Optimizing Tableau Reasoning: a Prolog-based Framework

Riccardo Zese a,∗, and Giuseppe Cota a

a Dipartimento di Ingegneria, Università di Ferrara, Via Saragat 1, 44122, Ferrara, Italy
E-mails: riccardo.zese@unife.it, giuseppe.cota@unife.it

Abstract. One of the foremost reasoning services for knowledge bases is finding all the justifications for a query. This is useful for debugging purpose and for coping with uncertainty. Among Description Logics (DLs) reasoners, the tableau algorithm is one of the most used. However, in order to collect the justifications, the reasoners must manage the non-determinism of the tableau method. For these reasons, a Prolog implementation can facilitate the management of such non-determinism.

The TRILL framework contains three probabilistic reasoners written in Prolog: TRILL, TRILL P and TORNADO. Since they are all part of the same framework, the choice about which to use can be done easily via the framework settings. Each one of them uses different approaches for probabilistic inference and handles different DLs flavours. Our previous work showed that they can achieve sometimes better results than state-of-the-art (non-)probabilistic reasoners.

In this paper we present two optimizations that improve the performances of the TRILL reasoners. In the first one, the reasoners build the hierarchy of the concepts contained in a knowledge base in order to quickly find connections among them during the expansion of the tableau. The second one modifies the order of application of the tableau rules in order to reduce the number of operations. Experimental results show the effectiveness of the introduced optimizations.

All systems can be tried online in the TRILL on SWISH web application at http://trill-sw.eu/.

Keywords: Reasoner, Axiom Pinpointing, Tableau Algorithm, (Probabilistic) Description Logic, Prolog

1. Introduction

The aim of the Semantic Web is to make information available in a form that is understandable and automatically manageable by machines. In order to realize this vision, the W3C has supported the development of a family of knowledge representation formalisms of increasing complexity for defining ontologies, called OWL (Web Ontology Languages). These formalisms are based on Description Logics (DLs). Many inference systems, generally called reasoners, have been proposed to reason upon these ontologies, such as Pellet [1], HermiT [2] and FaCT++ [3].

*Corresponding author. E-mail: riccardo.zese@unife.it.

Nonetheless, modelling real-world domains requires dealing with information that is uncertain. Therefore, many semantics for combining probability theory with OWL languages, or with the underlying DLs, were conceived [4–8]. Among them, in [9, 10] the authors proposed a semantics for probabilistic DLs, called DISPONTE. This semantics borrows the distribution semantics [11] from Probabilistic Logic Programming, that has emerged as one of the most effective approaches for representing probabilistic information in Logic Programming languages.

Probabilistic systems that can perform inference under DISPONTE are BUNDLE [12, 13] and the TRILL framework. The first one is implemented in Java and it can exploit several non-probabilistic reasoners, the lat-
ter is a framework, written in Prolog, which contains three reasoners: (i) TRILL [10, 14], able to collect the set of all the justifications and compute the probability of queries, (ii) TRILLP [10, 14], which implements in Prolog the tableau algorithm defined in [15, 16] for returning the pinpointing formula instead of the set of justifications, and (iii) TORNADO [17], which is similar to TRILLP but represents the pinpointing formula in a way that can be directly used to compute the probability of the query. To set which reasoner to use, the user needs only to specify it at the beginning of the knowledge base.

Most DL reasoners adopt the tableau algorithm [18, 19]. This algorithm applies some expansion rules on a tableau, a representation of the assertional part of the KB. However, some of these rules are nondeterministic, requiring the implementation of a search strategy in an or-branching search space. The reasoners contained in the TRILL framework exploit Prolog’s backtracking facilities for performing the search. In addition, the experiments performed in [17] showed that a Prolog implementation of the tableau algorithm can achieve competitive or even better results than other state-of-the-art (non-)probabilistic reasoners.

In this paper, we present two optimizations that improve the reasoning performance of the TRILL framework. In the first one, the reasoners build the hierarchy of the concepts contained in a knowledge base in order to quickly find connections among them during the expansion of the tableau. The second one modifies the application order of tableau rules in order to reduce the number of operations.

All the probabilistic reasoners in the TRILL framework are available in the TRILL on SWISH web application [20] at http://trill-sw.eu/.

Moreover, we present an extensive experimental evaluation where we compared the previous version of the TRILL framework with the latest one containing the optimizations. The experimental results show that, when the KB is relatively large and complex, the introduced extensions can significantly speed up both regular and probabilistic queries.

Furthermore, it is clear from the results that the overhead produced by the probabilistic computation is usually negligible compared with the time needed by the reasoning phase. All the KBs used in the experiments are available at https://github.com/rzese/docker-trill or packed in a docker container available at https://hub.docker.com/r/rzese/trill.

The paper is organized as follows: Section 2 illustrates the needed background by briefly introducing DLS, describing the tableau algorithm and presenting the probabilistic semantics DISPONTE, which motivates the need to find all the justifications and thus the need to build highly optimized systems. Section 3 presents the former TRILL framework, followed by a detailed description of the introduced optimizations in Section 4. Section 5 discusses related work. Section 6 shows the experimental evaluation and Section 7 concludes the paper and discusses future directions. Finally, the Appendix presents details about the Prolog implementation of the optimizations presented in Section 4.

2. Background

2.1. Description Logics

Description Logics (DLs) [21, 22] syntax is based on individuals, representing objects of the domain, concepts, which group individuals sharing the same characteristics, and roles, connecting pairs of individuals or individuals with datatype values (i.e. integers, strings, etc.). There are many DL languages that differ in the constructs that are allowed for defining concepts and roles. We first briefly describe the DL $SHI$ and then its extension $SHIQ$.

Let us consider a set of atomic concepts $C$, a set of atomic roles $R$ and a set of individuals $I$. A role could be an atomic role $R \in R$ or the inverse $R^-$ of an atomic role $R \in R$. We use $R^-$ to denote the set of all the inverses of roles in $R$. Each $A \in A$, Top (also called $\text{Thing}$ or $\top$) and Bottom (also called $\text{Nothing}$ or $\bot$) are concepts. If $C, C_1$ and $C_2$ are concepts and $R \in R \cup R^-$, then $(C_1 \cap C_2), (C_1 \cup C_2)$ and $\neg C$ are concepts, as well as $\exists R.C$ and $\forall R.C$.

A knowledge base $(KB) K = (T, R, A)$ consists of a TBox $T$, an RBox $R$ and an ABox $A$. An RBox $R$ is a finite set of transitivity axioms $\text{Trans}(R)$ and role inclusion axioms $R \subseteq S$, where $R, S \in R \cup R^-$. A TBox $T$ is a finite set of concept inclusion axioms $C \subseteq D$, where $C$ and $D$ are concepts. An ABox $A$ is a finite set of concept membership axioms $a : C$ and role membership axioms $(a, b) : R$, where $C$ is a concept, $R \in R$ and $a, b \in I$. With respect to $SHI$, the DL $SHIQ$ adds new constructs for the definition of qualified number restrictions, i.e., given a concept $C$ and a simple role $R \in R \cup R^-$, qualified number restrictions are concepts of the form $\geq nR.C$ and $\leq nR.C$ for an integer $n \geq 0$. Roles in number restriction must be simple to ensure decidability. A role $R \in R \cup R^-$ is
simple if it is not transitive and it has not any transitive sub-role.

A SHI or SHIQ KB is usually assigned a semantics in terms of interpretations \( \mathcal{I} = (\Delta^\mathcal{I}, \mathcal{I}) \), where \( \Delta^\mathcal{I} \) is a non-empty domain and \( \mathcal{I} \) is the interpretation function, which assigns an element in \( \Delta^\mathcal{I} \) to each \( a \in \mathcal{I} \), a subset of \( \Delta^\mathcal{I} \) to each concept and a subset of \( \Delta^\mathcal{I} \times \Delta^\mathcal{I} \) to each role (we do not consider datatype roles). A query \( Q \) over a KB \( \mathcal{K} \) is usually an axiom for which we want to test the entailment from the KB, written as \( \mathcal{K} \models Q \).

**Example 1.** The following KB is inspired by the ontology people+pets [23]:

\[
\exists \text{hasAnimal}. \text{Pet} \sqsubseteq \text{PetOwner}
\]

\[
\text{Cat} \sqsubseteq \text{Pet}
\]

\[
\text{fluffy} : \text{Cat} \quad \text{(kevin, fluffy)} : \text{hasAnimal}
\]

\[
\text{tom} : \text{Cat} \quad \text{(kevin, tom)} : \text{hasAnimal}
\]

It states that individuals that own an animal which is a pet are pet owners and that kevin owns the animals fluffy and tom, which are cats. Moreover, cats are pets. The KB entails the query \( Q = \text{kevin} : \text{PetOwner} \).

