A Description Logics Tableau Reasoner in Prolog

Riccardo Zese[1], Elena Bellodi[1], Evelina Lamma[1], and Fabrizio Riguzzi[2]

[1] Dipartimento di Ingegneria – University of Ferrara
[2] Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[riccardo.zese,elena.bellodi,evelina.lamma,fabrizio.riguzzi]@unife.it

Abstract. Description Logics (DLs) are gaining a widespread adoption as the popularity of the Semantic Web increases. Traditionally, reasoning algorithms for DLs have been implemented in procedural languages such as Java or C++. In this paper, we present the system TRILL for "Tableau Reasoner for descrIption Logics in proLog". TRILL answers queries to $\mathcal{SHOIN}(\mathbf{D})$ knowledge bases using a tableau algorithm. Prolog non-determinism is used for easily handling non-deterministic expansion rules that produce more than one tableau. Moreover, given a query, TRILL is able to return instantiated explanations for the query, i.e., instantiated minimal sets of axioms that allow the entailment of the query. The Thea2 library is exploited by TRILL for parsing ontologies and for the internal Prolog representation of DL axioms.

Keywords: Description Logics, Tableau, Prolog, Semantic Web

1 Introduction

The Semantic Web aims at making information available in a form that is understandable by machines [9]. In order to realize this vision, the World Wide Web Consortium has supported the development of the Web Ontology Language (OWL), a family of knowledge representation formalisms for defining ontologies. OWL is based on Description Logics (DLs), a set of languages that are restrictions of first order logic (FOL) with decidability and for some of them low complexity. For example, the OWL DL sublanguage is based on the expressive $\mathcal{SHOIN}(\mathbf{D})$ DL while OWL 2 corresponds to the $\mathcal{SROIQ}(\mathbf{D})$ DL [9].

In order to fully support the development of the Semantic Web, efficient DL reasoners, such us Pellet, RacerPro, FaCT++ and HermiT, must be available to extract implicit information from the modeled ontologies. Most DL reasoners implement a tableau algorithm in a procedural language. However, some tableau expansion rules are non-deterministic, requiring the developers to implement a search strategy in an or-branching search space. Moreover, in some cases we want to compute all explanations for a query, thus requiring the exploration of all the non-deterministic choices done by the tableau algorithm.

In this paper, we present the system TRILL for "Tableau Reasoner for descrIption Logics in proLog", a tableau reasoner for the $\mathcal{SHOIN}(\mathbf{D})$ DL implemented in Prolog. Prolog's search strategy is exploited for taking into account

non-determinism of the tableau rules. TRILL uses the Thea2 library [27] for parsing OWL in its various dialects. Thea2 translates OWL files into a Prolog representation in which each axiom is mapped into a fact.

TRILL can check the consistency of a concept and the entailment of an axiom from an ontology and return "instantiated explanations" for queries, a non-standard reasoning service that is useful for debugging ontologies and for performing probabilistic reasoning. Instantiated explanations record, besides the axioms necessary to entail the query, also the individuals involved in the application of the axioms. This service was used in [21] for doing inference from DL knowledge bases under the probabilistic DISPONTE semantics [20].

Our ultimate aim is to use TRILL for performing probabilistic reasoning. The availability of a Prolog implementation of a DL reasoner will also facilitate the development of a probabilistic reasoner for integrations of probabilistic logic programming [23] with probabilistic DLs.

In the following, section 2 briefly introduces $\mathcal{SHOIN}(\mathbf{D})$ and its translation into predicate logic. Section 3 defines the problem we are trying to solve while Section 4 illustrates related work. Section 5 discusses the tableau algorithm used by TRILL and Section 6 describes TRILL's implementation. Section 7 shows preliminary experiments and Section 8 concludes the paper.

2 Description Logics

Description Logics (DLs) are knowledge representation formalisms that possess nice computational properties such as decidability and/or low complexity, see [1, 2] for excellent introductions. DLs are particularly useful for representing ontologies and have been adopted as the basis of the Semantic Web.

While DLs can be translated into FOL, they are usually represented using a syntax based on concepts and roles. A concept corresponds to a set of individuals of the domain while a role corresponds to a set of couples of individuals of the domain. We now briefly describe $\mathcal{SHOIN}(\mathbf{D})$.

Let $\mathbf{A}$, $\mathbf{R}$ and $\mathbf{I}$ be sets of *atomic concepts*, *roles* and *individuals*, respectively. A *role* is either an atomic role $R \in \mathbf{R}$ or the inverse R^- of an atomic role $R \in \mathbf{R}$. We use $\mathbf{R}^-$ to denote the set of all inverses of roles in $\mathbf{R}$. An *RBox* $\mathcal{R}$ consists of a finite set of *transitivity axioms* $Trans(R)$, where $R \in \mathbf{R}$, and *role inclusion axioms* $R \sqsubseteq S$, where $R, S \in \mathbf{R} \cup \mathbf{R}^-$. *Concepts* are defined by induction as follows. Each $C \in \mathbf{A}$ is a concept, $\bot$ and $\top$ are concepts, and if $a \in \mathbf{I}$, then $\{a\}$ is a concept called *nominal*. If C, C_1 and C_2 are concepts and $R \in \mathbf{R} \cup \mathbf{R}^-$, then $(C_1 \sqcap C_2)$, $(C_1 \sqcup C_2)$, and $\neg C$ are concepts, as well as $\exists R.C$, $\forall R.C$, $\geq nR$ and $\leq nR$ for an integer $n \geq 0$. A *TBox* $\mathcal{T}$ is a finite set of *concept inclusion axioms* $C \sqsubseteq D$, where C and D are concepts. We use $C \equiv D$ to abbreviate the conjunction of $C \sqsubseteq D$ and $D \sqsubseteq C$. An *ABox* $\mathcal{A}$ is a finite set of *concept membership axioms* $a : C$, *role membership axioms* $(a, b) : R$, *equality axioms* $a = b$ and *inequality axioms* $a \neq b$, where C is a concept, $R \in \mathbf{R}$ and $a, b \in \mathbf{I}$. A *knowledge base* (KB) $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$ consists of a TBox $\mathcal{T}$, an RBox $\mathcal{R}$ and an ABox $\mathcal{A}$. A knowledge base $\mathcal{K}$ is usually assigned a semantics in terms of

set-theoretic interpretations and models of the form $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ where $\Delta^{\mathcal{I}}$ is a non-empty *domain* and $\cdot^{\mathcal{I}}$ is the *interpretation function* that assigns an element in $\Delta^{\mathcal{I}}$ to each $a \in \mathbf{I}$, a subset of $\Delta^{\mathcal{I}}$ to each $C \in \mathbf{A}$ and a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to each $R \in \mathbf{R}$.

