Logic Programming Techniques for Reasoning with Probabilistic Ontologies

Riccardo Zese, Elena Bellodi, Evelina Lamma
Dipartimento di Ingegneria
University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[riccardo.zese,elena.bellodi,evelina.lamma]@unife.it

Fabrizio Riguzzi
Dipartimento di Matematica e Informatica
University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
fabrizio.riguzzi@unife.it

Abstract

The increasing popularity of the Semantic Web drove to a widespread adoption of Description Logics (DLs) for modeling real world domains. To help the diffusion of DLs, a large number of reasoning algorithms have been developed. Usually these algorithms are implemented in procedural languages such as Java or C++. Most of the reasoners exploit the tableau algorithm which has to manage non-determinism, a feature that is not easy to handle using such languages. Reasoning on real world domains also requires the capability of managing probabilistic and uncertain information. We thus present TRILL, for "Tableau Reasoner for descrIption Logics in proLog", that implements a tableau algorithm and is able to return explanations for queries and their corresponding probability, and TRILL^P, for "TRILL powered by Pinpointing formulas", which is able to compute a Boolean formula representing the set of explanations for a query. This approach can speed up the process of computing the probability. Prolog non-determinism allows us to easily handle the tableau's non-deterministic expansion rules.

Introduction

The Semantic Web aims at making information regarding real world domains available in a form that is understandable by machines (Hitzler, Krötzsch, and Rudolph 2009). The World Wide Web Consortium is working for realizing this vision by supporting 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, in some cases, 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 (Hitzler, Krötzsch, and Rudolph 2009). Moreover, uncertain information is intrinsic to real world domains, thus the combination of probability and logic theories becomes of foremost importance.

In order to fully support the development of the Semantic Web, efficient DL reasoners, such us Pellet, RacerPro, FaCT++ and HermiT, are able to extract implicit information from the modeled ontologies. Despite the large number of available reasoners, only few of them are able to manage probabilistic information as well. One of the most common approaches for reasoning is the tableau algorithm that exploits some non-deterministic expansion rules. This requires the developers to implement a search strategy in an or-branching search space. Moreover, if we want to compute the probability of a query, the algorithm has to compute all the explanations for the query, thus it has to explore all the non-deterministic choices taken during the execution.

In this paper, we present the systems TRILL for "Tableau Reasoner for descrIption Logics in proLog" and TRILL^P for "TRILL powered by Pinpointing formulas". They are tableau reasoners for the $\mathcal{SHOIN}$ DL and for the $\mathcal{ALC}$ DL respectively, both implemented in Prolog. Prolog search strategy is exploited for taking into account the non-determinism of the tableau rules. They use the Thea2 library (Vassiliadis, Wielemaker, and Mungall 2009) 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 and TRILL^P can check the consistency of a concept and the entailment of an axiom from an ontology, and can compute the probability of the entailment following DISPONTE (Riguzzi et al. 2012), a semantics for probabilistic ontologies. The availability of a Prolog implementation of a DL reasoner will also facilitate the development of probabilistic reasoners that can integrate probabilistic logic programming with probabilistic DLs. In probabilistic logic programming one of the most widespread approaches is the Distribution Semantics (Sato 1995), on which DISPONTE is based. Since our systems follow DISPONTE, they are easily extensible to take into account this integration.

Description Logics

DLs are knowledge representation formalisms that are at the basis of the Semantic Web (Baader et al. 2003; Baader, Horrocks, and Sattler 2008) and are used for modeling ontologies. They possess nice computational properties such as decidability and/or low complexity.

Usually, DLs' syntax is based on concepts and roles which correspond respectively to sets of individuals and sets of pairs of individuals of the domain. We first briefly describe $\mathcal{ALC}$ and then $\mathcal{SHOIN}(\mathbf{D})$.

Let $\mathbf{C}$, $\mathbf{R}$ and $\mathbf{I}$ be sets of *atomic concepts*, *atomic roles* and *individuals*, respectively. *Concepts* are defined by induction as follows. Each $C \in \mathbf{C}$ is a concept, $\bot$ and $\top$ are concepts. If C, C_1 and C_2 are concepts and $R \in \mathbf{R}$, then $(C_1 \sqcap C_2)$, $(C_1 \sqcup C_2)$ and $\neg C$ are concepts, as well as $\exists R.C$

and $\forall R.C$. 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 $C \in \mathbf{C}$, $R \in \mathbf{R}$ and $a, b \in \mathbf{I}$. A *knowledge base* (KB) $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ consists of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$ and is usually assigned a semantics in terms of interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, 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 mapping $\cdot^{\mathcal{I}}$ is extended to all concepts (where $R^{\mathcal{I}}(x) = \{y|(x, y) \in R^{\mathcal{I}}\}$) as:

$$
\begin{array}{rcl}
\top^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \\
\bot^{\mathcal{I}} & = & \emptyset \\
(C_1 \sqcap C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\
(C_1 \sqcup C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(\forall R.C)^{\mathcal{I}} & = & \{x \in \Delta^{\mathcal{I}} | R^{\mathcal{I}}(x) \subseteq C^{\mathcal{I}}\} \\
(\exists R.C)^{\mathcal{I}} & = & \{x \in \Delta^{\mathcal{I}} | R^{\mathcal{I}}(x) \cap C^{\mathcal{I}} \neq \emptyset\}
\end{array}
$$

In the following we describe $\mathcal{SHOIN}(\mathbf{D})$ showing what it adds to $\mathcal{ALC}$. 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 $R, S \in \mathbf{R} \cup \mathbf{R}^-$.