2.2. The Tableau Algorithm

In this section we discuss the tableau algorithm [21], one of the most common approaches to answer queries. In its basic definition, a tableau is an ABox represented using a tuple \( G = (V, E, \mathcal{L}, \not\in) \) that contains a directed graph \( (V, E) \) where each node of \( V \) corresponds to an individual \( a \) and is labelled with the set of concepts \( \mathcal{L}(a) \) to which \( a \) belongs. Each edge \( (a, b) \in E \) in the graph is labelled with the set of roles \( \mathcal{L}(a, b) \). The binary predicate \( \not\in \) is used to specify inequalities between nodes. \( G \), which is also called completion graph, is initialized with a node for each individual \( a \) of the KB, labelled with all concepts \( C \) such that \( a : C \in \mathcal{K} \), and an edge \( e = (a, b) \) labelled with \( R \) for each assertion \( (a, b) : R \in \mathcal{K} \).

A tableau algorithm proves an axiom by refutation, starting from a tableau that contains the negation of the axiom. Then, the 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. If no clashes are found, the tableau represents a model for the negation of the query, which is thus not entailed.

The tableau expansion rules can be deterministic or non-deterministic. The first type takes as input one graph and returns a single updated graph. In order to manage non-deterministic rules, a set \( T \) of completion graphs is built instead of a single one. \( T \) is initialized with a single completion graph, let’s say \( G_0 \), which is initialized as described above for \( G \). Every application of a rule modifies \( T \). In particular, the application of a deterministic rule to a completion graph \( G_i \) returns a new graph \( G' \) which replaces \( G_i \). In the case of application of a non-deterministic rule, instead, the tableau \( G_i \) to which the rule is applied is replaced by the set of tableaux returned by the rule.

For ensuring the termination of the algorithm, a special condition known as blocking [24, 25] is used.

**Soundness** and **completeness** of the tableau algorithm are proved in [21].

2.2.1. Representing a Covering Set of Justification

An important problem to solve is finding the covering set of justifications for a given query. This non-standard reasoning service is also known as **axiom pinpointing** [26] and it is useful for tracing derivations and debugging ontologies. This problem has been investigated by various authors [26–29].

The covering set of justifications can be represented by means of either **minimal axiom sets**, which basically correspond with justifications, or a **pinpointing formula**.

**Definition 1** (Explanation). Given a KB \( \mathcal{K} \) and a query \( Q \), a subset of logical axioms \( \mathcal{E} \) of a KB \( \mathcal{K} \) such that \( \mathcal{E} \models Q \) is called explanation.

**Definition 2** (Justification). A justification is an explanation such that it is minimal w.r.t. set inclusion. Formally, we say that an explanation \( J \subseteq \mathcal{K} \) is a justification if for all \( J' \subseteq J \), \( J' \not\models Q \), i.e. \( J' \) is not an explanation for \( Q \).

**Definition 3** (Covering set of justifications). The set of all the justifications for the query \( Q \) is the covering set of justifications for \( Q \). Given a KB \( \mathcal{K} \), the covering set of justifications for \( Q \) is denoted by \( \text{ALL-JUST}(Q, \mathcal{K}) \).

The covering set of justifications can be represented also by means of a **pinpointing formula**, as presented in [15, 16]. This formula is built using Boolean variables (one for each axiom of the KB) and the conjunction and disjunction connectives. Let’s assume that each axiom \( E \) of a KB \( \mathcal{K} \) is associated with a propositional variable, indicated with \( \text{var}(E) \). The set of all propositional variables is indicated with \( \text{var}(\mathcal{K}) \). A valuation \( v \) of a monotone Boolean formula is the set of propositional variables that are true. For a valuation \( v \subseteq \text{var}(\mathcal{K}) \), let \( \text{K}_v := \{ t \in \mathcal{K} | \text{var}(t) \in v \} \).
Definition 4 (Pinpointing formula). Given a query $Q$ and a KB $\mathcal{K}$, a monotone Boolean formula $\phi$ over $\text{var}(\mathcal{K})$ is called a pinpointing formula for $Q$ if for every valuation $\nu \subseteq \text{var}(\mathcal{K})$ it holds that $\mathcal{K}, \nu \models Q$ iff $\nu$ satisfies $\phi$.

The pinpointing formula compactly encodes the set $\text{ALL-JUST}(Q, \mathcal{K})$ as proved in Lemma 2.4 of [16]. Moreover, as reported in Section 7 of [16], this approach is correct and terminating for the DL $\text{SHIT}$. 

Example 2. Consider the KB shown in Example 1. We associate Boolean variables with axioms as follows:

\[ E_1 = \exists \text{hasAnimal}.\text{Pet} \sqsubseteq \text{PetOwner} \]
\[ E_2 = \text{fluffy} : \text{Cat} \]
\[ E_3 = \text{tom} : \text{Cat} \]
\[ E_4 = \text{Cat} \sqsubseteq \text{Pet} \]
\[ E_5 = (\text{kevin}, \text{fluffy}) : \text{hasAnimal} \]
\[ E_6 = (\text{kevin}, \text{tom}) : \text{hasAnimal} \]

Let $Q = \text{kevin} : \text{PetOwner}$ be the query, then

\[ \text{ALL-JUST}(Q, \mathcal{K}) = \{ E_5, E_2, E_1, E_3 \}, \{ E_6, E_3, E_1, E_4 \} \]

while the pinpointing formula is

\[ (E_5 \land E_2) \lor (E_6 \land E_3) \land E_4 \land E_1 \]

2.2.2. Extending the Tableau Algorithm to Solve the Axiom Pinpointing Problem

In order to solve the axiom pinpointing problem, the tableau algorithm has been modified so that each expansion rule updates a tracing function $\tau$ as well, which associates each concept (role) in the label of a node (edge) with justifications [30] or the pinpointing formula [16] found so far.

During the initialization of the tableau, $\tau$ is initialized as empty 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 $\{ \langle a, b \rangle : R \}$ are assigned if $a : C$ and $\langle a, b \rangle : R$ are in the ABox. The tableau expansion rules for $\text{SHITQ}$ are shown in Figure 1, where the rules that are not used for $\text{SHIT} \ DL$ are marked by $(*)$. Here, $\text{Add}(C, a)$ stands for the addition of a concept $C$ to $\mathcal{L}(a)$ while $\text{Add}(R, \langle a, b \rangle)$ represents the addition of the role $R$ to $\mathcal{L}(\langle a, b \rangle)$.

The values of the tracing function associated with the labels which cause the clash are then put together to form the covering set of justification. The rules in Figure 1 are divided into deterministic and non-deterministic. As stated above, the first, when applied to a tableau, produce a single new tableau. The latter, when applied to a tableau, produce a set of tableaux.

Unfortunately, a classical tableau algorithm returns a single justification (or a Boolean formula not representing all the justifications) using the tracing function. To solve the axiom pinpointing problem, reasoners must explore the entire search space of the possible explanations. Classical tableau-based systems, implemented using imperative languages, forces the tableau algorithm to find a new justification in several ways. The most used approach, called Hitting Set Tree (HST) [31], repeatedly removes axioms from the KB following the justifications previously found and executes the tableau algorithm to find new justifications w.r.t. the modified KB. For instance, given a KB $\mathcal{K}$ and a query $Q$, if the justification $J_1 = \{ E_1, E_2, E_3 \}$ is found, where $E$ are axioms, to avoid the generation of the same justification, the HST algorithm tries to find a new justification on $\mathcal{K}' = \mathcal{K} \setminus E_1$. We can say that it creates a choice point where it must choose which axiom will be removed from $\mathcal{K}$. If $\mathcal{K}'$ does not lead to a new justification, it backtracks to the choice point and tries to find another justification by removing other axioms contained in $J$, one at a time, i.e., looks for a new justification w.r.t. $\mathcal{K} \setminus E_2$ and $\mathcal{K} \setminus E_3$. Otherwise, if a new justification $J_2 = \{ E_1, E_5 \}$ is found, before backtracking, the HST algorithm creates a new choice point and calls the reasoner to find a new justification w.r.t. $\mathcal{K}'' = \mathcal{K}' \setminus E_3$. At this point, again, if a new justification is found, one axiom is removed from the KB and the query tested w.r.t. the new KB, otherwise the HST backtracks to the last choice point to continue the search. In our example, $E_5$ is removed from $\mathcal{K}'$. When both $E_4$ and $E_5$ have been tested, the algorithm backtracks to the previous choice point and explores the other choices. In our example, the HST tries to remove $E_2$ and $E_3$ from $\mathcal{K}$. The algorithm terminates when all the backtracking points are explored and returns all the justifications found.