The semantics of DLs can be given equivalently by converting a KB into a predicate logic theory and then using the model-theoretic semantics of the resulting theory. A translation of $\mathcal{SHOIN}$ into First-Order Logic with Counting Quantifiers is given in the following as an extension of the one given in [24]. We assume basic knowledge of logic. In predicate logic, a concept is a unary predicate symbol while a role is a binary predicate symbol. The translation uses two functions π_x and π_y that map concept expressions to logical formulas, where π_x is given by

$$\begin{aligned}
\pi_x(A) &= A(x) & \pi_x(\neg C) &= \neg\pi_x(C)\\
\pi_x(\{a\}) &= (x = a) & \pi_x(C \sqcap D) &= \pi_x(C) \wedge \pi_x(D)\\
\pi_x(C \sqcup D) &= \pi_x(C) \vee \pi_x(D) & \pi_x(\exists R.C) &= \exists y.R(x,y) \wedge \pi_y(C)\\
\pi_x(\exists R^-.C) &= \exists y.R(y,x) \wedge \pi_y(C) & \pi_x(\forall R.C) &= \forall y.R(x,y) \rightarrow \pi_y(C)\\
\pi_x(\forall R^-.C) &= \forall y.R(y,x) \rightarrow \pi_y(C) & \pi_x(\geq nR) &= \exists^{\geq n} y.R(x,y)\\
\pi_x(\geq nR^-) &= \exists^{\geq n} y.R(y,x) & \pi_x(\leq nR) &= \exists^{\leq n} y.R(x,y)\\
\pi_x(\leq nR^-) &= \exists^{\leq n} y.R(y,x)
\end{aligned}$$

and π_y is obtained from π_x by replacing x with y and vice-versa. Table 1 shows the translation of each axiom of $\mathcal{SHOIN}$ knowledge bases into predicate logic.

Axiom	Translation
$C \sqsubseteq D$	$\forall x.\pi_x(C) \rightarrow \pi_x(D)$
$R \sqsubseteq S$	$\forall x,y.R(x,y) \rightarrow S(x,y)$
$Trans(R)$	$\forall x,y,z.R(x,y) \wedge R(y,z) \rightarrow R(x,z)$
$a : C$	$\pi_a(C)$
$(a,b) : R$	$R(a,b)$
$a = b$	$a = b$
$a \neq b$	$a \neq b$

Table 1. Translation of $\mathcal{SHOIN}$ axioms into predicate logic.

$\mathcal{SHOIN}(\mathbf{D})$ adds to $\mathcal{SHOIN}$ datatype roles, i.e., roles that map an individual to an element of a datatype such as integers, floats, etc. Then new concept definitions involving datatype roles are added that mirror those involving roles introduced above. We also assume that we have predicates over the datatypes.

A query Q over a KB $\mathcal{K}$ is usually an axiom for which we want to test the entailment from the knowledge base, written $\mathcal{K} \models Q$. The entailment test may be reduced to checking the unsatisfiability of a concept in the knowledge base, i.e., the emptiness of the concept.

$\mathcal{SHOIN}(\mathbf{D})$ is decidable if there are no number restrictions on non-simple roles. A role is non-simple iff it is transitive or has transitive subroles.

Given a predicate logic formula F, a *substitution* θ is a set of pairs x/a, where x is a variable universally quantified in the outermost quantifier in F and $a \in \mathbf{I}$. The application of θ to F, indicated by $F\theta$, is called an *instantiation of* F and is obtained by replacing x with a in F and by removing x from the external quantification for every pair x/a in θ. Formulas not containing variables are called *ground*. A substitution θ is *grounding* for a formula F if $F\theta$ is ground.

Example 1. The following KB is inspired by the ontology `people+pets` [16]:
$\exists hasAnimal.Pet \sqsubseteq NatureLover \quad \mathit{fluffy} : Cat \quad \mathit{tom} : Cat \quad Cat \sqsubseteq Pet$
$(\mathit{kevin}, \mathit{fluffy}) : hasAnimal \quad (\mathit{kevin}, \mathit{tom}) : hasAnimal$
It states that individuals that own an animal which is a pet are nature lovers and that *kevin* owns the animals *fluffy* and *tom*. Moreover, *fluffy* and *tom* are cats and cats are pets. The predicate logic formulas equivalent to the axioms are $F_1 = \forall x.\exists y.hasAnimal(x, y) \wedge Pet(y) \rightarrow NatureLover(x)$, $F_2 = hasAnimal(\mathit{kevin}, \mathit{fluffy})$, $F_3 = hasAnimal(\mathit{kevin}, \mathit{tom})$, $F_4 = Cat(\mathit{fluffy})$, $F_5 = Cat(\mathit{tom})$ and $F_6 = \forall x.Cat(x) \rightarrow Pet(x)$. The query $Q = \mathit{kevin} : NatureLover$ is entailed by the KB.

3 Querying KBs in $\mathcal{SHOIN}$(D)

Traditionally, a reasoning algorithm decides whether an axiom is entailed or not by a KB by refutation: axiom E is entailed if $\neg E$ has no model in the KB. Besides deciding whether an axiom is entailed by a KB, we want to find also *instantiated* explanations for the axiom.

The problem of finding explanations for a query has been investigated by various authors [25, 11, 13, 7, 12]. It was called *axiom pinpointing* in [25] and considered as a non-standard reasoning service useful for tracing derivations and debugging ontologies. In particular, Schlobach and Cornet [25] define *minimal axiom sets* or *MinAs* for short.

Definition 1 (MinA). *Let $\mathcal{K}$ be a knowledge base and Q an axiom that follows from it, i.e., $\mathcal{K} \models Q$. We call a set $M \subseteq \mathcal{K}$ a* minimal axiom set *or* MinA *for Q in $\mathcal{K}$ if $M \models Q$ and it is minimal w.r.t. set inclusion.*

The problem of enumerating all MinAs is called MIN-A-ENUM in [25]. ALL-MINAS$(Q, \mathcal{K})$ is the set of all MinAs for query Q in knowledge base $\mathcal{K}$.

However, in some cases, besides ALL-MINAS$(Q, \mathcal{K})$, we may want to know also the individuals to which the axioms were applied. We call this problem *instantiated axiom pinpointing*.

