the concept satisfiable.

In this paper, we present the following contributions:

- We extend our tracing algorithm to handle $\mathcal{SHOIN}(\mathcal{D})$ and hence OWL-DL. We provide a sound and complete decision procedure for the problem of finding all the MUPS for an unsatisfiable concept in an OWL-DL ontology and provide a preliminary implementation in the reasoner Pellet.
- We extend the applicability of the tableaux tracing technique for explaining *arbitrary entailments* and, hence, not just concept unsatisfiability. We provide a UI in the ontology editor Swoop.

2 Tableaux Tracing Algorithm

The MUPS for a concept A w.r.t a \mathcal{SHOIN} KB \mathbf{K} can be defined as follows:

Definition 1 (*MUPS*)

Let A be a concept, which is unsatisfiable w.r.t. a knowledge base \mathbf{K} . A fragment $\mathbf{K}' \subseteq \mathbf{K}$ is a MUPS of A in \mathbf{K} if A is unsatisfiable in \mathbf{K}' , and A is satisfiable in every $\mathbf{K}'' \subset \mathbf{K}'$.

We denote by $MUPS(A, \mathbf{K})$ the set of all the MUPS for A in \mathbf{K} . When the KB we are referring to is clear from the context, we will relax the notation and use $MUPS(A)$ instead.

We now present a tableaux-based algorithm for computing $MUPS(A, \mathbf{K})$. We assume that a reasoner has previously determined the consistency of \mathbf{K} and the unsatisfiability of A w.r.t. \mathbf{K} .

The algorithm is based on the tableaux-based decision procedure for concept satisfiability in \mathcal{SHOIN} recently presented in [4]. DL tableaux-based algorithms decide the satisfiability of a concept A w.r.t a KB \mathbf{K} by trying to construct (an abstraction of) a common model for A and \mathbf{K} , called a

completion graph. The graph is constructed by repeatedly applying a set of *expansion rules*. DL Tableaux algorithms are non-deterministic. Whenever a contradiction is encountered, a DL reasoner will either backtrack and select a different non-deterministic choice, or report the inconsistency and terminate, if no choice remains to be explored.

Obviously, for computing all the MUPS, the goal is no longer constructing a model for the input, but identifying which axioms in the input ontology are responsible for the contradictions that prevent the model from being built. Moreover, it is critical to record *all* the contradictions that could possibly invalidate the model. Thus, backtracking should no longer occur when a contradiction is encountered, but only when no more rules are applicable and other non-deterministic choices remain to be explored.

In order to identify the *causes* of contradictions, we introduce a *tracing function* that records the changes in the completion graph and the axioms and non-deterministic choices that are responsible for those changes to occur.

The tableaux algorithm runs on a completion graph and returns a collection S of fragments of the input knowledge base \mathbf{K} .

A completion graph for a concept A with respect to \mathbf{K} is a directed graph $\mathbf{G} = (V, E, \mathcal{L}, \neq)$. Each node $x \in V$ is labeled with a set of concepts $\mathcal{L}(x)$ and each edge $e = \langle x, y \rangle$ with a set $\mathcal{L}(e)$ of role names. The binary predicate \neq is used for recording inequalities between nodes.

If $\langle x, y \rangle \in E$, then y is called a *successor* of x and x a *predecessor* of y . *Ancestor* is the transitive closure of predecessor and *descendant* the transitive closure of successor. A node y is an R-successor of x as given in [4].

The collection S is initialized to the empty set and the completion graph to $\mathbf{G} = (\{v_0, \dots, v_l\}, \emptyset, \mathcal{L}, \emptyset, \emptyset)$, where $\mathcal{L}(v_i) = \{o_i\}$ for $1 \leq i \leq l$ and o_1, \dots, o_l are all the nominals occurring in \mathbf{K} and A . The graph \mathbf{G} is then expanded by repeatedly applying the rules in Table 1.

We have introduced two additional rules with respect to the ones presented in [4]: The *unfolding* rule adds the definition of a concept C to the label $\mathcal{L}(x)$ of a node x whenever C is contained in $\mathcal{L}(x)$. The GCI rule ($\rightarrow CE$) adds the disjunction $\neg C \sqcup D$ to the label of a node x if the GCI $C \sqsubseteq D$ is contained in \mathbf{K} . These rules are required in order to identify which axioms in \mathbf{K} are actually influencing the expansion of G .