If $a \in \mathbf{I}$, then $\{a\}$ is a concept called *nominal*, and if C, C_1 and C_2 are concepts and $R \in \mathbf{R} \cup \mathbf{R}^-$, then $\geq nR$ and $\leq nR$ for an integer $n \geq 0$ are also concepts. A $\mathcal{SHOIN}(\mathbf{D})$ KB $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$ consists of a TBox $\mathcal{T}$, an RBox $\mathcal{R}$ and an ABox $\mathcal{A}$.

The mapping $\cdot^{\mathcal{I}}$ is extended to all new concepts (where $\#X$ denotes the cardinality of the set X) as:

$$
\begin{array}{rcl}
(R^-)^{\mathcal{I}} & = & \{(y, x) | (x, y) \in R^{\mathcal{I}}\} \\
\{a\}^{\mathcal{I}} & = & \{a^{\mathcal{I}}\} \\
(\geq nR)^{\mathcal{I}} & = & \{x \in \Delta^{\mathcal{I}} | \#R^{\mathcal{I}}(x) \geq n\} \\
(\leq nR)^{\mathcal{I}} & = & \{x \in \Delta^{\mathcal{I}} | \#R^{\mathcal{I}}(x) \leq n\}
\end{array}
$$

$\mathcal{SHOIN}(\mathbf{D})$ allows for the definition of 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.

$\mathcal{SHOIN}(\mathbf{D})$ is decidable iff there are no number restrictions on roles which are transitive or have transitive subroles.

A query Q over a KB $\mathcal{K}$ is usually an axiom for which we want to test the entailment from the KB, 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. For example, the entailment of the axiom $C \sqsubseteq D$ may be tested by checking the unsatisfiability of the concept $C \sqcap \neg D$ while the entailment of the axiom $a : C$ may be tested by checking the unsatisfiability of $a : \neg C$.

Example 1 *The following KB is inspired by the ontology* `people+pets` *(Patel-Schneider, Horrocks, and Bechhofer 2003):*

$$
\begin{array}{l}
\exists hasAnimal.Pet \sqsubseteq NatureLover \\
fluffy : Cat \\
tom : Cat \\
Cat \sqsubseteq Pet \\
(kevin, fluffy) : hasAnimal \\
(kevin, tom) : hasAnimal
\end{array}
$$

It states that individuals that own an animal which is a pet are nature lovers and that $kevin$ owns the animals $fluffy$ and tom, which are cats. Moreover, cats are pets. The KB entails the query $Q = kevin : NatureLover$.

Querying KBs: The Tableau Algorithm

In order to answer queries to DL KBs, a *tableau algorithm* can be used. A *tableau* is an ABox represented as a graph G where each node corresponds to an individual a and is labeled with the set of concepts $\mathcal{L}(a)$ to which a belongs. 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, starting from a tableau that contains the negation of the axiom. For example, the axiom $C \sqsubseteq D$ can be proved by showing that $C \sqcap \neg D$ is empty, while, if the query is a class assertion, $C(a)$, we add $\neg C$ to the label of a. For testing the inconsistency of a concept C we have to test the emptiness of C by adding a new anonymous node a to the tableau whose label contains C. Then, the *tableau algorithm* repeatedly applies a set of consistency preserving *tableau expansion rules* until a clash (i.e., a contradiction, for example, a concept C and a node a where C and $\neg C$ are present in the label of a) is detected or a clash-free graph is found to which no more rules are applicable. If no clashes are found, the tableau represents a model for the negation of the query, which is thus not entailed.

Each expansion rule updates as well a *tracing function* τ, which associates labels of nodes and edges with a subset of the axioms of the KB. 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 tableau expansion rules for $\mathcal{SHOIN}(\mathbf{D})$ are shown in Figure 1, where the rules for the $\mathcal{ALC}$ DL are marked by $(*)$.

For ensuring the termination of the algorithm, a special condition known as *blocking* (Kalyanpur 2006) is used. In a tableau a node x can be a *nominal* node if its label $\mathcal{L}(x)$ contains a *nominal* or a *blockable* node. 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.

Finding Explanations

The problem of finding explanations for a query has been investigated by various authors (Schlobach and Cornet 2003; Kalyanpur 2006; Halaschek-Wiener, Kalyanpur, and Parsia 2006; Kalyanpur et al. 2007). It was called *axiom pinpointing* in (Schlobach and Cornet 2003) and considered as a non-standard reasoning service useful for tracing derivations and debugging ontologies. In particular, *minimal axiom sets* or *MinAs* for short, also called *explanations*, are introduced in (Schlobach and Cornet 2003).

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 (Schlobach and Cornet 2003). ALL-MINAS$(Q, \mathcal{K})$ is the set of all MinAs for query Q in the knowledge base $\mathcal{K}$.

The tableau algorithm returns a single MinA using the tracing function. To solve MIN-A-ENUM, reasoners written in imperative languages, like Pellet (Sirin et al. 2007), have to implement a search strategy in order to explore the entire search space of the possible explanations. In particular, Pellet, that is written entirely in Java, uses Reiter's *hitting set algorithm* (Reiter 1987). The algorithm, described in detail in (Kalyanpur 2006), starts from a MinA S and initializes a labeled tree called *Hitting Set Tree* (HST) with S as the label of its root v. Then it selects an arbitrary axiom E in S, it removes it from $\mathcal{K}$, generating a new knowledge base $\mathcal{K}' = \mathcal{K} - \{E\}$, and tests the unsatisfiability of C w.r.t. $\mathcal{K}'$. If C is still unsatisfiable, we obtain a new explanation. The algorithm adds a new node w and a new edge $\langle v, w\rangle$ to the tree, then it assigns this new explanation to the label of w and the axiom E to the label of the edge. The algorithm repeats this process until the unsatisfiability test returns negative: in that case the algorithm labels the new node with OK, makes it a leaf, backtracks to a previous node, selects a different axiom to be removed from the KB and repeats these operations until the HST is fully built. The algorithm also eliminates extraneous unsatisfiability tests based on previous results: once a path leading to a node labeled OK is found, any superset of that path is guaranteed to be a path leading to a node where C is satisfiable, and thus no additional unsatisfiability test is needed for that path, as indicated by an X in the node label. When the HST is fully built, all leaves of the tree are labeled with OK or X. The distinct non leaf nodes of the tree collectively represent the set ALL-MINAS$(C, \mathcal{K})$.