A fruitful approach to avoid implementing such an algorithm is to rely on the backtracking facilities that are built-in in Prolog. This idea has been explored by many researchers that implemented the tableau using Prolog [32–36]. Another possibility is to apply an extension of (disjunctive) ASP [37] or an abductive proof procedure to perform ontological reasoning [38, 39]. We proposed three systems, TRILL [10, 14], TRILL$^P$ [10, 14], and TORNADO [17], that im-
Deterministic rules:

\[ \rightarrow \text{unfold: if } A \in \mathcal{L}(a), \text{ A atomic and } (A \subseteq D) \in K, \text{ then} \]

\[ \rightarrow CE: \text{ if } (C \subseteq D) \in K, \text{ with } C \text{ not atomic, then} \]

\[ \rightarrow (\neg C \cup D) \notin \mathcal{L}(a), \text{ then } Add((\neg C \cup D), a), \tau((\neg C \cup D), a) := \{C \subseteq D\} \]

\[ \rightarrow \Gamma: \text{ if } (C_1 \cap C_2) \in \mathcal{L}(a), \text{ then} \]

\[ \rightarrow \exists: \text{ if } \exists S.C \in \mathcal{L}(a), \text{ then} \]

\[ \rightarrow a \text{ has no } S \text{-neighbour } b \text{ with } C \in \mathcal{L}(b), \text{ then} \]

\[ \rightarrow \forall: \text{ if } \forall(S.C) \in \mathcal{L}(a), \text{ and there is an } S \text{-neighbour } b \text{ of } a, \text{ then} \]

\[ \rightarrow \forall^+: \text{ if } \forall(S.C) \in \mathcal{L}(a), \text{ and there is an } R \text{-neighbour } b \text{ of } a, Trans(R) \text{ and } R \subseteq S, \text{ then} \]

\[ \rightarrow \exists^+: \text{ if } (\exists S.C) \in \mathcal{L}(a), \text{ then} \]

Non-deterministic rules:

\[ \rightarrow \mathcal{L}: \text{ if } (C_1 \cap C_2) \in \mathcal{L}(a), \text{ then} \]

\[ \rightarrow \exists S.C \in \mathcal{L}(a), \text{ and there are } m \text{-neighbours } b_1, \ldots, b_m \text{ of } a \text{ with } m > n \]

\[ \rightarrow \text{ch}(\ast): \text{ if } (\exists S.C) \in \mathcal{L}(a), \text{ and there is an } S \text{-neighbour } b \text{ of } a \text{ with } (C \subseteq \neg C) \cap \mathcal{L}(b) = \emptyset, \text{ then} \]

\[ \tau(C, b) := \tau((\exists S.C), a) \cup \tau(S, \langle a, b \rangle) \in G_1, \tau(\neg C, b) := \tau((\exists S.C), a) \cup \tau(S, \langle a, b \rangle) \in G_2 \]

Axiom pinpointing problem is also important for probabilistic inference. In the following we briefly describe the DISPONTE semantics [9], which requires the set of all the justifications to compute the probability of the queries.

DISPONTE [9, 10] applies the distribution semantics [11] to Probabilistic Description Logic KBs. In DISPONTE, a probabilistic knowledge base \( K \) contains a set of probabilistic axioms which take the form

\[ p :: E \]  

where \( p \) is a real number in \([0, 1]\) and \( E \) is a DL axiom. The probability \( p \) can be interpreted as the degree of our belief in the truth of axiom \( E \).
Following the semantics, DISPONTE associates independent Boolean random variables to the DL axioms. The set of axioms that have the random variable assigned to 1 constitutes a world, with a probability computed by multiplying the probability $p_i$ for each probabilistic axiom $E_i$ included in the world by the probability $1 - p_i$ for each probabilistic axiom $E_i$ not included in the world. The probability of a query is the sum of the probabilities of the worlds where the query is true.

For more detail about probabilistic inference with the TRILL framework, we refer the interested reader to [17].

### 3. The TRILL Framework

The TRILL framework contains a web interface called TRILL on SWISH [20] and three inference systems, a.k.a. reasoners, called TRILL [10, 14], TRILL* [10, 14], and TORNADO [17], which use the Prolog language to implement the tableau algorithm and compute the probability of the queries under the DISPONTE semantics. For the sake of clarity, in the following, if we need to consider all the three systems together, we will refer to “TRILL framework” or “TRILL systems”. While if we consider the single reasoner, we will use TRILL, TRILL*, and TORNADO.

TRILL solves the $\text{ALL-JUST}(Q, K)$ problem for the $\text{SHIQ}$ DL, it can return the set of all the justifications and the probability of the queries computed by means of knowledge compilation using BDDs\(^1\) [17]. TRILL* modifies TRILL in order to compute the pinpointing formula instead of the set of all the justifications. It can also compute the probability of the query by representing the formula with a BDD. Analogously, TORNADO builds the pinpointing formula but, differently from TRILL*, it is directly represented as a BDD, avoiding some exponential blow-ups in the inference process. As TRILL*, it is implemented in Prolog and supports the $\text{SHIQ}$ DL. However, TORNADO is able to perform only probabilistic query answering because it cannot return justifications for the query. Since all the three reasoners are part of the same framework, they can be switched by means of a simple directive included at the beginning of the file containing the KB as follows:

```
% Include the library
:- use_module(library(trill)).
% Load TRILL.
:- trill.
% Use :- trillp or :- tornado
% to choose the algorithm.

% KB axioms
...
```

The TRILL framework is implemented in SWI-Prolog [40], the code of all three systems is available at https://github.com/rzese/trill.

The TRILL framework forms a layer cake, shown in Figure 2, designed to facilitate its extension. The lower layer, called “Translation Utilities”, contains a library for translating the input KB in case it is given in the RDF/XML format and loading it in the Prolog database, in order to be accessible to the upper layers. This layer contains the module utility_translation which is based on the Thea2 library [41] for converting OWL DL KBs into Prolog. Thea2 performs a direct translation of OWL axioms into Prolog facts. Then, there is the “TRILL Library” layer containing system-specific modules, one per system. Each module contains predicates that act differently in the three systems and settings that are specific to each reasoner or that do not share the same values. Finally, the upper layer contains predicates and settings that are in common for all the systems and

---

\(^1\)Basically, a BDD is a rooted graph used to represent a function of Boolean variables. It has one level for each Boolean variable, each node of the graph has two children corresponding respectively to the 1 value and the 0 value of the variable associated with the node. Its leaves are either 0 or 1.
defines the user interface and the queries that can be asked. In particular, TRILL, TRILLP, and TORNADO can answer concept and role membership queries, subsumption queries and can test the unsatisfiability of a concept of the KB or the inconsistence of the entire KB. They can be executed by means of a SWI-Prolog console or tested online with the TRILL on SWISH web application at http://trill-sw.eu/.

In order to represent the tableau, a pair Tableau = (A, T) is used, 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 KB in TRILL, the pinpointing formula in TRILLP, and the BDD representing the pinpointing formula in TORNADO. T contains the structure of the tableau and information needed during its expansion. For detailed description of the implementation of the three systems we refer to the papers which presented them. In the following we recap the general design of the systems in order to better understand the differences between the old and the new version. Details about the implementation, with code snippets corresponding to all the procedures shown in the following, can be found in the Appendix.

Expansion rules are applied in order by the procedure APPLY_ALL_RULES, first the deterministic ones and then the non-deterministic ones, as shown in Figure 3. It is worth noting that, for SHIQ DLs, the order of rule application does not affect the completeness of the process of finding a covering set of justifications. Expansion rules can only add new assertions into the tableau, therefore if a rule is applicable on a tableau, it will be so also in every its expansion. In this case, the choice of which expansion rule to apply introduces “don’t care” non-determinism. Differently, “don’t know” non-determinism is introduced by non-deterministic rules, since a single tableau is expanded into a set of tableaux.

In our case, SHIQ DLs, the tableau algorithm terminates with any rule application order. However, this is not true for more expressive DLs like SHOTIQ, and therefore a rule application strategy must be adopted to assure termination [18].

Even though for SHIQ DLs the expansion rules application order does not affect the completeness of the algorithm, it can significantly affect the performance [42, 43]. Usually, non-deterministic rules are tried as late as possible in order to avoid performing same expansions more than once.