In instantiated axiom pinpointing we are interested in instantiated minimal sets of axioms that entail an axiom. An *instantiated axiom set* is a finite set $\mathcal{F} = \{(F_1, \theta_1), \ldots, (F_n, \theta_n)\}$ where $F_1, \ldots, F_n$ are axioms contained in $\mathcal{K}$ and $\theta_1, \ldots, \theta_n$ are substitutions. Given two instantiated axiom sets $\mathcal{F} = \{(F_1, \theta_1), \ldots, (F_n, \theta_n)\}$ and $\mathcal{E} = \{(E_1, \delta_1), \ldots, (E_m, \delta_m)\}$, we say that $\mathcal{F}$ *precedes* $\mathcal{E}$, written $\mathcal{F} \preceq \mathcal{E}$, iff, for each $(F_i, \theta_i) \in \mathcal{F}$, there exists an $(E_j, \delta_j) \in \mathcal{E}$ and a substitution η such that $F_j\theta_j = E_i\delta_i\eta$.

Definition 2 (InstMinA). *Let $\mathcal{K}$ be a knowledge base and Q an axiom that follows from it, i.e., $\mathcal{K} \models Q$. We call $\mathcal{F} = \{(F_1, \theta_1), \ldots, (F_n, \theta_n)\}$ an* instantiated minimal axiom set *or* InstMinA *for Q in $\mathcal{K}$ if $\{F_1\theta_1, \ldots, F_n\theta_n\} \models Q$ and $\mathcal{F}$ is minimal w.r.t. precedence.*

Minimality w.r.t. precedence means that axioms in an InstMinA are as instantiated as possible. We call INST-MIN-A-ENUM the problem of enumerating all InstMinAs. ALL-INSTMINAS$(Q, \mathcal{K})$ is the set of all InstMinAs for the query Q in knowledge base $\mathcal{K}$.

Example 2. The query $Q = kevin : NatureLover$ of Example 1 has two MinAs (in predicate logic): $\{\ hasAnimal(kevin, fluffy),\ Cat(fluffy),\ \forall x.Cat(x) \rightarrow Pet(x),\ \forall x.\exists y.hasAnimal(x, y) \wedge Pet(y) \rightarrow NatureLover(x)\}$ and $\{\ hasAnimal(kevin, tom),\ Cat(tom),\ \forall x.Cat(x) \rightarrow Pet(x),\ \forall x.\exists y.hasAnimal(x, y) \wedge Pet(y) \rightarrow NatureLover(x)\}$. The corresponding InstMinAs are $\{hasAnimal(kevin, fluffy),\ Cat(fluffy) \rightarrow Pet(fluffy),\ Cat(fluffy),\ hasAnimal(kevin, fluffy) \wedge Pet(fluffy) \rightarrow NatureLover(kevin)\}$ and $\{\ hasAnimal(kevin, tom),\ Cat(tom),\ Cat(tom) \rightarrow Pet(tom),\ hasAnimal(kevin, tom) \wedge Pet(tom) \rightarrow NatureLover(kevin)\}$.

Instantiated axiom pinpointing is useful for a more fine-grained debugging of the ontology: by highlighting the individuals to which axioms are applied, it may point to parts of the ABox to be modified for repairing the KB. INST-MIN-A-ENUM is also required to support reasoning in probabilistic DLs, in particular in those that follow the DISPONTE probabilistic semantics [20, 19].

4 Related Work

Usually, DL reasoners implement a tableau algorithm using a procedural language. Since some tableau expansion rules are non-determinsitic, the developers have to implement a search strategy from scratch. Moreover, in order to solve MIN-A-ENUM, all different ways of entailing an axiom must be found. For example, Pellet [26] is a tableau reasoner for OWL written in Java and able to solve MIN-A-ENUM. It computes ALL-MINAS$(Q, \mathcal{K})$ by finding a single MinA using the tableau algorithm and then applying the *hitting set algorithm* [17] to find all the other MinAs. This is a black box method: Pellet repeatedly removes an axiom from the KB and then computes again a MinA recording all the different MinAs so found. Recently, BUNDLE [21] was proposed for reasoning over KBs following the DISPONTE probabilistic semantics. BUNDLE computes the probability of queries by solving the INST-MIN-A-ENUM problem. BUNDLE is based on Pellet's source code and modifies it for recording the individuals to which the axioms are applied. As in Pellet, it uses a black box method to compute ALL-INSTMINAS$(Q, \mathcal{K})$.

Reasoners written in Prolog can exploit Prolog's backtracking facilities for performing the search. This has been observed in various works. In [3] the authors proposed a tableau reasoner in Prolog for FOL based on free-variable semantic

tableaux. However, the reasoner is not tailored to DLs. SWI Prolog [28] has an RDF and Semantic Web library but is more focused on storing and querying RDF triples, while it has limited support for OWL reasoning. Meissner [15] presented the implementation of a Prolog reasoner for the DL $\mathcal{ALCN}$. This work was the basis of [8], that considered $\mathcal{ALC}$ and improved [15] by implementing heuristic search techniques to reduce the running time. Faizi [6] added to [8] the possibility of returning explanations for queries but still handled only $\mathcal{ALC}$.

In [10] the authors presented the KAON2 algorithm that exploits basic superposition, a refutational theorem proving method for FOL with equality, and a new inference rule, called decomposition, to reduce a $\mathcal{SHIQ}$ KB into a disjunctive datalog program, while DLog [14] is an ABox reasoning algorithm for the $\mathcal{SHIQ}$ language that allows to store the content of the ABox externally in a database and to respond to instance check and instance retrieval queries by transforming the KB into a Prolog program. TRILL differs from these work for the considered DL and from DLog for the capability of answering general queries.

5 The Tableau Algorithm

A *tableau* is an ABox. It can also be seen as a graph G where each node represents an individual a and is labeled with the set of concepts $\mathcal{L}(a)$ it belongs to. Each edge $\langle a, b\rangle$ in the graph is labeled with the set of roles $\mathcal{L}(\langle a, b\rangle)$ to which the couple (a, b) belongs. A tableau algorithm proves an axiom by refutation: it starts from a tableau that contains the negation of the axiom and applies the tableau expansion rules. For example, if the query is a class assertion, $C(a)$, we add $\neg C$ to the label of a. If we want to test the emptyness (inconsistency) of a concept C, we add a new anonymous node a to the tableau and add C to the label of a. The axiom $C \sqsubseteq D$ can be proved by showing that $C \sqcap \neg D$ is empty. A *tableau algorithm* repeatedly applies a set of consistency preserving *tableau expansion rules* until a clash (i.e., a contradiction) is detected or a clash-free graph is found to which no more rules are applicable. A clash is, for example, a concept C and a node a where C and $\neg C$ are present in the label of a, i.e. $\{C, \neg C\} \subseteq \mathcal{L}(a)$. If no clashes are found, the tableau represents a model for the negation of the query, which is thus not entailed.