In (Baader and Peñaloza 2010a; 2010b) the authors consider the problem of finding a *pinpointing formula* instead of ALL-MINAS$(Q, \mathcal{K})$. The pinpointing formula is a monotone Boolean formula in which each Boolean variable corresponds to an axiom of the KB. This formula is built using the variables and the conjunction and disjunction connectives. It compactly encodes the set of all MinAs. Let assume that each axiom E of a KB $\mathcal{K}$ is associated with a propositional variable, indicated with $var(E)$. The set of all propositional variables is indicated with $var(\mathcal{K})$. A valuation ν of a monotone Boolean formula is the set of propositional variables that are true. For a valuation $\nu \subseteq var(\mathcal{K})$, let $\mathcal{K}_\nu := \{t \in \mathcal{K} | var(t) \in \nu\}$.

Definition 2 (Pinpointing formula) *Given a query Q and a KB $\mathcal{K}$, a monotone Boolean formula ϕ over $var(\mathcal{K})$ is called a* pinpointing formula *for Q if for every valuation $\nu \subseteq var(\mathcal{K})$ it holds that $\mathcal{K}_\nu \models Q$ iff ν satisfies ϕ.*

In Lemma 2.4 of (Baader and Peñaloza 2010b) the authors proved that the set of all MinAs can be obtained by transforming the pinpoiting formula into DNF and removing disjuncts implying other disjuncts. The example below illustrates axiom pinpointing and the pinpointing formula.

Example 2 (Pinpointing formula) *Consider the KB of Example 1. We associate Boolean variables to axioms as follows:*

$F_1 = \exists hasAnimal.Pet \sqsubseteq NatureLover$
$F_2 = (kevin, fluffy) : hasAnimal$
$F_3 = (kevin, tom) : hasAnimal$
$F_4 = fluffy : Cat$
$F_5 = tom : Cat$
$F_6 = Cat \sqsubseteq Pet.$

Let $Q = kevin : NatureLover$ be the query, then ALL-MINAS$(Q, \mathcal{K}) = \{\{F_2, F_4, F_6, F_1\}, \{F_3, F_5, F_6, F_1\}\}$, *while the pinpointing formula is* $((F_2 \wedge F_4) \vee (F_3 \wedge F_5)) \wedge F_6 \wedge F_1$.

In the following, we briefly define how a tableau algorithm can be modified to find the pinpointing formula. For more details and formal definitions see (Baader and Peñaloza 2010b).

Given a KB $\mathcal{K}$, the modified algorithm associates a label $lab(a)$ that is a monotone Boolean formula over $var(\mathcal{K})$ to every assertion a. For deciding whether a rule is applicable we have to control the insertability of the new assertion. Let A be a set of labeled assertions and ψ a monotone Boolean formula, the assertion a is $\psi{-}insertable$ into A if either $a \notin A$, or $a \in A$ but $\psi \not\models lab(a)$. Given a set B of assertions and a set A of labeled assertions, the set of $\psi{-}insertable$ elements of B into A is defined as $ins_\psi(B, A) := \{b \in B | b \text{ is } \psi{-}insertable \text{ into } A\}$. For deciding the applicability of a rule we need also to give the definition of *substitution*. A substitution is a mapping $\rho : V \to D$, where V is a finite set of variables and D is a countably infinite set of constants that contains all the individuals in the KB and all the fresh individuals created by the application of the rules. Variables are seen as placeholder for individuals in the assertions. For example, an assertion can be $C(x)$ or $R(x, y)$ where C is a concept, R is a role and x and y are variables. In this case, let $C(x)$ be an assertion with the variable x and $\rho : x \to a$ a substitution, then $C(x)\rho$ denotes the assertion obtained by replacing the variable with its $\rho{-}image$, i.e. $C(a)$. A rule is of the form $(B_0, S) \to \{B_1, ..., B_m\}$ where B_i are finite set of assertions and S is a finite set of axioms. A rule is applicable with a substitution ρ on the variable occurring in B_0 if $S \subseteq \mathcal{K}$, $B_0\rho \subseteq A$, where A is the set of assertions

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**
$Add(D, \mathcal{L}(a))$
$\tau(D, a) := (\tau(A, a) \cup \{A \sqsubseteq D\})$

$\rightarrow$ *CE* $(*)$: **if** $(C \sqsubseteq D) \in K$, with C not atomic, a not blocked, **then**
if $(\neg C \sqcup D) \notin \mathcal{L}(a)$, **then**
$Add((\neg C \sqcup D), a)$
$\tau((\neg C \sqcup D), a) := \{C \sqsubseteq D\}$