Example 3. Consider the following simple KB:

\[
\begin{align*}
C & \sqsubseteq D \\
x & : C \\
x & : E \sqcup F
\end{align*}
\]

The initial tableau contains a single node corresponding to the individual x whose label \(L(x)\) is (by omitting the tracing function) \(\{C, E \sqcup F\}\). Given this tableau, both the \(\rightarrow\) unfold rule and the \(\rightarrow\) rule are applicable. If we decide to first apply the \(\rightarrow\) unfold rule we will obtain a new tableau where \(L(x) = \{C, E \sqcup F, D\}\). Now, the application of the \(\rightarrow\) rule leads to two tableaux: in the first \(L(x) = \{C, E \sqcup F, D, E\}\), in the second we have \(L(x) = \{C, E \sqcup F, D, F\}\). In this way we have only two rule applications.

Otherwise, if we apply the \(\rightarrow\) rule first, we obtain two tableaux with \(L(x) = \{C, E \sqcup F, F, E\}\) and \(L(x) = \{C, E \sqcup F, F\}\) respectively. At this point, the \(\rightarrow\) unfold rule must be applied to both tableaux in order to fully expand them. In this case, we have three rule applications instead of two.

Therefore, even if the final results are the same, the performance is sensibly different.

The procedure APPLY_DET_RULES (lines 10-20) takes as input the list of deterministic rules and the current tableau and returns a new tableau obtained by the application of one of the rules or, if no rules can be applied, it returns the result obtained by calling the APPLY_NONDET_RULES procedure (line 18).

This procedure is used to sequentially try non-deterministic rules. It 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 (lines 21-31). 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 and a tableau from the list is non-deterministically chosen. If no rule is applicable, the input tableau is returned and the rule application stops, otherwise a new round of rule application is performed. Finally, by exploring all the possible choices opened by the procedure APPLY_NONDET_RULES, TRILL systems collect the list of all the possible tableaux and look for all the clashes contained in them in order to collect all the possible explanations. Information about the code implementing these procedures can be found in Section A.3 in the Appendix.
Fig. 3. Application of the expansion rules by means of the procedures \texttt{APPLY\_ALL\_RULES}, \texttt{APPLY\_DET\_RULES} and \texttt{APPLY\_NONDET\_RULES}. The list \texttt{Rules} contains the available rules and is different in TRILL, TRILL$^2$ and TORNADO.

4. Improving TRILL

The improvements implemented in the new version of the TRILL framework are basically twofold: a detailed analysis of the KB during the loading phase and an optimization of the application order of the tableau expansion rules. As we will see in the following, the first extension is used for speeding up the application of some expansion rules while the second for speeding up the general expansion process. In Section 6 we compare the two versions of the TRILL framework to show that the implemented extensions can improve the performances of the reasoners. In particular, we expect a little overhead in the initial phase of the query answering process, especially in the case of simple KBs, that will be amortized as much as more complex becomes answering the query.
The extensions described in this paper are available in a git repository at https://github.com/rzese/trill. They are implemented in the branch called trill-beta-version, in order to maintain both versions downloadable to allow the replication of the tests presented in Section 6.

4.1. KB Analysis

The first improvement allows TRILL to collect useful information about the KB during its loading in the Prolog database. In particular, the TRILL systems make use of a map, a collection of key-value pairs, to keep a structure in memory containing information such as the number and the list of (complex) concepts, individuals and properties of the KB, together with the concept hierarchy and information about the axioms used to build the hierarchy. These pieces of information will be kept in memory so that the time to collect them is paid only once at the start-up. The management of this data structure is implemented in a new module called utility_kb, which defines the layer “KB Utilities” positioned on top of the layer dedicated to the translation and the loading of the axioms, as shown in Figure 4. In turn, the module utility_translation has been refactored in order to quickly and recursively analyse the information contained in the KB and prepare it to be saved in the structure containing the hierarchy.

The structure built by TRILL systems contains many fields, in the following we report only those useful for the optimization presented in this paper:

- hierarchy: an unweighted graph representing the concepts hierarchy. It is initialized as a graph containing two disconnected nodes called 0 and \( n \) representing the OWL built-in concepts Thing (⊤) and Nothing (⊥) respectively.
- disjointClasses: a list of pairs indicating concepts that are disjoint.
- classes: is a map associating each node of the hierarchy tree to the concepts it represents. In particular, the concept Thing is associated to the 0 node while the concept Nothing to the \( n \) node.
- explanations: a list of pairs mapping each edge of the hierarchy with the set of axioms explaining the edge. These axioms can be seen as justifications for the edge. which, in turn, specifies a subclass relation between two concepts. This will be used during the expansion of the tableau to rapidly update the justifications.

After the initialization, each concept of the KB is added through the \texttt{ADD}\_CLASS procedure shown in Figure 5. The code implementing \texttt{ADD}\_CLASS is shown in Appendix B. Given the hierarchy structure \texttt{HierIn} and a concept \texttt{Class}, \texttt{ADD}\_CLASS looks for the concept in the map \texttt{classes} contained in \texttt{HierIn} (line 2). If the class is already present in the list, it means that the hierarchy already contains such a concept and therefore nothing must be done. Otherwise, the new concept is added to \texttt{classes} (line 5) using the identifier of the new node generated on line 4. This new node is then included in the concept hierarchy graph and linked to the \( \top \) concept (line 6) by means of the \texttt{ADD}\_EDGE procedure. This procedure simply takes a hierarchy graph and the identifiers of two nodes, adds a link between them and returns the updated graph. Moreover, the \texttt{ADD}\_\texttt{SUBCLASS}\_\texttt{EXPL} (line 7) procedure adds to the list of justifications a new explanation for the edge connecting the two classes taken in input. In this case, this explanation is simply the axiom \texttt{Class} \( \sqsubseteq \top \). The management of the individuals and roles is slightly different. In these cases, they are just added to their respective lists. These lists are used to check the arguments of the query to be answered by TRILL.

Then, subsumption axioms are considered to update the hierarchy. For each equivalent-classes axiom, the hierarchy and the map of nodes are checked to find the nodes corresponding to the classes specified by the
1: procedure ADD_CLASS(HierIn, Class)
2: HierIn: the input hierarchy structure.
3: Class: the class to be added.
4: Returns HierOut: the output hierarchy structure, or fails.
5: if not HierIn.FIND(Class) then  \( \triangleright \) Checks if Class is not present in HierIn
6:     HierOut ← HierIn
7:     ID_class ← generates the identifier of the concept Class
8:     HierOut.classes ← HierOut.classes \cup \{ \{ID_class, Class\}\}
9:     \((ID = 0)\) and Class in the hierarchy
10:     HierOut.hierarchy ← ADD_EDGE(HierOut.hierarchy, 0, ID_class)  \( \triangleright \) Adds a new link between the concept \( \top \)
11: \((ID = 0)\) and Class in the hierarchy
12:     HierOut.explanations ← ADD_SUBCLASS_EXPL(HierOut.explanations, Class, \( \top \))
13: end if
14: end procedure

Fig. 5. Pseudo-code of the procedure ADD_CLASS. It adds a new concept Class to the hierarchy by updating its structure and the list containing information about all the classes of the KB. The procedure FIND is used to perform the search by value in the dictionary.

axiom. Then, the collected nodes are merged into one node and the links in the graph are consequently updated. To do so, in the classes map, all the entries for the merged nodes are removed and a new pair key-value is added, where the value is a list containing the concepts that are equivalent. The new key is then used to gather all the links of the graph previously connected with one of the merged nodes, a node is added to the hierarchy graph representing such key and connected to the other nodes using the links above.

On the other hand, for each sub-class axiom of the form SubClass \( \sqsubseteq \) SupClass a new link in the hierarchy must be added. To do so, the procedure ADD_HIERARCHY_LINK shown in Figure 6 is called.

This procedure first checks if the two concepts SubClass and SupClass are mapped to different nodes (lines 2-4). If it is so, it calls ADD_LINK_ID (line 5), this procedure adds the new link by looking first if the information SupClass \( \sqsubseteq \) SubClass is already included in the hierarchy (line 12). If this is true, it means that there is a path connecting SupClass and SubClass. For example, if we already know that SupClass \( \sqsubseteq \) E, and E \( \sqsubseteq \) SubClass, it means that SupClass \( \sqsubseteq \) SubClass. In this case, when we add to the hierarchy the information SubClass \( \sqsubseteq \) SupClass we must merge these concepts as they are all equivalent. Merging these nodes leads to the need of updating the hierarchy by changing the links starting from or reaching the nodes that we have just merged in order to correctly connect the new merged node (as seen for equivalent-class axioms).