In TRILL we use the tableau expansion rules for $\mathcal{SHOIN}(\mathbf{D})$ shown in Figure 1 that are similar to those of Pellet [11]. Each expansion rule updates as well a *tracing function* τ, which associates sets of axioms with labels of nodes and edges. It maps couples (concept, individual) or (role, couple of individuals) to a fragment of the knowledge base $\mathcal{K}$. τ is initialized as the empty set for all the elements of its domain except for $\tau(C, a)$ and $\tau(R, \langle a, b\rangle)$ to which the values $\{a : C\}$ and $\{(a, b) : R\}$ are assigned if $a : C$ and $(a, b) : R$ are in the ABox respectively. The output of the tableau algorithm is a set S of axioms that is a fragment of $\mathcal{K}$ from which the query is entailed.

For ensuring the termination of the algorithm, TRILL, as Pellet, uses a special condition known as *blocking* [11]. In a tableau a node x can be a *nominal*

node if its label $\mathcal{L}(x)$ contains a *nominal* or a *blockable* node otherwise. If there is an edge $e = \langle x, y \rangle$ then y is a *successor* of x and x is a *predecessor* of y. *Ancestor* is the transitive closure of predecessor while *descendant* is the transitive closure of successor. A node y is called an *R-neighbour* of a node x if y is a successor of x and $R \in \mathcal{L}(\langle x, y \rangle)$, where $R \in \mathbf{R}$.

An R-neighbour y of x is *safe* if (i) x is blockable or if (ii) x is a nominal node and y is not blocked. Finally, a node x is *blocked* if it has ancestors x_0, y and y_0 such that all the following conditions are true: (1) x is a successor of x_0 and y is a successor of y_0, (2) y, x and all nodes on the path from y to x are blockable, (3) $\mathcal{L}(x) = \mathcal{L}(y)$ and $\mathcal{L}(x_0) = \mathcal{L}(y_0)$, (4) $\mathcal{L}(\langle x_0, x \rangle) = \mathcal{L}(\langle y_0, y \rangle)$. In this case, we say that y *blocks* x. A node is blocked also if it is blockable and all its predecessors are blocked; if the predecessor of a safe node x is blocked, then we say that x is indirectly blocked.

Since we want to solve also the INST-MIN-A-ENUM problem, we modified the tableau expansion rules of Pellet to return a set of pairs (axiom, substitution) instead of a set of axioms. The tracing function τ now stores, together with information regarding concepts and roles, also information concerning individuals involved in the expansion rules, which will be returned at the end of the derivation process together with the axioms. In Figure 1, $(A \sqsubseteq D, a)$ is the abbreviation of $(A \sqsubseteq D, \{x/a\})$, $(R \sqsubseteq S, a)$ of $(R \sqsubseteq S, \{x/a\})$, $(R \sqsubseteq S, a, b)$ of $(R \sqsubseteq S, \{x/a, y/b\})$, $(\mathit{Trans}(R), a, b)$ of $(\mathit{Trans}(R), \{x/a, y/b\})$ and $(\mathit{Trans}(R), a, b, c)$ of $(\mathit{Trans}(R), \{x/a, y/b, z/c\})$, with a, b, c individuals and x, y, z variables contained in the logical translation of the axioms (Table 1). The most important modifications of Pellet's tableau algorithm are in the rules $\rightarrow \forall^+$ and $\rightarrow \forall$. For rule $\rightarrow \forall^+$, we record in the explanation a transitivity axiom for the role R in which only two individuals, those connected by the super role S, are involved. For rule $\rightarrow \forall$, we make a distinction between the case in which $\forall S_1.C$ was added to $\mathcal{L}(a_1)$ by a chain of applications of $\rightarrow \forall^+$ or not. In the first case, we fully instantiate the transitivity and subrole axioms. In the latter case, we simply obtain $\tau(C, b)$ by combining the explanation of $\forall S_1.C(a_1)$ with that of $(a_1, b) : S_1$. To clarify how the rules $\rightarrow \forall$ and $\rightarrow \forall^+$ work we now give two examples.

Example 3. Let us consider the query $Q = ann : Person$ for the following knowledge base:

$kevin : \forall kin.Person \quad (kevin, lara) : relative \quad (lara, eva) : ancestor$
$(eva, ann) : ancestor \quad Trans(ancestor) \quad Trans(relative)$
$relative \sqsubseteq kin \quad ancestor \sqsubseteq relative$

TRILL first applies the $\rightarrow \forall^+$ rule to $kevin$, adding $\forall relative.Person$ to the label of $lara$. The tracing function τ is (in predicate logic):

$\tau(\forall relative.Person, lara) = \{\ \forall y.kin(kevin, y) \rightarrow Person(y),$
$relative(kevin, lara),\ relative(kevin, lara) \rightarrow kin(kevin, lara),$
$\forall z.relative(kevin, lara) \wedge relative(lara, z) \rightarrow relative(kevin, z)\}$

Note that the transitivity axiom is not fully instantiated, the variable z is still present. Then TRILL applies the $\rightarrow \forall^+$ rule to $lara$ adding $\forall ancestor.Person$ to eva. The tracing function τ is (in predicate logic):

Deterministic rules:

$\rightarrow$ *unfold*: **if** $A \in \mathcal{L}(a)$, A atomic and $(A \sqsubseteq D) \in K$, **then**
  if $D \notin \mathcal{L}(a)$, **then**
    $\mathcal{L}(a) := \mathcal{L}(a) \cup \{D\}$
    $\tau(D, a) := \tau(A, a) \cup \{(A \sqsubseteq D, a)\}$

$\rightarrow$ *CE*: **if** $(C \sqsubseteq D) \in K$, **then**
  if $(\neg C \sqcup D) \notin \mathcal{L}(a)$, **then**
    $\mathcal{L}(a) := \mathcal{L}(a) \cup \{\neg C \sqcup D\}$
    $\tau(\neg C \sqcup D, a) := \{(C \sqsubseteq D, a)\}$