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

$\rightarrow \exists$ $(*)$: **if** $\exists S.C \in \mathcal{L}(a)$, a is not blocked, **then**
if a has no S-neighbor b with $C \in \mathcal{L}(b)$, **then**
create new node b, $Add(S, \langle a, b\rangle)$, $Add(C, b)$
$\tau(C, b) := \tau((\exists S.C), a)$
$\tau(S, \langle a, b\rangle) := \tau((\exists S.C), a)$

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

$\rightarrow \forall^+$: **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**
$Add(\forall R.C, b)$
$\tau((\forall R.C), b) := \tau((\forall S.C), a) \cup \tau(R, \langle a, b\rangle) \cup \{Trans(R)\} \cup \{R \sqsubseteq S\}$

$\rightarrow\geq$: **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 $b_1, ..., b_n$; $Add(S, \langle a, b_i\rangle)$; $\neq(b_i, b_j)$
$\tau(S, \langle a, b_i\rangle) := \tau((\geq nS), a)$
$\tau(\neq(b_i, b_j)) := \tau((\geq nS), a)$

$\rightarrow O$: **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 $\mathcal{L}(a)$, $\tau(Add(C_i, \mathcal{L}(b))) := \tau(Add(C_i, \mathcal{L}(a))) \cup \tau(Merge(a, b))$
(similarly for roles merged, and correspondingly for concepts in $\mathcal{L}(b)$)

Non-deterministic rules:

$\rightarrow \sqcup$ $(*)$: **if** $(C_1 \sqcup C_2) \in \mathcal{L}(a)$, a is not indirectly blocked, **then**
if $\{C_1, C_2\} \cap \mathcal{L}(a) = \emptyset$, **then**
Generate graphs $G_i := G$ for each $i \in \{1, 2\}$
$Add(C_i, a)$ in G_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'
$\tau(Merge(b_i, b_j)) := \tau((\leq nS), a) \cup \tau(S, \langle a, b_1\rangle) ... \cup \tau(S, \langle a, b_m\rangle)$
if b_j is a nominal node, **then** $Merge(b_i, b_j)$ in G',
else if b_i is a nominal node or ancestor of b_j, **then** $Merge(b_j, b_i)$
else $Merge(b_i, b_j)$ in G'
if b_i is merged into b_j, **then** 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 b_j if merged into b_i)

Figure 1: TRILL tableau expansion rules; the subset of rules marked by $(*)$ is employed by TRILL^P.

contained in the ABox and found during inference, and, for every $1 \leq i \leq m$ and every substitution ρ' on the variables occurring in $B_0 \cup B_i$, we have $ins_\psi(B_i\rho', A) \neq \emptyset$, where $\psi := \bigvee_{b \in B_0} lab(b\rho) \wedge \bigvee_{s \in S} lab(s)$. Moreover, except for the unfold rule, the node N to which the rule is applicable is not (indirectly) blocked. When the tableau is fully built, the algorithm conjoins the labels of each clash for building the final pinpointing formula.

TRILL and TRILL^P

Both TRILL and TRILL^P implement a tableau algorithm, the first solves MIN-A-ENUM while the second computes the pinpointing formula representing the set of MinAs. They can answer concept and role membership queries, subsumption queries and can test the unsatifiability of a concept of the KB or the inconsistency of the entire KB. TRILL and TRILL^P are implemented in Prolog, so the management of the non-determinism of the rules is delegated to the language.

We use the Thea2 library (Vassiliadis, Wielemaker, and Mungall 2009) for converting OWL DL KBs into Prolog. Thea2 performs 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, TRILL and TRILL^P use a pair $Tableau = (A, T)$, where A is a list containing information about individuals and class assertions with the corresponding value of the tracing function. The tracing function stores a fragment of the knowledge base in TRILL and the pinpointing formula in TRILL^P. 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 (a key-value dictionary), where a key is a couple of individuals and its value is the set of the labels of the edge between the two individuals, and RBR is a red-black tree, where a key is a role and its value is the set of couples of individuals that are linked by the role. This representation allows to quickly find the information needed during the execution of the tableau algorithm. For managing the *blocking* system we use a predicate for each blocking state: `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 individual Ind in 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.

Deterministic and non-deterministic tableau expansion rules are treated differently. Non-deterministic rules are implemented by a predicate $rule\_name(Tab, TabList)$ that, given the current tableau Tab, returns the list of tableaux $TabList$ created by the application of the rule to Tab. Deterministic rules are implemented by a predicate $rule\_name(Tab, Tab1)$ that, given the current tableau Tab, returns the tableau $Tab1$ obtained by the application of the rule to Tab. 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/3` takes as input the list of non-deterministic rules and the current tableau and returns a tableau obtained by the application of one of the rules. It is called as `apply_nondet_rules(RuleList,Tab,Tab1)` and is shown in Figure 2.

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

apply_nondet_rules([],Tab,Tab1):-
  apply_det_rules([...],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).