This operation is done by procedure MERGE_CLASSES called on line 13. Otherwise, a new link between the nodes corresponding to SupClass and SubClass is added and the edge going from \( \top \) to SubClass, if present, is removed to facilitate the management of the hierarchy graph (lines 15-17).

Finally, if the two concepts SubClass and SupClass are mapped to the same node, ADD_HIERARCHY_LINK simply adds the new explanation for the edge (line 7). This means that there may be more ways to explain why SubClass \( \sqsubseteq \) SupClass. This set of explanations will be considered during the application of the expansion rules. For example, consider the case where the \( \rightarrow \) unfold rule, starting from an individual belonging to SubClass, will add to the label of this individual also SupClass. If there are more than one explanation that prove this subsumption, it will consider all these explanations in a single application of the rule. The same result in the old version would be achieved after more than one application of the \( \rightarrow \) unfold rule.

Details about the implementation of the procedure ADD_HIERARCHY_LINK can be found in Appendix B.

Moreover, subsumption axioms are considered to find sub-class connections also regarding complex concepts, e.g., if there is an axiom C \( \sqsubseteq \) D and the hierarchy contains two nodes representing the concepts \( \exists R.C \) and \( \exists R.D \), the two nodes are linked. For each of these connections, the steps described above are performed to update the hierarchy. Finally, the set of disjoint classes axioms dca are considered in order to add links from node n, corresponding to \( \bot \), to unsatisfiable classes if there are any. This is done by checking in the hierarchy built so far if two classes that must be disjoint given the axioms in dca are connected, i.e., there is a subsumption relation between them. In this case, these two classes and all their subclasses are unsatisfiable, thus they must be connected to the n node as they are subclass of \( \bot \).
The structure is used during the expansion of the tableau in the → unfold rule that, instead of looking for a subsumption axiom $C \sqsubseteq D$, searches for a concept $D$ in the hierarchy. A possible further optimization here is to collect all the concepts directly connected to $C$ instead of a single concept per rule call. Moreover, the structure can be exploited to quickly control whether the KB is inconsistent by checking if some individuals belong to an unsatisfiable concept, i.e., if the concept is equivalent to ⊥.

### 4.2. Improving the Tableau Expansion

As described in Section 2.2, the tableau algorithm can return a single justification. Therefore, a reasoner must implement a strategy to search the entire search space. Classical tableau-based systems, implemented using imperative languages, forces the tableau algorithm to find a new justification by implementing ad-hoc algorithms such as the Hitting Set Algorithm. As seen in Section 3, the previous versions of TRILL, TRILL$^p$ and TORNADO leave the burden of this search to Prolog’s backtracking facilities, the rules simply pick a label from the tableau and try to expand it. However, this approach is far from being optimized.

**Example 4.** Suppose that the tableau contains $k$ assertions $\{a : C_1, \ldots, a : C_k\}$ and that the KB contains only the axiom $C_k \sqsubseteq C_{k+1}$. Every expansion rule is called following a certain order usually defined in the specific system. In this example, the only rule that will modify the tableau is the → unfold rule. Therefore, to fully expand the tableau, all the rules that are called before will be called twice. The → unfold rule will also be called twice, the first to modify the tableau and the second to check whether a new modification can be done. The remaining rules, those considered after the → unfold, will be called once. Every time a rule is called, a set of tests is performed to check whether the
rule can be applied or not. For the sake of simplicity,
we assume that every test is performed by an atomic
instruction but, actually, the complexity of the applica-
tility test heavily depends on the expansion rule itself.

Let’s suppose that we try $n$ rules and the $→$ unfold
rule is the third in the order, suppose also that the as-
sertions are tested in the order of their subscript. In the
first round the first and second rules are tested on all
the $k$ assertions and they all fail. Then, the $→$ unfold
rule is applied adding the new assertion $a : C_{k+1}$ to
the tableau. In the second round of application of the
expansion rules, each expansion rule is tested for each
of the $k + 1$ assertions, but no one can modify the
tableau, which is fully expanded. At the end of the pro-
cess $3k + n(k + 1)$ tests have been performed to decide
whether a rule can be applied or not.

In the worst case, where the $→$ unfold is the last rule
applied, the number of tests is $nk + n(k + 1) = 2nk + n$
However, as one can see from Figure 1, every rule
to be applied looks for the presence of an assertion in
the tableau. This means that, if an assertion has already
been fully tested (every rule has already been applied
and its tracing function has not been changed by now),
it does not lead to further expansions of the tableau
and, hence, it won’t until the end of the expansion.

Following this observation, we changed the way
TRILL, TRILL$^p$ and TORNADO apply the expansion
rules. In the new version, in each round of rule applica-
tion, instead of trying every rule w.r.t. every assertion,
they keep updated a structure called $ExpansionPairQ$.

This structure is composed by a pair of expan-
sion queues: $DetQueue$ and $NonDetQueue$. The first
one contains every assertion not triggering any non-
deterministic rules contained in the tableau, while the
second contains assertions triggering non-deterministic
rules, i.e., conjunctions of concepts ($\sqcap$) and maximum
cardinality restrictions ($\leq$). Each assertion can be in-
cluded in only one queue and, in this queue, it can ap-
ppear only once. Both of them are initialized together
with the initial tableau. Each tableau is associated with
an instance of $ExpansionPairQ$, as every tableau has
its own pair of queues depending on the applied ex-
ansion rules. The procedure $APPLY\_ALL\_RULES$ of
Figure 3 is replaced by the $EXPAND\_QUEUES$ proce-
dure, shown in Figure 7, which calls a slightly differ-
et version of $APPLY\_ALL\_RULES$ which takes now
two arguments instead of one.

The $EXPAND\_QUEUES$ procedure, whose code is
reported in Section A.3 in the Appendix, extracts
from $ExpansionPairQ$ the next assertion to expand $AssToExpand$. It first tries to dequeue $DetQueue$, i.e.
the queue containing assertions triggering only de-
terministic rules, then, if $AssToExpand$ is null, i.e.,
$DetQueue$ is empty, it dequeues $NonDetQueue$, i.e. the
queue containing those triggering non-deterministic
rules (lines 3-6).

The $EXPAND\_QUEUES$ procedure stops when there
are no more assertions to expand. The procedure
$APPLY\_ALL\_RULES$ used in Figure 7 differs from
that shown in Figure 3 because it takes as input
also the assertion $AssToExpand$. Then, this assertion
is passed to the expansion rules that try to expa-
tend the tableau considering the assertion. The
rules now follow a slightly different interface: de-
terministic tableau expansion rules are of the form
rule name($TabIn$, $AssToExpand$, $TabOut$), while
non-deterministic rules are implemented as rule
name($TabIn$, $AssToExpand$, $TabList$). They
take as input the current tableau $TabIn$ and the asser-
tion to handle $AssToExpand$ and return respectively
the tableau $TabOut$ and the list of tableaux $TabList$
created by the application of the rule to $TabIn$ by con-
sidering the assertion $AssToExpand$.

Moreover, the rules exploit the hierarchy created
during the KB loading to quickly find new concepts to
add to the individual instead of looking for axioms.

Example 5. Let’s consider the case depicted in Exam-
ple 4, where we have a tableau containing $k$ assertions
$\{a : C_1, \ldots, a : C_k\}$, the axiom $C_k \sqsubseteq C_{k+1}$, and $n$ ex-
pansion rules of which the $→$ unfold rule is applied as third.

The new version of TRILL tries to apply every rule
that match with the assertion and, for each matching
rule, performs the applicability test. Let assume for
simplicity that every rule matches every assertion in
the tableau, thus every rule performs the test for every
assertion. Under this simplification, similarly to Ex-
ample 4, in the first round TRILL tests all the asser-
tions. The main difference in this case is in the second
round, where $DetQueue$ contains only the new asser-
tion $a : C_{k+1}$, therefore, the number of tests in this case
is $nk + n$, which corresponds with the worst case and
it is clearly less than $3k + n(k + 1) = nk + 3k + n$ or
even less than the worst case $2nk + n$ of the previous
version.

Note that many rules match only with certain as-
sertions, e.g., the $→$ $\sqcup$ matches only with union
of concepts. This means that not all the rules will test
their applicability, reducing even further the number
of tests.
The improvement that one can achieve hugely depends on the KB itself. In the simplest case, the number of tests to perform are the same in the two versions. In this case, the expansion of the tableau implemented in the new version presents an overhead as the expansion queues must be kept updated. However, as the number of rule applications grows up, this overhead is expected to become negligible and the systems should become faster.