$\rightarrow \sqcap$: **if** $(C_1 \sqcap C_2) \in \mathcal{L}(a)$, **then**
  if $\{C_1, C_2\} \not\subseteq \mathcal{L}(a)$, **then**
    $\mathcal{L}(a) := \mathcal{L}(a) \cup \{C_1, C_2\}$
    $\tau(C_i, a) := \tau((C_1 \sqcap C_2), a)$

$\rightarrow \exists$: **if** $\exists S.C \in \mathcal{L}(a)$, **then**
  if a has no S-neighbor b with $C \in \mathcal{L}(b)$, **then**
    create new node b, $\mathcal{L}(b) := \{C\}$, $\mathcal{L}(\langle a, b\rangle) := \{S\}$,
    $\tau(C, b) := \tau((\exists S.C), a)$, $\tau(S, \langle a, b\rangle) := \tau((\exists S.C), a)$

$\rightarrow \forall$: **if** $\forall S_1.C \in \mathcal{L}(a_1)$, a_1 is not indirectly blocked and there is an S_1-neighbor b of a_1, **then**
  if $C \notin \mathcal{L}(b)$, **then** $\mathcal{L}(b) := \mathcal{L}(a) \cup \{C\}$
  if there is a chain of individuals $a_2, \ldots, a_n$ and roles $S_2, \ldots, S_n$ such that
    $\bigcup_{i=2}^{n} \{(Trans(S_{i-1}), a_i, a_{i-1}), (S_{i-1} \sqsubseteq S_i, a_i, a_{i-1})\} \subseteq \tau(\forall S_1.C, a_1)$
    and $\neg\exists a_{n+1} : \{(Trans(S_n), a_{n+1}, a_n), (S_n \sqsubseteq S_{n+1}, a_{n+1})\} \subseteq \tau(\forall S_1.C, a_1)$, **then**
    $\tau(C, b) := \tau(\forall S_1.C, a_1) \setminus \bigcup_{i=2}^{n} \{(Trans(S_{i-1}), a_i, a_{i-1}), (S_{i-1} \sqsubseteq S_i, a_i, a_{i-1})\} \cup$
    $\bigcup_{i=2}^{n} \{(Trans(S_{i-1}), a_i, a_{i-1}, b), (S_{i-1} \sqsubseteq S_i, a_i, b)\} \cup \tau(S_1, \langle a_1, b\rangle)$
  else
    $\tau(C, b) := \tau(\forall S_1.C, a_1) \cup \tau(S_1, \langle a_1, b\rangle)$

$\rightarrow \forall^{+}$: **if** $\forall(S.C) \in \mathcal{L}(a)$, a is not indirectly blocked
    and there is an R-neighbor b of a, $Trans(R)$ and $R \sqsubseteq S$, **then**
  if $\forall R.C \notin \mathcal{L}(b)$, **then** $\mathcal{L}(b) := \mathcal{L}(b) \cup \{\forall R.C\}$
    $\tau(\forall R.C, b) := \tau(\forall S.C, a) \cup \tau(R, \langle a, b\rangle) \cup \{(Trans(R), a, b), (R \sqsubseteq S, a, b)\}$

$\rightarrow \geq$: **if** $(\geq nS) \in \mathcal{L}(a)$, a is not blocked, **then**
  if there are no n safe S-neighbors $b_1, ..., b_n$ of a with $b_i \neq b_j$, **then**
    create n new nodes $b_1, ..., b_n$; $\mathcal{L}(\langle a, b_i\rangle) := \{S\}$;
    add in the ABox $\neq (b_i, b_j)$
    $\tau(S, \langle a, b_i\rangle) := \tau((\geq nS), a)$
    $\tau(\neq (b_i, b_j)) := \tau((\geq nS), a)$

$\rightarrow O$: **if**, $\{o\} \in \mathcal{L}(a) \cap \mathcal{L}(b)$ and not $a \neq b$, **then**
  $Merge(a, b)$
  $\tau(Merge(a, b)) := \tau(\{o\}, a) \cup \tau(\{o\}, b)$
  For each concept C_i in $\mathcal{L}(a)$, $\tau(C_i, b) := \tau(C_i, a) \cup \tau(Merge(a, b))$
  (similarly for roles merged, and correspondingly for concepts in $\mathcal{L}(b)$)

Non-deterministic rules:

$\rightarrow \sqcup$: **if** $(C_1 \sqcup C_2) \in \mathcal{L}(a)$ and a is not indirectly blocked, **then**
  if $\{C_1, C_2\} \cap \mathcal{L}(a) = \emptyset$, **then**
    Generate graphs $G_i := G$ for each $i \in \{1, 2\}$, $\mathcal{L}(a) := \mathcal{L}(a) \cup \{C_i\}$ for each $i \in \{1, 2\}$
    $\tau(C_i, a) := \tau((C_1 \sqcup C_2), a)$

$\rightarrow \leq$: **if** $(\leq nS) \in \mathcal{L}(a)$, a is not indirectly blocked
    and there are m S-neighbors $b_1, ..., b_m$ of a with $m > n$, **then**
  For each possible pair b_i, b_j, $1 \leq i, j \leq m; i \neq j$ **then**
    Generate a graph $G_k := G$
    $\tau(Merge(b_i, b_j)) := (\tau((\leq nS), a) \cup \tau(S, \langle a, b_1\rangle) ... \cup \tau(S, \langle a, b_m\rangle))$
    For each concept C_i in $\mathcal{L}(b_i)$, $\tau(C_i, b_j) := \tau(C_i, b_i) \cup \tau(Merge(b_i, b_j))$
    (similarly for roles merged, and correspondingly for concepts in $\mathcal{L}(b_j)$)

Fig. 1. TRILL tableau expansion rules for OWL DL.