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).
```

Figure 2: Definition of the non-deterministic expansion rules by means of the predicates `apply_all_rules/2`, `apply_nondet_rules/3` and `apply_det_rules/3`. The list `[...]` contains the available rules and is different in TRILL and TRILL^P.

If a non-deterministic rule is applicable, the list of tableaux obtained by its application is returned by the predicate corresponding to the applied rule, 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 2, 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 with the application of one of the rules. 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 Figure 1, the symbol $(*)$ denotes the rules shared by TRILL and TRILL^P. In these rules, the operator $\cup$ for τ

means union between two sets in TRILL, while in TRILL^P it joins two Boolean formulas with the `OR` Boolean operator. Moreover, when a concept is already present in a node label, TRILL checks whether to update the tracing function by verifying that the corresponding set of axioms is not a subset of τ, while TRILL^P performs a $\psi-insertability$ test.

In case the assertion a to be inserted is already associated with the corresponding individual, TRILL^P tests its $\psi-insertability$ by means of a satisfiability solver. In particular, it conjoins the negation of $lab(a)$ with the Boolean formula associated to the individual in the tableau, and tests the satisfiability of such formula. If the test returns true, the two Boolean formulas are combined with the `OR` Boolean operator.

Computing the Probability

The aim of our work is to implement algorithms which can compute the probability of queries to KBs following DISPONTE (Riguzzi et al. 2012). DISPONTE applies the distribution semantics (Sato 1995) of probabilistic logic programming to DLs. A program following this semantics defines a probability distribution over normal logic programs called *worlds*. Then the distribution is extended to a joint distribution over worlds and queries from which the probability of a query is obtained by marginalization.

In DISPONTE, a *probabilistic knowledge base* $\mathcal{K}$ contains a set of *probabilistic axioms* which take the form

$$p :: E \tag{1}$$

where p is a real number in $[0, 1]$ and E is a DL axiom. The probability p can be interpreted as an *epistemic probability*, i.e., as the degree of our belief in the truth of axiom E. For example, a probabilistic concept membership axiom $p :: a : C$ means that we have degree of belief p in $C(a)$. A probabilistic concept inclusion axiom of the form $p :: C \sqsubseteq D$ represents the fact that we believe in the truth of $C \sqsubseteq D$ with probability p.

The idea of DISPONTE is to associate independent Boolean random variables to the axioms. To obtain a *world* w we decide whether to include each axiom or not in w. A world therefore is a non probabilistic KB that can be assigned a semantics in the usual way. A query is entailed by a world if it is true in every model of the world.

Formally, an *atomic choice* is a pair (E_i, k) where E_i is the ith probabilistic axiom and $k \in \{0, 1\}$ indicates whether E_i is chosen to be included in a world (k = 1) or not (k = 0). If a set of atomic choices κ is consistent, i.e., $(E_i, k) \in \kappa, (E_i, m) \in \kappa$ implies $k = m$ (only one decision is taken for each axiom), it is called a *composite choice*. The probability of a composite choice κ is $P(\kappa) = \prod_{(E_i,1)\in\kappa} p_i \prod_{(E_i,0)\in\kappa}(1 - p_i)$, where p_i is the probability associated with axiom E_i. A *selection* σ is a composite choice which contains an atomic choice (E_i, k) for every axiom of the theory. A selection σ identifies a theory w_σ called a *world* in this way: $w_\sigma = \{E_i|(E_i, 1) \in \sigma\}$. The probability of a world w_σ is $P(w_\sigma) = P(\sigma) = \prod_{(E_i,1)\in\sigma} p_i \prod_{(E_i,0)\in\sigma}(1-p_i)$. $P(w_\sigma)$ is a probability distribution over worlds, i.e., $\sum_{w\in\mathcal{W}_\mathcal{K}} P(w) = 1$, where $\mathcal{W}_\mathcal{K}$ is the set of all worlds.

We can now assign probabilities to queries. Given a world w, the probability of a query Q is defined as $P(Q|w) = 1$ if $w \models Q$ and 0 otherwise. The probability of a query can be defined by marginalizing the joint probability of the query and the worlds.

$$\begin{aligned} P(Q) &= \sum_{w\in\mathcal{W}_\mathcal{K}} P(Q, w) && (2)\\ &= \sum_{w\in\mathcal{W}_\mathcal{K}} P(Q|w)P(w) && (3)\\ &= \sum_{w\in\mathcal{W}_\mathcal{K}:w\models Q} P(w) && (4)\end{aligned}$$

The following example illustrates inference under DISPONTE semantics.

Example 3 *Consider the following KB, a probabilistic version of that proposed in Example 1.*

$$\begin{array}{lll} (1)\ 0.5 & :: & \exists hasAnimal.Pet \sqsubseteq NatureLover \\ & & fluffy : Cat \\ & & tom : Cat \\ (2)\ 0.6 & :: & Cat \sqsubseteq Pet \\ & & (kevin, fluffy) : hasAnimal \\ & & (kevin, tom) : hasAnimal \end{array}$$

It indicates that the individuals that own an animal which is a pet are nature lovers with a 50% probability and cats are pets with a 60% probability. The KB has four possible worlds:

$$\{(1), (2)\},\ \{(1)\},\ \{(2)\},\ \{\}$$

and the query axiom $Q = kevin : NatureLover$ is true in the first of them, while in the remaining ones it is false. The probability of the query is $P(Q) = 0.5 \cdot 0.6 = 0.3$.

When a probabilistic KB is given as input, all the axioms are translated by means of the Thea2 library. Then, for each probabilistic axiom of the form $Prob :: Axiom$, a fact `p(Axiom,Prob)` is asserted in the Prolog KB.