Example 6. Consider a slightly different version of the tableau and the KB of Examples 4 and 5 where the tableau contains only the assertion \( \{a : C_1\} \) and the KB contains a set of axioms \( C_i \subseteq C_{i+1} \) with \( i \) from 1 to \( k \). With the previous version of TRILL, the number of tests to perform in the worst case are \( n(1+\ldots+(k+1)) \) while the new version needs only \( n(k+1) \) tests. For \( k = 1 \) the difference is \( 3 : 2 \) that increases to \( 6 : 1 \) when \( k = 10 \).

5. Related Work

As far as the creation of the concepts hierarchy is concerned, many reasoners implement similar solutions to classify the KB named concepts. HermiT [44], for example, represents the hierarchy as a triple \( (V, H, \rho) \), where \( V \) is the set of nodes of the hierarchy, \( H \) is a set of pairs representing the subsumption connections between elements in \( V \) and \( \rho \) is a mapping between nodes in \( V \) and (set of) concepts and roles. This representation is similar to that implemented in the TRILL framework, where \( V \subseteq \mathbb{N} \cup \{n\} \), \( \rho \) can be seen as the classes map while \( H \) is a representation of the hierarchy tree. HermiT starts by adding the pairs representing obvious subsumptions in \( H \) and then puts the Nothing concept as the leaf of each branch. Then, as in the TRILL systems, it scans the graph for cycles in order to group equivalent classes and looks for unsatisfiable classes and/or disjoint concepts that are linked in the graph to update the mapping and the hierarchy.

The main difference with the TRILL framework is that HermiT considers the construction of the hierarchy as a separate task, called classification, while the TRILL systems use the hierarchy creation to collect information about the KB that can be used during the reasoning process. TRILL systems only perform structural checks and simple inferences to build the hierarchy and find possible unsatisfiable concepts without considering assertions but only subsumptions and disjunctions. On the other hand, HermiT performs many satisfiability tests on the nodes of the graph during the construction of the hierarchy. For these reasons, a direct comparison of the two methods would not be much meaningful. However, given the similarities of the two approaches, the TRILL systems can easily be extended to perform the classification task. However, this is left as a future work. Similarly to HermiT, state-of-the-art reasoners, such as FaCT++ [3] and Pellet [1], perform the classification task as a stand-alone inference process. The main differences between the implementations of this task in the different reasoners can be found in the optimizations implemented for reduc-
ing the calls to the reasoner to perform consistency and
subsumption tests. All these reasoners consider named
classes and only a restricted set of inferred subclass rel-
lations, while the TRILL systems consider a larger set
of possible inferred connections and both atomic and
complex concepts.

Similarly to the TRILL framework, Pellet also al-

dows to pre-compute the hierarchy to be used for an-
swering queries, however this setting is not enabled by
default and, as far as we know\(^2\), it executes the clas-
sification task and keeps in memory the information

collected.

Other systems, such as Konclude, exploit saturation
graphs in order to reduce the number of executions
of expansion rules [45]. A saturation graph is a graph
where the nodes are labelled with concepts and edges
with roles from the KB. The saturation process satu-
rates this graph by adding concepts and roles to the la-

ebels of nodes and edges. It is initialized with one or
more nodes labelled with the concepts that must be satu-
rated. Then, following the sub-class axioms defined
in the KB, every node is saturated in order to include
all the concepts subsumed by those added in the initial-
ization. Once built, this graph can be used to speed up
the expansion of the tableau and possibly avoid apply-
ing as many expansion rules as possible to the nodes
of the tableau.

This approach and the optimization presented in this
paper follow a similar idea. TRILL builds a graph from
which, given a node (a concept), it is easy to collect all
the super classes of that concept by following the path
towards the \(\top\) concept. This set is similar to the label of
a node in the saturated graph of [45]. However, the two
approaches differ in the fact that the saturation pro-
cedure is not intended to build justifications but only
to solve Boolean query answering. In fact, every node
or edge is labelled with concepts and role and usually
the saturation graph contains only concepts useful for
solving the query. To build justifications, the rules used
to expand the saturation graph must be extended with
a tracing function. This extension will then affect the
definition of the tableau expansion rules as the tracing
function of these rules must consider what contained in
the saturation graph making the combination not triv-
ial. By using a graph representing the hierarchy of the
concepts we can obtain a simplification of the applica-
tion of the tableau expansion rules similar to that ob-
tained in Konclude (given a concept it is easy to collect
all its super concepts that can be added to the label of
the individual in the tableau) but we can more easily
manage the construction of the justifications.

As for the optimization of the expansion queues,
the closest approach is the ToDo list implemented
in FaCT++ [3]. This approach creates one or more
queues, depending on the priority given to expansion
rules, which contain pairs (node, concept). Each time
a new label is added to the tableau, the corresponding
pair is added to the queues to be expanded. If there is a
single queue the pair is added at the end of the queue,
if there are more than one queue the pair is included
in the right queues according with the expansion rules
that can be applied to the new entry. The tableau algo-
rithm takes a pair from the queues following their pri-
ority and stops when all queues are empty. The priority
thus defines an order in the application of the rules that
in FaCT++ can be changed by the user. This means that
if there are three different queues, the rules associated
with the first queue are applied to the assertions in the
queue, then the rules of the second queue are applied
to the pairs of the second queue and so on. If a pair
is added to more than one queue, for example the first
and the third, its expansion will be performed in two
different moments, applying first the rules of the first
queue and then those of the third, but after the applica-
tion of all the other possible expansions from the first
and the second queues.

It is easy to see that the pairs (node, concept)
are equivalent to class assertions because a node in
the tableau represents an individual. Unlike FaCT++,
TRILL systems consider also assertions concerning
roles. As regards the management of the queues, simi-
larly to FaCT++, TRILL systems manage two queues.
However, the definition of these queues is fixed, and
the user cannot change their settings. Moreover, the
management of the assertions is slightly different be-
cause once an assertion is considered, every applica-
ble rule is executed following the order of application.
However, as seen in Section 4.2, assertions that may
match the condition of non-deterministic rules are in-
cluded in \textit{NonDetQueue} (that with “lower priority”) and
therefore considered later. This means that asser-
tions matching non-deterministic rules are considered
only when the \textit{DetQueue} is empty.

6. Experiments

To test the effectiveness of the optimizations intro-
duced in the new version of the TRILL framework,
we compared the systems TRILL, TRILL\(^p\), and TORNADO with their old versions. To facilitate the reader, in the following we refer to the new version using the word new, while the previous version will be indicated with old.

The experiments presented here are done by partly following the tests performed in [17], where old TRILL systems have been compared with the probabilistic reasoners BORN [46], BUNDLE [13, 47], and PRONTO [48], and with the non-probabilistic reasoners Pellet [1], Konclude\(^3\) [49], HermiT [2], FaCT++ [3], and JFact\(^4\). The results of these tests showed that a Prolog implementation of the tableau algorithm, and in particular the old TRILL systems, can achieve better results than state-of-the-art (non-) probabilistic reasoners. We refer the reader to [17] for an in-depth description of these results.

All the KBs and the scripts to reproduce the tests presented in this section can be found at https://github.com/rzese/docker-trill. A docker container is also available at https://hub.docker.com/r/rzese/trill.

Differently from [17], in this paper we consider only the different implementations of the TRILL framework. To facilitate the reader that wants to compare the results presented in the two articles, below we briefly discuss the similarities and differences with the tests conducted in [17]. To carry out the comparison among the two versions of the framework, we re-ran Tests 1, 2, 3, and 6 from [17], whose results are discussed in Test 1, Test 3, Test 4, and Test 5 respectively.

In particular, Test 1 is extended by also answering queries that have not any justification. Moreover, it considers also probabilistic query answering as in Test 4 of [17]. As proved in [17], the probability computation does not affect the inference performance, therefore, reporting the results of all the reasoners would result in reporting almost duplicated values for non-probabilistic and probabilistic query answering settings. Thus, for tests concerning probabilistic inference, we will show only the performance of TORNADO to prove that the implemented optimization does not affect the probability computation but only the search of the justifications.

We have also added a new test experimenting with the scalability of the old and new version of the TRILL framework. Therefore, in this paper we performed 5 tests considering both classical and probabilistic inference. Since the overhead added by the probability computation is slight, the results shown can be considered representative for both probabilistic and non-probabilistic settings.