Some of the expansion rules are *non-deterministic* and thus introduce *choice points*¹ in the expansion of \mathbf{G} . A choice point α can be characterized as a tuple $\alpha = (x, \mathcal{R}, C, l, LS_\alpha)$, where $x \in \mathbf{V}$ is a node, \mathcal{R} is a non-deterministic expansion rule, C is the concept in $\mathcal{L}(x)$ that triggers the application of

¹i.e. backtracking points

\mathcal{R} , $l = [c_1, \dots, c_n]$ is a list of choices available at the choice point (with a pointer marking the current choice), and $LS_\alpha = [S_{(\alpha, c_1)}, \dots, S_{(\alpha, c_n)}]$ is a list of collections that has a 1-1 correspondence with the choices, each collection $S_{(\alpha, c_i)}$ containing sets whose elements are fragments of the input knowledge base \mathbf{K} , and optionally, some choice points. A choice $c \in l$ is one of the following: 1) A concept C , if \mathcal{R} is the disjunction rule $\rightarrow \sqcup$; 2) A set of edges \mathbf{E}' , if \mathcal{R} is the at-most rule $\rightarrow \leq$; 3) a natural number m if \mathcal{R} is the $\rightarrow NN$ rule. We denote by Δ the ordered list of all choice points recorded during the expansion of \mathbf{G} and initialize it to the empty list.

The formal overhead introduced above is required to accurately compute the MUPS since the algorithm could detect a contradiction in the completion graph after making a choice, in which case, we need to consider if the contradiction *depends* on the choice or not, i.e., if it occurs strictly as a result of the choice. Intuitively, if the contradiction is independent of a choice, its trace (i.e., the axioms responsible for the contradiction) are likely to be a MUPS, whereas if it depends on a choice, its trace is insufficient to be a MUPS as we need to consider other choices at the choice point. To check this dependency condition (which is elaborated on later), we keep track of every choice c_i at the choice point α . Additionally, we maintain a collection $S_{(\alpha, c_i)}$ that contains elements responsible for creating contradictions in the completion graph after making choice c_i at the choice point α , and finally, when no more choices remain to be explored at the choice point, we combine the collections in LS_α together.

For ensuring termination, the algorithm in [4] applies the expansion rules with different priorities and incorporates a mechanism for cycle detection called *pair-wise blocking*. Both the rule priorities and the blocking condition are also required for determining $MUPS(A)$. The application of the expansion rules triggers a set of *events* that change the state of the completion graph, or the flow of the algorithm. We now describe each of the possible events in detail:

- **Add** $(C, \mathcal{L}(x))$ represents the action of adding a concept C to the label of a node x , i.e. the operation $\mathcal{L}(x) \leftarrow \mathcal{L}(x) \cup \{C\}$.
- **Add** $(R, \mathcal{L}(< x, y >))$ represents the addition of a role R to the label of an edge $< x, y >$, i.e., $\mathcal{L}(< x, y >) \leftarrow \mathcal{L}(< x, y >) \cup \{R\}$.
- $x = y$ is the action of *merging* the nodes x, y . A detailed description of the *merge* operation can be found in [4].
- $x \neq y$ stands for the addition of an inequality relation between two nodes x, y , where x, y , i.e. $\neq \leftarrow \neq \cup \{(x, y)\}$.
- **Add** $(g, Clash(\mathbf{G}))$ represents the detection of a clash g in the graph,

i.e. $\text{Clash}(\mathbf{G}) \leftarrow \text{Clash}(\mathbf{G}) \cup \{g\}$

We denote by \mathcal{E} the set of all events recorded during the execution of the algorithm. The completion graph \mathbf{G} contains a *clash* if ²:

1. $\{C, \neg C\} \subseteq L(x)$ for some concept C and some node $x \in V$
2. The events $x = y$ and $(x \neq y)$ belong to \mathcal{E} for some pair of nodes $x, y \in V$

We denote by $\text{Clash}(\mathbf{G})$ the set of all clashes in \mathbf{G}

The *tracing function* τ maps each event $e \in \mathcal{E}$ to a collection of sets, each set containing a fragment of \mathbf{K} (axioms) and optionally, some choice points. The function τ is initialized as empty and defined by construction as given in Table 1 ³ and for the following condition:

If a clash g is detected then:

- if g is of the form: $\{C, \neg C\} \subseteq \mathcal{L}(x)$,
then $\tau(\text{Add}(g, \text{Clash}(\mathbf{G}))) \leftarrow \tau(\text{Add}(C, \mathcal{L}(x))) \oplus \tau(\text{Add}(\neg C, \mathcal{L}(x)))$
- If g is of the form: events $x = y$ and $(x \neq y)$ belong to \mathcal{E}

then $\tau(\text{Add}(g, \text{Clash}(\mathbf{G}))) \leftarrow \tau(x = y) \oplus \tau(x \neq y)$

where the operator \oplus is defined as follows: let $\mathbf{S}_1 = \{S_1^1, \dots, S_m^1\}$ and $\mathbf{S}_2 = \{S_1^2, \dots, S_n^2\}$ be two collections of sets, then:

$$\mathbf{S}_1 \oplus \mathbf{S}_2 = \bigcup_{1 \leq i \leq m; 1 \leq j \leq n} S_i^1 \cup S_j^2$$

Now, each time a clash g is detected, we need to check if it depends on some non-deterministic choice or not. The dependency condition can be tested as follows: For every set $S' \in \tau(\text{Add}(g, \text{Clash}(\mathbf{G})))$ we check if there is some choice point $\alpha_i \in \Delta$, such that $\alpha_i \in S'$:

- If S' contains some α_i : we obtain all the choice points $\{\alpha_1.. \alpha_n\}$ in S' and determine the most recent choice point among these $\alpha_k, 1 \leq k \leq n$ from Δ . We then obtain the collection $S_{(\alpha_k, c_i)}$ associated with the most recent choice c_i from the pointer in l and update it as follows:
 $S_{(\alpha_k, c_i)} \leftarrow S_{(\alpha_k, c_i)} \cup S'$.
- If S' does not contain any α_i : we directly add S' to the collection S as follows: if S' is not a superset of any set already in S , we let $S \leftarrow S \cup S'$. On the other hand, if S' is a subset of a set S'' already in S , we replace S'' in S by S' .

²Note that this differs from the definition of a clash in [4], however, conditions (2) and (3) described in [4] reduce to condition (2) above.

³In Table 1, for simplicity, we have used a slightly abbreviated notation: we use $\tau(C, x)$ and $\tau(R, \langle x, y \rangle)$ instead of $\tau(\text{Add}(C, \mathcal{L}(x)))$ and $\tau(\text{Add}(R, \mathcal{L} \langle x, y \rangle))$ respectively.

<p>→ <i>unfold</i>: if $A \in \mathcal{L}(x)$, A atomic and $(A \sqsubseteq D) \in \mathbf{K}$, then $\tau(D, x) \leftarrow \tau(D, x) \cup (\tau(A, x) \oplus \{\{A \sqsubseteq D\}\})$ if $D \notin \mathcal{L}(x)$, then $Add(D, \mathcal{L}(x))$</p> <p>→ <i>CE</i>: if $(C \sqsubseteq D) \in \mathbf{K}$, with C not atomic, x not indirectly blocked then, for every node x, if $(\neg C \sqcup D) \notin \mathcal{L}(x)$, $Add(\neg C \sqcup D, \mathcal{L}(x))$</p> <p>→ \sqcap: if $(C_1 \sqcap C_2) \in \mathcal{L}(x)$, x is not indirectly blocked, then $\tau(C_1, x) \leftarrow \tau(C_1, x) \cup \tau((C_1 \sqcap C_2), x)$, $\tau(C_2, x) \leftarrow \tau(C_2, x) \cup \tau((C_1 \sqcap C_2), x)$ if $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$, then $Add(\{C_1, C_2\}, \mathcal{L}(x))$.</p> <p>→ \sqcup: if $(C_1 \sqcup C_2) \in \mathcal{L}(x)$, x is not indirectly blocked, if $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$, then Generate $\alpha = (x, \rightarrow \sqcup, C_1 \sqcup C_2, [C_1, C_2], [\emptyset, \emptyset])$ Do $\Delta \leftarrow [\Delta \alpha]$ and select $C_i, i \in \{1, 2\}$ $Add(C_i, \mathcal{L}(x))$ $\tau(C_i, x) \leftarrow \tau(C_i, x) \cup (\tau((C_1 \sqcup C_2), x) \oplus \{\{\alpha\}\})$</p> <p>→ \exists: if $\exists S.C \in \mathcal{L}(x)$, x is not blocked, $\tau(C, y) \leftarrow \tau(C, y) \cup \tau((\exists S.C), x)$ $\tau(S, \langle x, y \rangle) \leftarrow \tau(S, \langle x, y \rangle) \oplus \tau((\exists S.C), x)$ if x has no S-neighbor y with $C \in \mathcal{L}(y)$, then create new node y, $Add(S, \mathcal{L}(\langle x, y \rangle))$, $Add(C, \mathcal{L}(y))$</p> <p>→ \forall: if $\forall S.C \in \mathcal{L}(x)$, x is not indirectly blocked, and there is an S-neighbor y of x, then $\tau(C, y) \leftarrow \tau(C, y) \cup (\tau((\forall S.C), x) \oplus \tau(S, \langle x, y \rangle))$ if $C \notin \mathcal{L}(y)$, then $Add(C, \mathcal{L}(y))$</p> <p>→ \forall^+ if $\forall S.C \in \mathcal{L}(x)$, x is not indirectly blocked, and there is an R-neighbor y of x, $Trans(R)$ and $R \sqsubseteq S$, then $\tau((\forall S.C), y) \leftarrow \tau((\forall S.C), y) \cup (\tau((\forall S.C), x) \oplus \tau(R, \langle x, y \rangle) \oplus \{\{Trans(R)\}\} \oplus \{\{R \sqsubseteq S\}\})$ if $\forall S.C \notin \mathcal{L}(y)$ then $Add(\forall S.C, \mathcal{L}(y))$</p> <p>→ \geq: if $(\geq nS) \in \mathcal{L}(x)$, x is not blocked, $\tau(S, \langle x, y_i \rangle) \leftarrow \tau(S, \langle x, y_i \rangle) \cup \tau((\geq nS), x)$, for $1 \leq i \leq n$ $\tau(y_i \neq y_j) \leftarrow \tau(y_i \neq y_j) \cup \tau((\geq nS), x)$, for $1 \leq i < j \leq n$ if there are no n safe S-neighbors y_1, \dots, y_n of x with $y_i \neq y_j$ for $1 \leq i < j \leq n$, then create n new nodes y_1, \dots, y_n; $Add(S, \mathcal{L}(\langle x, y_i \rangle))$</p> <p>→ \leq: if $(\leq nS) \in \mathcal{L}(x)$, x is not indirectly blocked, and there are m S-neighbors y_1, \dots, y_m of x with $m > n$, then Select two S-neighbors of x: y, z Generate $\alpha = (x, \rightarrow \leq, \leq nS, [\{S_{x,y}, S_{x,z}\}, \emptyset])$ do $\Delta \leftarrow [\Delta \alpha]$ set $y = z$ $\tau(y = z) \leftarrow \tau(y = z) \cup (\tau((\leq nS), x) \oplus \tau(S, \langle x, y_1 \rangle) \oplus \tau(S, \langle x, y_m \rangle) \oplus \{\{\alpha\}\})$ 1. if z is a nominal node, Merge(y, z), 2. else if y is a nominal node or ancestor of z, Merge(z, y), 3. else Merge(y, z),</p> <p>→ O: if, for some $\{o\} \in N_I$, there are 2 nodes x, y with $\{o\} \in \mathcal{L}(x) \cap \mathcal{L}(y)$, $\tau(x = y) \leftarrow \tau(x = y) \cup (\tau(\{o\}, x) \oplus \tau(\{o\}, y))$ if not $x \neq y$, then Merge(x, y), and set $x = y$</p> <p>→ <i>NN</i>: if 1. $(\leq nS) \in \mathcal{L}(x)$, x is a nominal node, and there is a blockable S-neighbor y of x and x is a successor of y. 2. there is no m such that $1 \leq m \leq n$, $(\leq mS) \in \mathcal{L}(x)$, and there exist m nominal S-neighbors z_1, \dots, z_m of x s.t. $z_i \neq z_j, 1 \leq i < j \leq m$, then Generate $\alpha = (x, \rightarrow NN, \leq nS, [1..n], [\emptyset_1.. \emptyset_n])$ 1. guess m with $1 \leq m \leq n$. $Add(\leq mS, \mathcal{L}(x))$. $\tau((\leq mS), x) \leftarrow \tau((\leq mS), x) \cup (\tau((\leq nS), x) \oplus \tau(S, \langle y, x \rangle) \oplus \{\{\alpha\}\})$ 2. create m new nodes y_1, \dots, y_m. $Add(S, \mathcal{L}(\langle x, y_i \rangle))$, $Add(\{o_i\}, \mathcal{L}(y_i))$, for each $o_i \in N_I$ new in G and $y_i \neq y_j$ for $1 \leq i < j \leq m$. $\tau(S, \langle x, y_i \rangle) \leftarrow \tau(S, \langle x, y_i \rangle) \cup (\tau((\leq nS), x) \oplus \tau(S, \langle y, x \rangle) \oplus \{\{\alpha\}\})$ $\tau(\{o_i\}, y_i) \leftarrow \tau(\{o_i\}, y_i) \cup (\tau((\leq nS), x) \oplus \tau(S, \langle y, x \rangle) \oplus \{\{\alpha\}\})$</p>