$\tau(\forall ancestor.Person, eva) = \{\ \forall y.kin(kevin, y) \rightarrow Person(y),$
$relative(kevin, lara),\ \mathbf{ancestor(lara, eva)},$
$\forall z.relative(kevin, lara) \wedge relative(lara, z) \rightarrow relative(kevin, z),$
$\forall \mathbf{z}.\mathbf{ancestor(lara, eva)} \wedge \mathbf{ancestor(eva, z)} \rightarrow \mathbf{ancestor(lara, z)},$
$relative(kevin, lara) \rightarrow kin(kevin, lara),$
$\mathbf{ancestor(lara, eva)} \rightarrow \mathbf{relative(lara, eva)}\ \}$

Now TRILL applies the $\rightarrow \forall$ rule to *eva* adding $Person$ to the label of *ann*. The tracing function τ is (in predicate logic):

$\tau(Person, ann) = \{\ \forall y.kin(kevin, y) \rightarrow Person(y),$
$relative(kevin, lara),\ ancestor(lara, eva),\ \mathbf{ancestor(eva, ann)},$
$relative(kevin, lara) \wedge relative(lara, \mathbf{ann}) \rightarrow relative(kevin, \mathbf{ann}),$
$ancestor(lara, eva) \wedge ancestor(eva, \mathbf{ann}) \rightarrow ancestor(lara, \mathbf{ann}),$
$relative(kevin, \mathbf{ann}) \rightarrow kin(kevin, \mathbf{ann}),$
$ancestor(lara, \mathbf{ann}) \rightarrow relative(lara, \mathbf{ann})\ \}$

Here the chain of transitivity and subrole axioms becomes ground. At this point the tableau contains a clash so the algorithm stops and returns the explanation given by $\tau(Person, ann)$.

It is easy to see that the explanation entails the axiom represented by the arguments of τ. In general, the following theorem holds.

Theorem 1. *Let Q be an axiom entailed by $\mathcal{K}$ and let S be the output of a reasoner with the tableau expansion rules of Figure 1, such as TRILL, with input Q and $\mathcal{K}$. Then $S \in$ All-InstMinAs$(Q, \mathcal{K})$.*

Proof. The full details of the proof are given in [18], Theorem 5, with reference to the reasoner BUNDLE that implements the same tableau algorithm as TRILL. The proof proceeds by induction on the number of rule applications following the proof of Theorem 2 of [11].

6 TRILL

We use the Thea2 library [27] that converts OWL DL ontologies to Prolog by exploiting a direct translation of the OWL axioms into Prolog facts. For example, a simple subclass axiom between two named classes $Cat \sqsubseteq Pet$ is written using the `subClassOf/2` predicate as `subClassOf('Cat','Pet')`. For more complex axioms Thea2 exploits the list construct of Prolog, so the axiom $NatureLover \equiv PetOwner \sqcup GardenOwner$ becomes `equivalentClasses(['NatureLover', unionOf(['PetOwner', 'GardenOwner']))`.

In order to represent the tableau, we use a couple $Tableau = (A, T)$, where A is a list containing all the class assertions of the individuals with the corresponding value of τ and the information about nominal individuals, while T is a triple (G,RBN,RBR) in which G is a directed graph that contains the structure of the tableau, RBN is a red-black tree in which each key is a couple of individuals and the value associated to it is the set of the labels of the edge between the two individuals, and RBR is a red-black tree in which each key is a role and the value associated to it is the set of couples of individuals that are linked by the role. This representation allows us to rapidly find the information

needed during the execution of the tableau algorithm. For managing the *blocking* system we use a predicate for each blocking state, so we have the following predicates: `nominal/2`, `blockable/2`, `blocked/2`, `indirectly_blocked/2` and `safe/3`. Each predicate takes as arguments the individual Ind and the tableau, (A, T). `safe/3` takes as input also the role R. For each nominal individual Ind at the time of the creation of the ABox we add the atom $nominal(Ind)$ to A, then every time we have to check the blocking status of an individual we call the corresponding predicate that returns the status by checking the tableau.

In TRILL deterministic and non-deterministic tableau expansion rules are treated differently, see Figure 1 for the list of rules. Deterministic rules are implemented by a predicate $rule\_name(Tab, Tab1)$ that, given the current tableau Tab, returns the tableau $Tab1$ to which the rule was applied. Figure 2 shows the code of the deterministic rule $\rightarrow$ *unfold*. The predicate `unfold_rule/2` searches in Tab for an individual to which the rule can be applied and calls the predicate `find_sub_sup_class/3` in order to find the class to be added to the label of the individual. `find/2` implements the search for a class assertion. Since the data structure that stores class assertions is currently a list, `find/2` simply calls `member/2`. `absent/3` checks if the class assertion axiom with the associated explanation is already present in A. Non-deterministic rules are implemented by a

```
unfold_rule((A,T),([(classAssertion(D,Ind),[(Ax,Ind)|Expl])|A],T)):-
  find((classAssertion(C,Ind),Expl),A),
  atomic(C),
  find_sub_sup_class(C,D,Ax),
  absent(classAssertion(D,Ind),[(Ax,Ind)|Expl],(A,T)).

find_sub_sup_class(C,D,subClassOf(C,D)):-
  subClassOf(C,D).

find_sub_sup_class(C,D,equivalentClasses(L)):-
  equivalentClasses(L),
  member(C,L),
  member(D,L),
  C\==D.
```

Fig. 2. Code of $\rightarrow$ *unfold* rules.

predicate $rule\_name(Tab, TabList)$ that, given the current tableau Tab, returns the list of tableaux $TabList$ obtained by applying the rule. Figure 3 shows the code of the non-deterministic rule $\rightarrow \sqcup$. The predicate `or_rule/2` searches in Tab for an individual to which the rule can be applied and unifies $TabList$ with the list of new tableaux created by `scan_or_list/6`.

Expansion rules are applied in order by `apply_all_rules/2`, first the non-deterministic ones and then the deterministic ones. The predicate `apply_nondet_rules(RuleList,Tab,Tab1)` takes as input the list of

```
or_rule((A,T),L):-
  find((classAssertion(unionOf(LC),Ind),Expl),A),
  \+ indirectly_blocked(Ind,T0),
  findall((A1,T),scan_or_list(LC,Ind,Expl,A,T,A1),L),
  L\=[],!.

scan_or_list([],_Ind,_Expl,A,T,A).

scan_or_list([C|_T],Ind,Expl,A,T,[(classAssertion(C,Ind),Expl)|A]):-
  absent(classAssertion(C,Ind),Expl,(A,T)).

scan_or_list([_C|T],Ind,Expl,A0,T,A):-
  scan_or_list(T,Ind,Expl,A0,T,A).
```

Fig. 3. Code of $\rightarrow \sqcup$ rule.