To compute the probability of queries to KBs following the DISPONTE semantics, we can first perform MIN-A-ENUM. Then the explanations must be made mutually exclusive, so that the probabilities of individual explanations are computed and summed. This can be done by exploiting a splitting algorithm as shown in (Poole 2000). Alternatively, we can assign independent Boolean random variables to the axioms contained in the explanations and define the DNF Boolean formula f_K which models the set of explanations K. Thus $f_K(\mathbf{X}) = \bigvee_{\kappa\in K} \bigwedge_{(E_i,1)} X_i \bigwedge_{(E_i,0)} \overline{X_i}$ where $\mathbf{X} = \{X_i|(E_i, k) \in \kappa, \kappa \in K\}$ is the set of Boolean random variables.

TRILL^P, instead, computes directly a pinpointing formula which is a monotone Boolean formula that represents the set of all MinAs.

Irrespective of which representation of the explanations we choose, a DNF or a general pinpointing formula, we can apply knowledge compilation and transform it into a Binary Decision Diagram (BDD), from which we can compute the probability of the query with a dynamic programming algorithm that is linear in the size of the BDD.

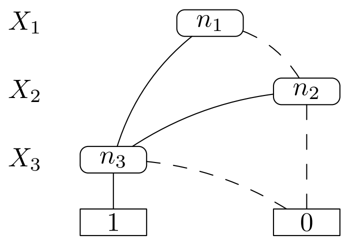

Figure 3: BDD representing the function $f(\mathbf{X}) = (X_1 \wedge X_3) \vee (X_2 \wedge X_3)$.

A BDD for a function of Boolean variables is a rooted graph that has one level for each Boolean variable. A node n in a BDD has two children: one corresponding to the 1 value of the variable associated with the level of n, indicated with $child_1(n)$, and one corresponding to the 0 value of the variable, indicated with $child_0(n)$. When drawing BDDs, the 0--branch - the one going to $child_0(n)$ - is distinguished from the 1-branch by drawing it with a dashed line. The leaves store either 0 or 1. Figure 3 shows a BDD for the function $f(\mathbf{X}) = (X_1 \wedge X_3) \vee (X_2 \wedge X_3)$, where the variables $\mathbf{X} = \{X_1, X_2, X_3\}$ are independent Boolean random variables.

A BDD performs a Shannon expansion of the Boolean formula $f(\mathbf{X})$, so that, if X is the variable associated with the root level of a BDD, the formula $f(\mathbf{X})$ can be represented as $f(\mathbf{X}) = X \wedge f^X(\mathbf{X}) \vee \overline{X} \wedge f^{\overline{X}}(\mathbf{X})$ where $f^X(\mathbf{X})$ ($f^{\overline{X}}(\mathbf{X})$) is the formula obtained by $f(\mathbf{X})$ by setting X to 1 (0). Now the two disjuncts are pairwise exclusive and the probability of $f(\mathbf{X})$ being true can be computed as $P(f(\mathbf{X})) = P(X)P(f^X(\mathbf{X})) + (1 - P(X))P(f^{\overline{X}}(\mathbf{X}))$ by knowing the probabilities of the Boolean variables of being true.

TRILL-on-SWISH

In order to popularize DISPONTE, we developed a Web application called "TRILL-on-SWISH" and available at `http://trill.lamping.unife.it`. We exploited SWISH (Lager and Wielemaker 2014), a recently proposed Web framework for logic programming that is based on various features and packages of SWI-Prolog. SWISH allows the user to write Prolog programs and ask queries in the browser without installing SWI-Prolog on his machine. We modified it in order to manage OWL KBs. SWISH also allows users to collaborate on code development. TRILL-on-SWISH allows users to write a KB in the RDF/XML format directly in the web page or load it from a URL, and specify queries that are answered by TRILL running on the server. Once the computation ends, the results are sent to the client browser and visualized in the Web page.

Experiments

In order to evaluate TRILL and TRILL^P performances, we compared them with BUNDLE, a reasoner for DISPONTE based on Pellet. We used four different knowledge bases of various complexity to which we added 50 probabilistic axioms:

- BRCA[1], which models the risk factor of breast cancer;
- an extract of the DBPedia[2] ontology obtained from Wikipedia;
- Biopax level 3[3], which models metabolic pathways;
- Vicodi[4], which contains information on European history.

For the tests, we used a version of the DBPedia and Biopax KBs without the ABox and a version of BRCA and of Vicodi with an ABox containing 1 individual and 19 individuals respectively. We added 50 probabilistic axioms to each KB. In this experimentation, the probabilistic parameter values were learned using EDGE (Riguzzi et al. 2013b), a system that computes the probability value associated with axioms starting from a set of positive and negative examples in the form of class assertion axioms that we regard as true (false), and for which we would like to get an high (low) probability respectively.

For each dataset, we randomly created 100 different queries. In particular, for the DBPedia and Biopax datasets, we created 100 subclass-of queries, while for the other KBs we created 80 subclass-of and 20 instance-of queries. For generating the subclass-of queries, we randomly selected two classes that are connected in the hierarchy of classes, so that each query had at least one explanation. For the instance-of queries, we randomly selected an individual a and a class to which a belongs by following the hierarchy of the classes, starting from the classes to which a explicitly belongs in the KB.

Table 1 shows, for each ontology, the average number of different MinAs computed and the average time in seconds that TRILL, TRILL^P and BUNDLE took for computing the probability of the queries. In particular, 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 both TRILL and TRILL^P performances can sometimes be better than BUNDLE, even if they lack 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 practical systems. Moreover, TRILL^P provides an improvement of the execution time with respect to TRILL when more MinAs are present.