Table 1: Modified Tableaux Expansion Rules

We say that \mathbf{G} is *saturated* if no more rules are applicable. When the completion graph saturates, the algorithm *backtracks* to the latest choice point α introduced in the expansion and selects the next choice from the list l . If no more choices are available for α , then α is removed from the tail of the list Δ . At this point, we obtain the list LS_α associated with the choice point α and create a new collection $S_\alpha = S_{(\alpha, c_1)} \dots \oplus S_{(\alpha, c_n)}$. Then, for every set $S' \in S_\alpha$, we check if S' contains some choice point α_i , $\alpha_i \in \Delta$ and $\alpha_i \neq \alpha$:

- if S' contains some α_i ($\neq \alpha$): we obtain all the choice points $\{\alpha_1 \dots \alpha_n\}$ in S' that are not equal to α and determine the most recent choice point among these α_k , $1 \leq k \leq n$ from Δ . We then obtain the most recent choice c_i from the pointer in l and set $S_{(\alpha_k, c_i)} \leftarrow S_{(\alpha_k, c_i)} \cup S_\alpha$.
- if S' does not contain any α_i ($\neq \alpha$): we directly update the global collection S by checking if S' is not a superset of any set already in S ($S \leftarrow S \cup S'$), or if S' is a subset of a set S'' already in S (replace S'' in S by S').

The algorithm terminates when \mathbf{G} is saturated and no more choice points remain to be explored (i.e. if Δ is the empty list). Note that the final collection S may contain choice points in addition to fragment of \mathbf{K} . The algorithm now removes all choice points from S and then returns S as output.