non-deterministic rules and the current tableau and returns a tableau obtained by the application of one rule. `apply_nondet_rules/3` is called as `apply_nondet_rules([or_rule,max_rule],Tab,Tab1)` and is shown in Fig. 4. If a non-deterministic rule is applicable, the list of tableaux obtained by its

```
apply_all_rules(Tab,Tab2):-
  apply_nondet_rules([or_rule,max_rule],Tab,Tab1),
  (Tab=Tab1 -> Tab2=Tab1 ; apply_all_rules(Tab1,Tab2)).


apply_nondet_rules([],Tab,Tab1):-
  apply_det_rules([o_rule,and_rule,unfold_rule,add_exists_rule,
    forall_rule,forall_plus_rule,exists_rule,min_rule],Tab,Tab1).

apply_nondet_rules([H|T],Tab,Tab1):-
  C=..[H,Tab,L],
  call(C),!,
  member(Tab1,L),
  Tab \= Tab1.

apply_nondet_rules([_|T],Tab,Tab1):-
  apply_nondet_rules(T,Tab,Tab1).
```

Fig. 4. Code of the predicates `apply_all_rules/2` and `apply_nondet_rules/3`.

application is returned by the rule predicate, a cut is performed to avoid backtracking to other rule choices and a tableau from the list is non-deterministically chosen with the `member/2` predicate. If no non-deterministic rule is applicable, deterministic rules are tried sequentially with the predicate `apply_det_rules/3`,

shown in Figure 5, that is called as `apply_det_rules(RuleList, Tab,Tab1)`. It takes as input the list of deterministic rules and the current tableau and returns a tableau obtained by the application of one rule. After the application of a deterministic rule, a cut avoids backtracking to other possible choices for the deterministic rules. If no rule is applicable, the input tableau is returned and rule application stops, otherwise a new round of rule application is performed. In each rule application round, a rule is applied if its result is not already present

```
apply_det_rules([],Tab,Tab).

apply_det_rules([H|T],Tab,Tab1):-
  C=..[H,Tab,Tab1],
  call(C),!.

apply_det_rules([_|T],Tab,Tab1):-
  apply_det_rules(T,Tab,Tab1).
```

Fig. 5. Code of the predicates `apply_det_rules/3`.

in the tableau. This avoids both infinite loops in rule application and considering alternative rules when a rule is applicable. In fact, if a rule is applicable in a tableau, it will also be so in any tableaux obtained by its expansion, thus the choice of which expansion rule to apply introduces "don't care" non-determinism. Differently, non-deterministic rules introduce in the algorithm also "don't know" non-determinism, since a single tableau is expanded into a set of tableaux. We use Prolog search only to handle "don't know" non-determinism.

Example 4. Let us consider the knowledge base presented in Example 1 and the query $Q = kevin : NatureLover$. After the initialization of the tableau, TRILL can apply the $\rightarrow$ *unfold* rule to the individuals *tom* or *fluffy*. Suppose it selects *tom*. The tracing function τ becomes (in predicate logic):

$\tau(Pet, tom) = \{\ Cat(tom),\ Cat(tom) \rightarrow Pet(tom)\}$

At this point TRILL applies the $\rightarrow$ CE rule to *kevin*, adding $\neg(\exists hasAnimal.Pet) \sqcup NatureLover = \forall hasAnimal.(\neg Pet) \sqcup NatureLover$ to $\mathcal{L}(kevin)$ with the following tracing function:

$\tau(\forall hasAnimal.(\neg Pet) \sqcup NatureLover, kevin) = \{$
$\exists y.hasAnimal(kevin, y) \wedge Pet(y) \rightarrow NatureLover(kevin)\}$

Then it applies the $\rightarrow \sqcup$ rule to *kevin* generating two tableaux. In this step we have a backtracking point because we have to choose which tableau to expand. In the first one TRILL adds $\forall hasAnimal.(\neg Pet)$ to the label of *kevin* with the tracing function

$\tau(\forall hasAnimal.(\neg Pet), kevin) = \{$
$\exists y.hasAnimal(kevin, y) \wedge Pet(y) \rightarrow NatureLover(kevin)\}$

Now it can apply the $\rightarrow \forall$ rule to *kevin*. In this step it can use either *tom* or *fluffy*, supposing it selects *tom* the tracing function will be:

$\tau(\neg(Pet), tom) = \{\ hasAnimal(kevin, tom),$
$hasAnimal(kevin, tom) \wedge Pet(tom) \rightarrow NatureLover(kevin)\}$

At this point this first tableau contains a clash for the individual *tom*, thus TRILL backtracks and expands the second tableau. The second tableau was created by applying the $\rightarrow CE$ rule that added $NatureLover$ to the label of *kevin*, so the second tableau contains a clash, too. Now TRILL joins the tracing functions of the two clashes to find the following InstMinA:

$\{hasAnimal(kevin, tom) \wedge Pet(tom) \rightarrow NatureLover(kevin),$
$hasAnimal(kevin, tom),\ Cat(tom),\ Cat(tom) \rightarrow Pet(tom)\}.$

The tableau algorithm returns a single InstMinA. The computation of All-InstMinAs$(Q, \mathcal{K})$ is performed by simply calling `findall/3` over the tableau predicate.

Example 5. Let us consider Example 4. Once the first InstMinA is found, TRILL performs backtracking. Supposing it applies the $\rightarrow$ *unfold* rule to the individual *fluffy* instead of *tom* and following the same steps used in Example 4 it finds a new InstMinA:

$\{hasAnimal(kevin, fluffy) \wedge Pet(fluffy) \rightarrow NatureLover(kevin),$
$hasAnimal(kevin, fluffy),\ Cat(fluffy),\ Cat(fluffy) \rightarrow Pet(fluffy)\}.$

7 Experiments

In this section, we evaluate TRILL performances when computing instantiated explanations by comparing it to BUNDLE that also solves the inst-min-a-enum problem. We consider four different knowledge bases of various complexity: the BRCA[3] that models the risk factor of breast cancer, an extract of the DBPedia[4] ontology that has been obtained from Wikipedia, the Biopax level 3[5] that models metabolic pathways and the Vicodi[6] that contains information on European history. For the tests, we used the DBPedia and the Biopax KBs without ABox while for BRCA and Vicodi we used a little ABox contaning 1 individual for the first one and 19 individuals for the second one. We ran two different subclass-of queries w.r.t. the DBPedia and the Biopax datasets and two different instance-of queries w.r.t. the other KBs. For each KB, we ran each query 50 times for a total of 100 executions of the reasoners. Table 2 shows, for each ontology, the number of axioms, the average number of explanations and the average time in milliseconds that TRILL and BUNDLE took for answering the queries. In particular, in order to stress the algorithm, the BRCA and the version of DBPedia that we used contain a large number of subclass axioms between complex concepts.These preliminary tests show that TRILL performance can sometimes be better than BUNDLE, even if it lacks all the optimizations that BUNDLE inherits from Pellet. This represents evidence that a Prolog implementation of a Semantic Web tableau reasoner is feasible and that may lead to a practical system.