Related Work

Usually, DL reasoners implement a tableau algorithm using a procedural language. Since some tableau expansion rules are non-deterministic, 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 (Sirin et al. 2007) is a tableau

[1] `http://www2.cs.man.ac.uk/~klinovp/pronto/brc/cancer_cc.owl`

[2] `http://dbpedia.org/`

[3] `http://www.biopax.org/`

[4] `http://www.vicodi.org/`

Table 1: Average number of MinAs and average time (in seconds) for computing the probability of queries with the reasoners TRILL, TRILL^P and BUNDLE.

Dataset	avg. n. MinAs	TRILL time (s)	TRILL^P time (s)	BUNDLE time (s)
BRCA	6.49	27.87	4.74	6.96
DBPedia	16.32	51.56	4.67	3.79
Biopax level 3	3.92	0.12	0.12	1.85
Vicodi	1.02	0.19	0.19	1.12

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 to find all the other MinAs. Recently, BUNDLE (Riguzzi et al. 2013a) was proposed for reasoning over DISPONTE KBs. BUNDLE exploits Pellet for solving MIN-A-ENUM and computes the probability of queries.

Reasoners written in Prolog can exploit its backtracking facilities for performing the search. This has been observed in various works. Beckert and Posegga (1995) proposed a tableau reasoner in Prolog for FOL based on free-variable semantic tableaux. However, the reasoner is not tailored to DLs. Meissner (2004) presented the implementation of a Prolog reasoner for the DL $\mathcal{ALCN}$. This work was the basis of (Herchenröder 2006), that considered $\mathcal{ALC}$ and improved (Meissner 2004) by implementing heuristic search techniques to reduce the running time. Faizi (2011) added to (Herchenröder 2006) the possibility of returning explanations for queries but still handled only $\mathcal{ALC}$.

Hustadt, Motik, and Sattler (2008) 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 to a disjunctive datalog program.

DLog (Lukácsy and Szeredi 2009) is an ABox reasoning algorithm for the $\mathcal{SHIQ}$ language that permits storing the content of the ABox externally in a database and answers instance check and instance retrieval queries by transforming the KB into a Prolog program. TRILL differs from these works for the considered DL and from DLog for the capability of answering general queries.

A different approach is shown in (Ricca et al. 2009), who introduced a system for reasoning on a logic-based ontology representation language, called OntoDLP, which is an extension of (disjunctive) ASP and can interoperate with OWL. This system, called OntoDLV, rewrites the OWL KB into the OntoDLP language, can retrieve information directly from external OWL ontologies and answers queries by using ASP. OntoDLV cannot find the set of explanations thus it is not applicable to DISPONTE KBs. All the presented systems are not able to compute the probability of queries.

Bruynooghe et al. (2010) presented FOProbLog, an extension of ProbLog where a program contains a set of *probabilistic facts*, i.e. facts annotated with probabilities, and a set of general clauses which can have positive and negative probabilistic facts in their body. Each fact is assumed to be probabilistically independent. FOProbLog follows the distribution semantics and exploits BDDs to compute the probability of queries. FOProblog is a reasoner for FOL that is not tailored to DLs, so the algorithm could be suboptimal for them.

Calì et al. (2009) combine DLs and logic programs in order to integrate ontologies and rules. They use a tightly coupled approach to (probabilistic) disjunctive description logic programs. They define a description logic program as a pair (L, P), where L is a DL KB and P is a disjunctive logic program which contains rules on concepts and roles of L. P may contain probabilistic alternatives in the style of ICL (Poole 1997). Interpretations assign a probability to ground atoms, in the style of Nilsson probabilistic logic (Nilsson 1986). Queries can be answered by finding all answer sets. Differently from (Calì et al. 2009), in DISPONTE interpretations are not probabilistic and they are assigned a probability, instead of being a mapping from atoms to probabilities.

In (Gavanelli et al. 2015a) and (Gavanelli et al. 2015b), we addressed representation and reasoning for Datalog$^{\pm}$ ontologies in an Abductive Logic Programming framework, with existential and universal variables, and Constraint Logic Programming constraints in rule heads. The underlying abductive proof procedure can be directly exploited as an ontological reasoner for query answering and consistency check.

Conclusions

In this paper we have presented the algorithm TRILL for reasoning on $\mathcal{SHOIN}$ KBs and the algorithm TRILL^P for reasoning on $\mathcal{ALC}$ KBs. The experiments performed show that Prolog is a viable language for implementing DL reasoning algorithms and that their performances are comparable with those of a state-of-art reasoner such as BUNDLE.

In the future we plan to apply various optimizations to our systems in order to better manage the expansion of the tableau. In particular, we plan to carefully choose the rule and node application order. We are also studying an extension of our systems for managing KBs integrating rules and DL axioms. Moreover, we plan to exploit TRILL for implementing algorithms for learning the parameters of probabilistic DISPONTE KBs, along the lines of (Bellodi and Riguzzi 2012; 2013; Riguzzi et al. 2014).

References