All tests were run on the HPC System Galileo equipped with Intel Xeon E5-2697 v4 (Broadwell) @ 2.30 GHz.

### Test 1

We used four real-world KBs:

- BioPAX level \(^3\), which models metabolic pathways;
- BRCA\(^6\), which models the risk factors of breast cancer depending on many factors such as age and drugs taken;
- an extract of DBPedia ontology\(^7\), containing structured information of Wikipedia, usually those contained in the information box on the right-hand side of a page;
- Vicodi [50], which contains information on European history and models historical events and important personalities.

The details about the KBs are shown in Table 1. We randomly created 50 queries for each KB, ensuring that each query had at least one explanation. For each query, we asked TRILL, TRILL\(^p\), and TORNADO to find respectively the covering set of justifications, the complete pinpointing formula and the final BDD for the query and checking that it is not equivalent to the 0 Boolean value.

Table 2 shows the average CPU time in seconds to answer queries on each KB in the settings described above for the old version and the new version of the systems. “TORN.” column shows the running time of TORNADO.

The results show that the new version of the systems presents as expected a little overhead due to the use of the hierarchy and the use of the expansion queues, which is clearly visible when considering sim-

---

\(^3\)http://derivo.de/produkte/konclude/
\(^4\)http://jfact.sourceforge.net/
\(^5\)http://www.biopax.org/
\(^6\)http://www2.cs.man.ac.uk/~klinovp/pronto/brc/cancer_cc.owl
\(^7\)http://dbpedia.org/
We have also tested the performance of the reasoners when answering queries that have not justifications. In this case, the reasoner has to explore the entire search space similarly to the case of finding all the justifications. The average CPU time achieved by TRILL are reported in Table 3, the column “No Just.” contains the results for queries with no justifications, while the columns “1 Just.” and “All Just.” contain the times needed to return a single justification and all the justifications respectively. The values of the last column are taken from Table 2.

We compared the results obtained performing probabilistic query answering with those reported in Table 2.

We ran the two versions of TORNADO to compute the probability of the same queries generated for Test 6 to collect the running times. Table 4 shows the average CPU time in seconds taken by the two versions of the system for performing probabilistic inference (grey column) and non-probabilistic inference. As the results show, the extra time for the probability computation is negligible in this setting as due to a time measurement error in some cases the results of probabilistic query answering are even better than those of non-probabilistic ones. Thus, TORNADO can be used for Boolean query answering as well, without significant loss of performance. We have performed the same test to compare also TRILL and TRILL\(^p\), the results, omitted here for the sake of readability, showed that the same considerations hold for all the systems contained in the framework. Clearly, the optimization presented in this paper does not affect the computation of the probability, as also proved by these results.

### Test 2
To further test the TRILL framework on real world KBs, we have conducted a test using the Foundational Model of Anatomy Ontology (FMA for short)\(^6\). FMA is a KB for biomedical informatics that models the phenotypic structure of the human body anatomy. To perform this test, we created 12 versions of the KB containing an increasing number of individuals. The KBs contain 4,706 axioms in the TBox and RBox, with 2626 different classes. We then added

\(^6\)Edge is an algorithm for parameter learning from a set of positive and negative examples.
a number of individuals varying from 10 to 100 with a step of 10, 200 and 300, each of them described with 1 to 11 assertions. We ran 10 queries w.r.t. each KB and collected the averaged CPU time, reported in Table 5. From the results, we can see that the overhead introduced by the management of the hierarchy may be detrimental when the TBox is significantly larger than the ABox. This is due to the fact that when the individuals are few, the time to expand them may be small even when the TBox is very large. However, when the number of individuals increases, their expansion in the tableau becomes expensive and the overhead introduced by the optimizations presented in this paper is balanced by a more efficient expansion.

This test clearly shows both the pros and cons of the implemented optimizations, indicating where this approach can be further optimized. Note that TRILL$^P$ and TORNADO applies less tableau expansion rules than TRILL, therefore in some cases, like for FMA, their new version benefit more from the optimizations than TRILL, as shown by the results of the last two lines of Table 5.

**Test 3** In this third test we consider the artificial KB shown in the following:

\[ C_{1,1} \sqsubseteq C_{1,2} \sqsubseteq \ldots \sqsubseteq C_{1,m} \sqsubseteq C_{n+1} \]

\[ C_{1,1} \sqsubseteq C_{2,2} \sqsubseteq \ldots \sqsubseteq C_{2,m} \sqsubseteq C_{n+1} \]

\[ C_{1,1} \sqsubseteq C_{3,2} \sqsubseteq \ldots \sqsubseteq C_{3,m} \sqsubseteq C_{n+1} \]

\[ \ldots \]

\[ C_{1,1} \sqsubseteq C_{m,2} \sqsubseteq \ldots \sqsubseteq C_{m,m} \sqsubseteq C_{n+1} \]

\[ a : C_{1,1} \]

with $m$ and $n$ varying to increase the number of axioms.

When applying the expansion rules, we have to cope with two different kind of non-determinism: “don’t care” and “don’t know”. The first is inherent to the choice of which rule on which assertion to apply. The name “don’t care” is due to the fact that if a rule is applicable in a tableau to a certain assertion, it will also be so in any tableau obtained by its expansion (considering different pairs assertion/rule). Differently, the second kind of non-determinism is introduced by non-deterministic rules, since a single tableau is expanded into a set of tableaux.

The KB considered in this test forces the creation of an increasing number of choice points in order to investigate the effect of the “don’t care” non-determinism in the choice of rules. Which is semantically less important than the “don’t know” but acquires importance in terms of performance of the system. Finding all the justifications for the query $Q = a : C_{n+1}$ forces to find $n$ justifications containing $n + 1$ axioms each: $n$ subclass-of axioms and 1 assertion axiom. Moreover, during the expansion of this KB the differences between the rule calls made by the old and the new versions is near to 0, as only one call of the $\rightarrow$ unfold rule on each assertion succeeds. In fact, the new assertion in the old version is added at the beginning of the list of assertions in the tableau of the TRILL systems and the $\rightarrow$ unfold rule is the third applied therefore the new version can cut a small number of rule tests.

We executed the query $Q$ 100 times to compute the average running time that each system in each version needs to compute all the justifications. We varied $m$ and $n$ between 1 to 7. Tables 6, 7, and 8 report the results for TRILL, TRILL$^P$, and TORNADO respectively. Columns correspond to $n$ while rows correspond to $m$.

It is clear from the results that a little overhead is present in the new version. As already said, this overhead is more visible when the KB is simple as in this case. The running time is almost always of the same order of magnitude for both versions while the construction of the hierarchy tends to make the new version of the reasoners slower because it has the larger impact on the run time. But this impact decreases with the increment of the size of the KB. This is clearly shown by the results of TRILL$^P$. When $m = 7$ the inference part of the process becomes the predominant one and the reduction in terms of running time achieved by means of
Table 5
Average CPU time (in seconds) for answering queries with TRILL, TRILLP, and TORNADO in their two versions, w.r.t. KBs of increasing size based on FMA (Test 2). The first two columns show respectively the number of individuals and the number of axioms contained in each KB.

<table>
<thead>
<tr>
<th>N. Ind</th>
<th>N. axioms</th>
<th>old</th>
<th>new</th>
<th>old</th>
<th>new</th>
<th>old</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4788</td>
<td>0.702</td>
<td>2.885</td>
<td>1.586</td>
<td>2.865</td>
<td>0.750</td>
<td>2.643</td>
</tr>
<tr>
<td>20</td>
<td>4830</td>
<td>1.587</td>
<td>5.055</td>
<td>2.679</td>
<td>4.851</td>
<td>1.656</td>
<td>4.677</td>
</tr>
<tr>
<td>30</td>
<td>4895</td>
<td>3.336</td>
<td>7.709</td>
<td>8.017</td>
<td>7.116</td>
<td>3.393</td>
<td>6.650</td>
</tr>
<tr>
<td>50</td>
<td>5020</td>
<td>11.159</td>
<td>14.862</td>
<td>16.912</td>
<td>15.044</td>
<td>11.450</td>
<td>10.021</td>
</tr>
<tr>
<td>60</td>
<td>5082</td>
<td>18.476</td>
<td>36.384</td>
<td>35.965</td>
<td>25.956</td>
<td>18.989</td>
<td>15.044</td>
</tr>
<tr>
<td>70</td>
<td>5124</td>
<td>24.639</td>
<td>40.037</td>
<td>35.965</td>
<td>25.956</td>
<td>18.989</td>
<td>15.044</td>
</tr>
<tr>
<td>80</td>
<td>5220</td>
<td>40.037</td>
<td>68.942</td>
<td>68.942</td>
<td>40.037</td>
<td>68.942</td>
<td>25.956</td>
</tr>
<tr>
<td>90</td>
<td>5244</td>
<td>68.942</td>
<td>83.986</td>
<td>83.986</td>
<td>40.037</td>
<td>68.942</td>
<td>25.956</td>
</tr>
<tr>
<td>100</td>
<td>5354</td>
<td>120.191</td>
<td>120.191</td>
<td>24.780</td>
<td>28.836</td>
<td>120.191</td>
<td>28.836</td>
</tr>
<tr>
<td>200</td>
<td>6008</td>
<td>35.965</td>
<td>72.455</td>
<td>72.455</td>
<td>15.508</td>
<td>15.508</td>
<td>35.965</td>
</tr>
<tr>
<td>300</td>
<td>6650</td>
<td>1785.792</td>
<td>1785.792</td>
<td>36.384</td>
<td>72.455</td>
<td>36.384</td>
<td>15.508</td>
</tr>
</tbody>
</table>