Theorem 1 (*Correctness and Completeness*)

Let A be an unsatisfiable concept in the signature of a consistent SHOIN knowledge base \mathbf{K} and let S be the output of the tableaux algorithm with input A , \mathbf{K} , then the algorithm terminates and $S = \mathit{MUPS}(A, \mathbf{K})$

For details of the proof of this theorem, we refer the reader to [5].

2.1 Tracing Example

We now demonstrate how the tableaux tracing algorithm works using an example (see Figure 1). In the figure, the algorithm is used to find the MUPS of the unsatisfiable concept A in an ontology that consists of 4 axioms (numbered as shown). The axioms are processed in increasing order and the completion graph is initialized with a single node x whose label $\mathcal{L}(x) = \{A\}$. The superscript for each concept in the label of a node (or role in the label of an edge) represents the value of the tracing function τ for the event that adds the concept (or role) to the particular node (or edge). These function values are updated iteratively based on the rules shown in Table 1. For readability sake, we avoid showing the trace explicitly for every event that occurs during tableaux expansion, instead key elements are highlighted in Figure 1.

Finding MUPS of unsatisfiable class A

Initialize $S \leftarrow \emptyset$

Axioms processed in the following order:

1. $A \sqsubseteq \neg B \sqcap \exists R.D \sqcap E \sqcap C$
2. $C \sqsubseteq \neg B \sqcap (\neg E \sqcup \forall R.F)$
3. $A \sqsubseteq B \sqcap C$
4. $F \sqsubseteq \neg D$

Axioms processed so far: 1,2

$$L(x) = \{ A^\emptyset, \dots, \neg B^{\{\{1\}, \{1,2\}\}}, \dots, E^{\{\{1\}\}}, \dots, (\neg E \sqcup \forall R.F)^{\{\{1,2\}\}}, \neg E^{\{\{1,2\}, \alpha\}} \}$$



(I)

First Clash g_1 Detected:

$\tau(Add(g_1, Clash(\mathbf{G}))) \leftarrow \tau(Add(E, x)) \oplus \tau(Add(\neg E, x))$
 $\tau(Add(g_1, Clash(\mathbf{G}))) \leftarrow \{\{1,2,\alpha\}\}$
 $S' \in \tau(Add(g_1, Clash(\mathbf{G})))$ contains α with choice c_1 .
Hence, $S_{(\alpha, c_1)} \leftarrow \{\{1,2,\alpha\}\}$
 $LS_\alpha \leftarrow [\{\{1,2,\alpha\}\}, \emptyset]$
 $S \leftarrow \emptyset$ (unchanged)

Axioms processed so far: 1,2,3

$$L(x) = \{ A^\emptyset, \dots, \neg B^{\{\{1\}, \{1,2\}\}}, \dots, \neg E^{\{\{1,2,\alpha\}\}}, (B \sqcap C)^{\{\{3\}\}}, B^{\{\{3\}\}} \}$$



(II)

Second Clash g_2 Detected:

$\tau(Add(g_2, Clash(\mathbf{G}))) \leftarrow \tau(Add(B, x)) \oplus \tau(Add(\neg B, x))$
 $\tau(Add(g_2, Clash(\mathbf{G}))) \leftarrow \{\{1,3\}, \{1,2,3\}\}$
 $\forall S' \in \tau(Add(g_2, Clash(\mathbf{G})))$, S' does not contain α .
Hence,
 $S \leftarrow \{\{1,3\}\}$ (**Note:** $\{1,2,3\} \supset \{1,3\}$ is not added to S)
 $LS_\alpha \leftarrow [\{\{1,2,\alpha\}\}, \emptyset]$ (unchanged)

Axioms processed so far: 1,2,3,4

$$L(x) = \{ A^\emptyset, \dots, \exists R.D^{\{\{1\}\}}, \dots, (\neg E \sqcup \forall R.F)^{\{\{1,2\}\}}, \forall R.F^{\{\{1,2,\alpha\}\}}, \dots \}$$



$$L(x,y) = \{ R^{\{\{1\}\}} \}$$



$$L(y) = \{ D^{\{\{1\}\}}, F^{\{\{1,2,\alpha\}\}}, \neg D^{\{\{1,2,\alpha,4\}\}} \}$$

(III)

Third Clash g_3 Detected:

$\tau(Add(g_3, Clash(\mathbf{G}))) \leftarrow \tau(Add(D, y)) \oplus \tau(Add(\neg D, y))$
 $\tau(Add(g_3, Clash(\mathbf{G}))) \leftarrow \{\{1,2,\alpha,4\}\}$
 $S' \in \tau(Add(g_3, Clash(\mathbf{G})))$ contains α with choice c_2
Hence, $S_{(\alpha, c_2)} \leftarrow \{\{1,2,\alpha,4\}\}$
 $LS_\alpha \leftarrow [\{\{1,2,\alpha\}\}, \{\{1,2,\alpha,4\}\}]$
 $S \leftarrow \{\{1,3\}\}$ (unchanged)