Baader, F., and Peñaloza, R. 2010a. Automata-based axiom pinpointing. *J. Autom. Reasoning* 45(2):91–129.

Baader, F., and Peñaloza, R. 2010b. Axiom pinpointing in general tableaux. *J. Log. Comput.* 20(1):5–34.

Baader, F.; Calvanese, D.; McGuinness, D. L.; Nardi, D.; and Patel-Schneider, P. F., eds. 2003. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Baader, F.; Horrocks, I.; and Sattler, U. 2008. Description logics. In *Handbook of knowledge representation*. Elsevier. chapter 3, 135–179.

Beckert, B., and Posegga, J. 1995. leantap: Lean tableau-based deduction. *J. Autom. Reasoning* 15(3):339–358.

Bellodi, E., and Riguzzi, F. 2012. Learning the structure of probabilistic logic programs. In Muggleton, S. H.; Tamaddoni-Nezhad, A.; and Lisi, F. A., eds., *ILP 2011*, volume 7207 of *LNCS*, 61–75. Springer.

Bellodi, E., and Riguzzi, F. 2013. Expectation Maximization over binary decision diagrams for probabilistic logic programs. *Intel. Data Anal.* 17(2):343–363.

Bruynooghe, M.; Mantadelis, T.; Kimmig, A.; Gutmann, B.; Vennekens, J.; Janssens, G.; and Raedt, L. D. 2010. Problog technology for inference in a probabilistic first order logic. In *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, 719–724. IOS Press.

Calì, A.; Lukasiewicz, T.; Predoiu, L.; and Stuckenschmidt, H. 2009. Tightly coupled probabilistic description logic programs for the semantic web. In *Journal on Data Semantics XII*. Springer. 95–130.

Faizi, I. 2011. A Description Logic Prover in Prolog, Bachelor's thesis, Informatics Mathematical Modelling, Technical University of Denmark.

Gavanelli, M.; Lamma, E.; Riguzzi, F.; Bellodi, E.; Zese, R.; and Cota, G. 2015a. Abductive logic programming for datalog$^{\pm}$ ontologies. In *Proceedings of the 30th Italian Conference on Computational Logic (CILC2015). Accepted for publication*, CEUR Workshop Proceedings. Sun SITE Central Europe.

Gavanelli, M.; Lamma, E.; Riguzzi, F.; Bellodi, E.; Zese, R.; and Cota, G. 2015b. Abductive logic programming for Datalog$\pm$ ontologies and deductive rule integration. In *Technical Communications of the 31st International Conference on Logic Programming (ICLP15). Accepted for publication*.

Halaschek-Wiener, C.; Kalyanpur, A.; and Parsia, B. 2006. Extending tableau tracing for ABox updates. Technical report, University of Maryland.

Herchenröder, T. 2006. Lightweight semantic web oriented reasoning in Prolog: Tableaux inference for description logics. Master's thesis, School of Informatics, University of Edinburgh.

Hitzler, P.; Krötzsch, M.; and Rudolph, S. 2009. *Foundations of Semantic Web Technologies*. CRCPress.

Hustadt, U.; Motik, B.; and Sattler, U. 2008. Deciding expressive description logics in the framework of resolution. *Inf. Comput.* 206(5):579–601.

Kalyanpur, A.; Parsia, B.; Horridge, M.; and Sirin, E. 2007. Finding all justifications of OWL DL entailments. In Aberer, K., and et al., eds., *ISWC/ASWC 2007*, volume 4825 of *LNCS*, 267–280. Springer.

Kalyanpur, A. 2006. *Debugging and Repair of OWL Ontologies*. Ph.D. Dissertation, The Graduate School of the University of Maryland.

Lager, T., and Wielemaker, J. 2014. Pengines: Web logic programming made easy. *TPLP* 14(4-5):539–552.

Lukácsy, G., and Szeredi, P. 2009. Efficient description logic reasoning in prolog: The dlog system. *TPLP* 9(3):343–414.

Meissner, A. 2004. An automated deduction system for description logic with alcn language. *Studia z Automatyki i Informatyki* 28-29:91–110.

Nilsson, N. J. 1986. Probabilistic logic. *Artif. Intell.* 28(1):71–87.

Patel-Schneider, P, F.; Horrocks, I.; and Bechhofer, S. 2003. Tutorial on OWL.

Poole, D. 1997. The Independent Choice Logic for modelling multiple agents under uncertainty. *Artif. Intell.* 94(1-2):7–56.

Poole, D. 2000. Abducing through negation as failure: stable models within the independent choice logic. *J. Log. Program.* 44(1-3):5–35.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artif. Intell.* 32(1):57–95.

Ricca, F.; Gallucci, L.; Schindlauer, R.; Dell'Armi, T.; Grasso, G.; and Leone, N. 2009. OntoDLV: An ASP-based system for enterprise ontologies. *J. Log. Comput.* 19(4):643–670.

Riguzzi, F.; Bellodi, E.; Lamma, E.; and Zese, R. 2012. Epistemic and statistical probabilistic ontologies. In Bobillo, F., and et al., eds., *URSW 2012*, volume 900 of *CEUR Workshop Proceedings*, 3–14. Sun SITE Central Europe.

Riguzzi, F.; Bellodi, E.; Lamma, E.; and Zese, R. 2013a. BUNDLE: A reasoner for probabilistic ontologies. In *RR 2013*, volume 7994 of *LNCS*, 183–197. Springer.

Riguzzi, F.; Bellodi, E.; Lamma, E.; and Zese, R. 2013b. Parameter learning for probabilistic ontologies. In Faber, W., and Lembo, D., eds., *RR 2013*, volume 7994 of *LNCS*, 265–270. Springer. `http://www.ing.unife.it/docenti/FabrizioRiguzzi/Papers/RigBelLam-RR13a.pdf`.

Riguzzi, F.; Bellodi, E.; Lamma, E.; Zese, R.; and Cota, G. 2014. Learning probabilistic description logics. In *URSW III*, volume 8816 of *LNCS*, 63–78. Springer.

Sato, T. 1995. A statistical learning method for logic programs with distribution semantics. In *ICLP 1995*, 715–729. MIT Press.

Schlobach, S., and Cornet, R. 2003. Non-standard reasoning services for the debugging of description logic terminologies. In Gottlob, G., and Walsh, T., eds., *IJCAI 2003*, 355–362. Morgan Kaufmann.

Sirin, E.; Parsia, B.; Cuenca-Grau, B.; Kalyanpur, A.; and Katz, Y. 2007. Pellet: A practical OWL-DL reasoner. *J. Web Sem.* 5(2):51–53.

Vassiliadis, V.; Wielemaker, J.; and Mungall, C. 2009. Processing OWL2 ontologies using thea: An application of logic programming. In *OWLED 2009*, volume 529 of *CEUR Workshop Proceedings*. CEUR-WS.org.