[3] `http://www2.cs.man.ac.uk/~klinovp/pronto/brc/cancer_cc.owl`
[4] `http://dbpedia.org/`
[5] `http://www.biopax.org/`
[6] `http://www.vicodi.org/`

Dataset	n. axioms	av. n. expl	TRILL time (ms)	BUNDLE time (ms)
BRCA	322	6.5	95,691	10,210
DBPedia	535	16.0	80,804	28,040
Biopax level 3	826	2.0	24	1,451
Vicodi	220	1.0	136	1,004

Table 2. Results of the experiments in terms of average times for inference.

8 Conclusions

In this paper we presented the algorithm TRILL for reasoning on $\mathcal{SHOIN}(\mathbf{D})$ knowledge bases and its Prolog implementation. The results we obtained show that Prolog is a viable language for implementing DL reasoning algorithms and that performances are comparable with those of a state of the art reasoner.

In the future we plan to apply various optimizations to TRILL in order to better manage the expansion of the tableau. In particular, we plan to carefully choose the rule and node application order. Moreover, we plan to exploit TRILL for performing reasoning on probabilistic ontologies and on integration of probabilistic logic programming with DLs and for implementing learning algorithms for such integration, along the lines of [4, 5, 22].

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
2. Baader, F., Horrocks, I., Sattler, U.: Description logics. In: Handbook of knowledge representation, chap. 3, pp. 135–179. Elsevier (2008)
3. Beckert, B., Posegga, J.: leantap: Lean tableau-based deduction. J. Autom. Reasoning 15(3), 339–358 (1995)
4. Bellodi, E., Riguzzi, F.: Learning the structure of probabilistic logic programs. In: Muggleton, S.H., Tamaddoni-Nezhad, A., Lisi, F.A. (eds.) ILP 2011. LNCS, vol. 7207, pp. 61–75. Springer (2012)
5. Bellodi, E., Riguzzi, F.: Expectation Maximization over binary decision diagrams for probabilistic logic programs. Intel. Data Anal. 17(2), 343–363 (2013)
6. Faizi, I.: A Description Logic Prover in Prolog, Bachelor's thesis, Informatics Mathematical Modelling, Technical University of Denmark (2011)
7. Halaschek-Wiener, C., Kalyanpur, A., Parsia, B.: Extending tableau tracing for ABox updates. Tech. rep., University of Maryland (2006)
8. Herchenröder, T.: Lightweight Semantic Web Oriented Reasoning in Prolog: Tableaux Inference for Description Logics. Master's thesis, School of Informatics, University of Edinburgh (2006)
9. Hitzler, P., Krötzsch, M., Rudolph, S.: Foundations of Semantic Web Technologies. CRCPress (2009)
10. Hustadt, U., Motik, B., Sattler, U.: Deciding expressive description logics in the framework of resolution. Inf. Comput. 206(5), 579–601 (2008)

11. Kalyanpur, A.: Debugging and Repair of OWL Ontologies. Ph.D. thesis, The Graduate School of the University of Maryland (2006)
12. Kalyanpur, A., Parsia, B., Horridge, M., Sirin, E.: Finding all justifications of OWL DL entailments. In: Aberer, K., et al. (eds.) ISWC/ASWC 2007. LNCS, vol. 4825, pp. 267–280. Springer (2007)
13. Kalyanpur, A., Parsia, B., Sirin, E., Hendler, J.A.: Debugging unsatisfiable classes in OWL ontologies. J. Web Sem. 3(4), 268–293 (2005)
14. Lukácsy, G., Szeredi, P.: Efficient description logic reasoning in prolog: The dlog system. TPLP 9(3), 343–414 (2009)
15. Meissner, A.: An automated deduction system for description logic with alcn language. Studia z Automatyki i Informatyki 28-29, 91–110 (2004)
16. Patel-Schneider, P, F., Horrocks, I., Bechhofer, S.: Tutorial on OWL (2003)
17. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. 32(1), 57–95 (1987)
18. Riguzzi, F., Bellodi, E., Lamma, E., , Zese, R.: Probabilistic description logics under the distribution semantics. Tech. Rep. ML-01, University of Ferrara (2013), `http://sites.unife.it/ml/bundle`
19. Riguzzi, F., Bellodi, E., Lamma, E.: Probabilistic Datalog+/- under the distribution semantics. In: Kazakov, Y., Lembo, D., Wolter, F. (eds.) DL 2012. CEUR Workshop Proceedings, vol. 846. Sun SITE Central Europe (2012)
20. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R.: Epistemic and statistical probabilistic ontologies. In: Bobillo, F., et al. (eds.) URSW 2012. CEUR Workshop Proceedings, vol. 900, pp. 3–14. Sun SITE Central Europe (2012)
21. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R.: BUNDLE: A reasoner for probabilistic ontologies. In: Faber, W., Lembo, D. (eds.) RR 2013. LNCS, vol. 7994, pp. 183–197. Springer (2013), `http://www.ing.unife.it/docenti/FabrizioRiguzzi/Papers/RigBelLam-RR13b.pdf`
22. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R.: Parameter learning for probabilistic ontologies. In: Faber, W., Lembo, D. (eds.) RR 2013. LNCS, vol. 7994, pp. 265–270. Springer (2013), `http://www.ing.unife.it/docenti/FabrizioRiguzzi/Papers/RigBelLam-RR13a.pdf`
23. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: ICLP 1995. pp. 715–729. MIT Press (1995)
24. Sattler, U., Calvanese, D., Molitor, R.: Relationships with other formalisms. In: Description Logic Handbook, chap. 4, pp. 137–177. Cambridge University Press (2003)
25. Schlobach, S., Cornet, R.: Non-standard reasoning services for the debugging of description logic terminologies. In: Gottlob, G., Walsh, T. (eds.) IJCAI 2003. pp. 355–362. Morgan Kaufmann (2003)
26. Sirin, E., Parsia, B., Cuenca-Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. Web Sem. 5(2), 51–53 (2007)
27. Vassiliadis, V., Wielemaker, J., Mungall, C.: Processing owl2 ontologies using thea: An application of logic programming. In: International Workshop on OWL: Experiences and Directions. CEUR Workshop Proceedings, vol. 529. CEUR-WS.org (2009)
28. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming 12(1-2), 67–96 (2012)