Final Computation:

$S_\alpha \leftarrow \{\{1,2,\alpha\}\} \oplus \{1,2,\alpha,4\}$; $S_\alpha \leftarrow \{\{1,2,4,\alpha\}\}$
 $S \leftarrow S \cup S_\alpha$; $S \leftarrow \{\{1,3\}, \{1,2,4,\alpha\}\}$

After removing choice point:

$S \leftarrow \{\{1,3\}, \{1,2,4\}\}$

$S \leftarrow \{$
 $\{A \sqsubseteq \neg B \sqcap \exists R.D \sqcap E \sqcap C, A \sqsubseteq B \sqcap C\},$
 $\{A \sqsubseteq \neg B \sqcap \exists R.D \sqcap E \sqcap C,$
 $\quad C \sqsubseteq \neg B \sqcap (\neg E \sqcup \forall R.F), F \sqsubseteq \neg D\},$
 $\}$

Thus, $S = \text{MUPS}(A)$

Figure 1: Example Walkthrough for the Tableau Tracing Algorithm

In region (I), an interesting point to note is that after processing axioms 1, 2, the superscript of the concept $\neg B$ in $\mathcal{L}(x)$ is $\{\{1\}, \{1,2\}\}$, which captures both the ways in which $\neg B$ can be added to $\mathcal{L}(x)$, even though $\neg B$ is not duplicated in the label.

Also in (I), the algorithm makes a non-deterministic choice due to the concept $(\neg E \sqcup \forall R.F)$ present in axiom 2. The choice point generated in this case, α , is specified by the tuple $(x, \rightarrow \sqcup, (\neg E \sqcup \forall R.F), [\neg E, \forall R.F], [\emptyset, \emptyset])$, and the first choice the algorithm makes c_1 is to add the concept $\neg E$ to the label of node x . This generates a clash g_1 due to the presence of both E and $\neg E$ in $\mathcal{L}(x)$, which is added to $Clash(\mathbf{G})$. Now, computing the value of $\tau(Add(g_1, Clash(\mathbf{G})))$ gives a single set S' which includes the choice point α

whose most recent choice is c_1 . Therefore, S' is added to $S_{(\alpha, c_1)}$.

In region (II), the algorithm detects a second clash g_2 after processing the first conjunct (B) in axiom 3, and the clash occurs due to both B and $\neg B$ in $\mathcal{L}(x)$. In this case, $\tau(\text{Add}(g_2, \text{Clash}(\mathbf{G}))$ contains two sets, both of which do not include the choice point α . Also, one of the sets in $\tau(\text{Add}(g_2, \text{Clash}(\mathbf{G}))$ is a subset of the other. Therefore, the minimal set $\{1,3\}$ is added to S .

In region (III), the algorithm explores the second choice $c_2 = (\forall R.F)$ at the choice point α (after no more rules can be applied given the first choice) and adds it to the label of node x . It now detects a third clash g_3 after creating a successor node y , and adding both D and $\neg D$ to $\mathcal{L}(y)$. In this case, $\tau(\text{Add}(g_3, \text{Clash}(\mathbf{G}))$ contains a single set S' , which includes the choice point α whose most recent choice is c_2 . Therefore, S' is added to $S_{(\alpha, c_2)}$.

Finally, when no more rules are applicable, and both choices in the choice point α have been explored, the algorithm combines both collections of LS_α into a single collection S_α . Now, since no choice points remain in Δ , it adds the value of S_α to S . S is then pruned by removing the choice point and returned as output of the algorithm.

3 Explaining Arbitrary Entailments

Theorem 1 shows that our tableaux tracing algorithm finds all the MUPS of an unsatisfiable *SHOIN* concept w.r.t. a knowledge base \mathbf{K} . However, in general, our technique can be used to explain arbitrary entailments in \mathbf{K} . Since every entailment in \mathbf{K} can be reduced to a concept satisfiability check w.r.t \mathbf{K} , the problem of finding the sets of axioms responsible for an entailment to occur reduces to identifying the MUPS for some unsat. concept.

For example, consider a pair of concepts C, D in \mathbf{K} such that $\mathbf{K} \models C \sqsubseteq D$. It is not hard to see that $\mathbf{K} \models C \sqsubseteq D$ iff the concept $C \sqcap \neg D$ is unsatisfiable. Thus, the minimal sets of axioms responsible for the subsumption to be entailed are precisely given by $MUPS(C \sqcap \neg D, \mathbf{K})$. A suitable reduction can be found for any axiom entailed by \mathbf{K} .

4 Preliminary Evaluation

We have implemented the tableaux tracing algorithm for *SHOIN* in our reasoner Pellet. Pellet is a sound and complete tableaux reasoner for *SHOIN*(\mathcal{D}) and provides an easily extensible architecture, which enabled us to implement our algorithm without affecting the core internals of the reasoner. This, in effect, allowed an external application to use Pellet both, as a standard OWL-DL reasoner or as a debugger, as and when needed.

Our first cut implementation was designed to find only one MUPS for an unsatisfiable *SHOIN* concept. Not surprisingly, the performance results were similar to our earlier *SHIF* tracing solution, i.e., our *SHOIN* tableaux tracing algorithm introduced very little memory overhead and only marginal increase in the running time of the normal consistency checking procedure. However, extending the algorithm to continue until saturation in order to find all the MUPS obviously affected the performance adversely. A summary of our initial results is shown in the table below.

Ontology (sat. tests)	Single MUPS % inc. (avg. ms)	All MUPS % inc. (avg. ms)
Koala (3)	0.04 (16.66)	87.5 (30)
Mad-Cow (1)	0 (30)	133.3 (70)
University (8)	1.25 (18)	227.5 (52.4)
Sweet-JPL (1)	6 (350)	48.7 (491)
Galen (10)	4 (1300)	548 (8100)

Table 2: Computational Performance

We selected five OWL-DL ontologies⁴ and compared the performance of satisfiability tests on the unsatisfiable concepts in the ontology⁵ using a normal reasoning algorithm, a modified tracing algorithm that terminates after determining the first MUPS, and a full tracing version which computes all the MUPS by saturating the tableaux. For the latter, all the standard tableaux optimizations [3] such as early clash detection, dependency directed backjumping, semantic branching etc. were disabled.

The table displays the percentage increase and the average runtime for the satisfiability tests. As can be seen, the overhead was marginal when computing a single MUPS, and substantial when computing all the MUPS. Note that there were a few cases in the Galen ontology which timed out before reaching saturation, which is understandable given the complexity of the problem, but there were a lot of cases that saturated successfully and the table lists results for 10 such tests randomly selected.

Interestingly, though finding all the MUPS involves tableaux saturation, the situation is not as bleak as it seems. There are two main reasons for this:

- We only need to undergo saturation for a few select satisfiability tests
- A critical reasoning optimization, *absorption* [3], is not precluded by our approach

⁴The ontologies are available at <http://www.mindswap.org/ontologies/debugging>.

⁵For the Galen ontology, there were no unsatisfiable classes found, and so 10 satisfiability tests were randomly selected, each for explaining a separate subsumption.

To elaborate, we use the standard optimized tableaux reasoning procedure to check ontology consistency and identify the unsatisfiable concepts in it first. We only need to resort to our tracing algorithm, which saturates the tableaux, on the unsatisfiable concepts separately⁶. Thus, the total number of satisfiability tests that need to be run till saturation is kept to a small size.

Also, note that saturation is feasible in the absence of extensive non-determinism in the tableaux, and our tracing algorithm can make use of the pre-processing optimization, *absorption*, which eliminates non-determinism arising from General Concept Inclusion axioms (GCIs) in the KB. Obviously, our results are still preliminary and we need to test further to get a better understanding of real-world use case characteristics.

Having developed this tableaux tracing service in Pellet, we exposed its functionality in our hypermedia-inspired OWL Ontology Editor, Swoop. Pellet is integrated into Swoop as one of its default reasoners for displaying inferences while browsing or editing terms in OWL ontologies.

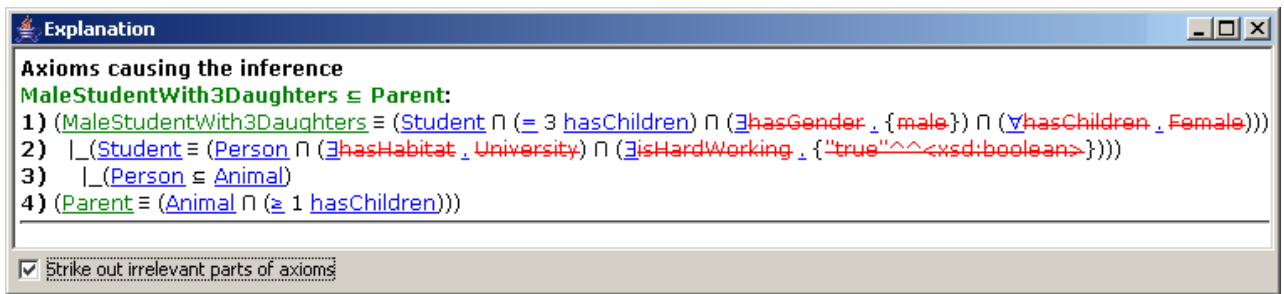


Figure 2: **Explaining arbitrary entailments using Tableaux Tracing:** Displaying the minimal set of axioms from the ontology (with key entities highlighted and irrelevant parts struck out) responsible for the subsumption `MaleStudentWith3Daughters ⊆ Parent` in the Koala ontology.

To incorporate explanations for inferred entailments via the Swoop UI, we added a “*Why?*” hyperlink next to each inference. When the hyperlink is clicked, we use the tracing service in Pellet to extract and present to the user, the axioms responsible for the inference (thus the explanations are generated on demand only). The axioms are ordered and highlighted as described in [7]. Also note that though generating the complete explanation requires saturation for a single satisfiability test, a worst case delay of 5-10 seconds is not unreasonable from a Swoop user’s perceptive.

To further improve the explanation, we modified the tableaux tracing code to tag axiom components responsible for the inconsistency. This was

⁶For explaining an arbitrary entailment, which is the case for Galen, the problem again reduces to a single satisfiability test.

achieved by following the trace back from the clash through each of the expansion rules and marking components in this traceback responsible for the clash (see details in [5]). Axiom tagging enabled the option in the UI to strike out irrelevant parts of axioms that do not contribute to the inference (see Figure 2). Though our tagging implementation is still incomplete, we tested it on a few cases and found that it made a significant difference in the presentation and thus in the overall efficacy of our solution. Thus far, reactions from our user base has been very encouraging.

5 Conclusion and Future Work

In this paper, we have presented a sound and complete decision procedure for the problem of finding all the MUPS for an unsatisfiable concept in an OWL-DL ontology. Based on this procedure, we have formulated a non-standard reasoning service that is used to explain and debug unsatisfiable concepts in the ontology by presenting, in a suitable manner, the precise axioms responsible for the contradiction. The service is then extended to explain any arbitrary entailment in the ontology, making it a significant addition to the overall usability of a real-world OWL-DL based KR system.

An initial worry with our theoretical approach is its reliance on tableaux saturation to find all the MUPS, making our solution seem impractical. However, it is important to remember that explanation generation is usually done on demand, and that the problem reduces to a single satisfiability test. Our preliminary evaluation using Pellet has shown that undergoing saturation for a satisfiability test is not unfeasible for a lot of real world cases (especially by using absorption to eliminate non-determinism). In situations where the algorithm times out before reaching saturation, it is observed to have found at least one MUPS, which can be presented to the user at that point.

As future work, we plan on improving the performance of the tableaux tracing algorithm to find all the MUPS as follows: instead of running it once and proceeding till saturation, we can run it several times, each time terminating when the first inconsistency is detected, and ensuring that a different clash and hence distinct MUPS is exposed. This can be achieved by *re-ordering* the tableaux expansion rules in future runs of the completion based on clashes discovered in its earlier runs (see [5]).

Besides fine tuning our implementation, our goal is to explore the use of axiom tracing for various ontology engineering tasks (for a list of potential uses, see [5]). We intend on conducting a more thorough usability evaluation to determine if our axiom-based solution for explanation of entailments is as useful as our debugging techniques for unsatisfiable concepts.

References

- [1] A. Borgida, R. Brachman, D. McGuinness, and L. Resnick. Classic: A structural data model for objects. In *SIGMOD*, 1989.
- [2] A. Borgida, E. Franconi, I. Horrocks, D. McGuinness, and P. Patel-Schneider. Explaining alc subsumption. In *Proceedings of the International Workshop on Description Logics - DL-99*, 1999.
- [3] I. Horrocks. Implementation and optimisation techniques. In Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors, *The Description Logic Handbook: Theory, Implementation, and Applications*, pages 306–346. Cambridge University Press, 2003.
- [4] Ian Horrocks and Ulrike Sattler. A tableaux decision procedure for SHOIQ. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*. Morgan Kaufman, 2005.
- [5] A. Kalyanpur, B. Parsia, B. Cuenca-Grau, and E. Sirin. Tableaux tracing for *SHOIN*. Technical report, University of Maryland Institute for Advanced Computer Studies (UMIACS), 2005-66, 2005. Available online at <http://www.mindswap.org/papers/TR-tracingSHOIN.pdf>.
- [6] A. Kalyanpur, B. Parsia, E. Sirin, B. Cuenca-Grau, and J. Hendler. Swoop: A web ontology editing browser. *Journal of Web Semantics*, 2005. To Appear.
- [7] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, and James Hendler. Debugging unsatisfiable classes in owl ontologies. *Journal of Web Semantics*, 2005. To Appear.
- [8] D. McGuinness. *Explaining Reasoning in Description Logics*. PhD thesis, New Brunswick, New Jersey, 1996.
- [9] P. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax W3C Recommendation. <http://www.w3.org/tr/2004/rec-owl-semantic-20040210/>. February 2004.
- [10] P.F. Patel-Schneider, P. Hayes, and I. Horrocks. Web ontology language OWL Abstract Syntax and Semantics. *W3C Recommendation*, 2004.

- [11] S. Schlobach and R. Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In *Proceedings of IJCAI, 2003*, 2003.
- [12] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. Technical report, University of Maryland Institute for Advanced Computer Studies (UMIACS), 2005-68, 2005. Available online at <http://www.mindswap.org/papers/PelletDemo.pdf>.