Table 6
Average CPU time (in seconds) for computing all the explanations with the reasoner TRILL in its two versions in Test 3. Columns correspond to \( n \) while rows correspond to \( m \). \( n \) and \( m \) vary from 1 to 7 in step of 1. In bold the best time for each size.

<table>
<thead>
<tr>
<th>TRILL</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>old</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0004</td>
<td>0.0004</td>
<td>0.0004</td>
<td>0.0005</td>
</tr>
<tr>
<td>new</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0004</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0007</td>
</tr>
<tr>
<td>old</td>
<td>0.0004</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0007</td>
</tr>
<tr>
<td>new</td>
<td>0.0004</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0005</td>
<td>0.0007</td>
</tr>
<tr>
<td>old</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0009</td>
</tr>
<tr>
<td>new</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0006</td>
<td>0.0009</td>
</tr>
<tr>
<td>old</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0008</td>
</tr>
<tr>
<td>new</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0007</td>
<td>0.0009</td>
</tr>
<tr>
<td>old</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0009</td>
</tr>
<tr>
<td>new</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0008</td>
<td>0.0009</td>
</tr>
<tr>
<td>old</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0009</td>
</tr>
<tr>
<td>new</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0009</td>
<td>0.0009</td>
</tr>
<tr>
<td>old</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.0010</td>
</tr>
<tr>
<td>new</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.0010</td>
<td>0.0010</td>
</tr>
<tr>
<td>old</td>
<td>0.0011</td>
<td>0.0011</td>
<td>0.0011</td>
<td>0.0011</td>
<td>0.0011</td>
<td>0.0011</td>
<td>0.0011</td>
</tr>
<tr>
<td>new</td>
<td>0.0011</td>
<td>0.0011</td>
<td>0.0011</td>
<td>0.0011</td>
<td>0.0011</td>
<td>0.0011</td>
<td>0.0011</td>
</tr>
</tbody>
</table>

The implemented optimizations becomes more effective as the initial overhead will be increasingly dominated by the inference part of the process.

To further test how the two versions perform with larger KBs we extended this test by increasing the values of \( m \) and \( n \), varying them from 10 to 100 in step of 10. We set a timeout of 95 minutes. The cells with the symbol “–” indicate that the timeout on the old system occurred while italic values show the ratio when the old system exceeded 30 minutes. The new systems never exceeded 30 minutes of computation. Tables 9, 10, and 11 show the ratio old/new for TRILL, TRILLP, and TORNADO for increasing \( m \) and \( n \).

As expected, the overhead in this test is significant. However, the larger the KB the higher the ratio. It is important to notice that the more difficult the application of the rule, the higher the ratio. In other words, if we consider TORNADO that is the fastest system, the application of the \( \rightarrow unfold \) rule is almost instantaneous. In this case the new version is always slower in this test with the highest running time around 300 seconds for the larger KB. However, from Table 11 it
is possible to see that the ratio becomes higher and higher (up to 0.843) as the size of the KB increases. Regarding TRILL, which is the second fastest system, there are cases where the ratio is higher than 1. With \( n = m = 100 \) the running time was few higher than 300 seconds showing, as expected, that the overhead is more significant when the rule is simple to apply.

Results on TRILL\(^P\) confirm this theory. In fact, rule application in this system is more difficult than in the other systems because it relies on a SAT solver to test the applicability of the rule. In this case the reduced number of tests impacts significantly on the running time. For example, the new version reaches 30 minutes of computation only for \( n = m = 100 \) while the old version reaches this time with \( n = 30, m = 60 \). With this KB the new version’s average running time is lower than 42 seconds. The timeout occurs for the old version with \( n = 60, m = 90 \). With this KB the new version’s average running time is lower than 495 seconds.
Table 9

Ratio old/new for TRILL in Test 3. Columns correspond to $n$ while rows correspond to $m$ and $m$ vary from 10 to 100 in step of 10.

<table>
<thead>
<tr>
<th>TRILL</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.374</td>
<td>0.416</td>
<td>0.393</td>
<td>0.473</td>
<td>0.443</td>
<td>0.460</td>
<td>0.529</td>
<td>0.592</td>
<td>0.567</td>
<td>0.565</td>
</tr>
<tr>
<td></td>
<td>0.393</td>
<td>0.415</td>
<td>0.457</td>
<td>0.465</td>
<td>0.486</td>
<td>0.526</td>
<td>0.585</td>
<td>0.549</td>
<td>0.550</td>
<td>0.581</td>
</tr>
<tr>
<td></td>
<td>0.454</td>
<td>0.484</td>
<td>0.484</td>
<td>0.533</td>
<td>0.615</td>
<td>0.558</td>
<td>0.595</td>
<td>0.633</td>
<td>0.689</td>
<td>0.758</td>
</tr>
<tr>
<td></td>
<td>0.470</td>
<td>0.501</td>
<td>0.594</td>
<td>0.552</td>
<td>0.593</td>
<td>0.662</td>
<td>0.707</td>
<td>0.628</td>
<td>0.657</td>
<td>0.684</td>
</tr>
<tr>
<td></td>
<td>0.520</td>
<td>0.571</td>
<td>0.662</td>
<td>0.632</td>
<td>0.712</td>
<td>0.743</td>
<td>0.656</td>
<td>0.698</td>
<td>0.743</td>
<td>0.797</td>
</tr>
<tr>
<td></td>
<td>0.571</td>
<td>0.622</td>
<td>0.607</td>
<td>0.692</td>
<td>0.729</td>
<td>0.742</td>
<td>0.782</td>
<td>0.825</td>
<td>0.838</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.591</td>
<td>0.661</td>
<td>0.675</td>
<td>0.744</td>
<td>0.688</td>
<td>0.746</td>
<td>0.769</td>
<td>0.820</td>
<td>0.705</td>
<td>0.767</td>
</tr>
<tr>
<td></td>
<td>0.558</td>
<td>0.626</td>
<td>0.711</td>
<td>0.678</td>
<td>0.768</td>
<td>0.788</td>
<td>0.881</td>
<td>0.785</td>
<td>0.849</td>
<td>0.897</td>
</tr>
<tr>
<td></td>
<td>0.634</td>
<td>0.703</td>
<td>0.806</td>
<td>0.773</td>
<td>0.822</td>
<td>0.925</td>
<td>0.865</td>
<td>0.924</td>
<td>0.970</td>
<td>0.977</td>
</tr>
<tr>
<td></td>
<td>0.696</td>
<td>0.798</td>
<td>0.924</td>
<td>0.885</td>
<td>0.957</td>
<td>1.028</td>
<td>0.927</td>
<td>0.965</td>
<td>0.977</td>
<td>1.053</td>
</tr>
</tbody>
</table>

Table 10

Ratio old/new for TRILL in Test 3. Columns correspond to $n$ while rows correspond to $m$ and $m$ vary from 10 to 100 in step of 10.

<table>
<thead>
<tr>
<th>TRILL</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.881</td>
<td>2.663</td>
<td>2.718</td>
<td>2.755</td>
<td>2.784</td>
<td>2.852</td>
<td>2.860</td>
<td>2.853</td>
<td>2.803</td>
<td>2.906</td>
</tr>
<tr>
<td></td>
<td>17.417</td>
<td>17.187</td>
<td>17.317</td>
<td>17.103</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>18.331</td>
<td>18.485</td>
<td>19.909</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>21.290</td>
<td>22.037</td>
<td>23.163</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 11

Ratio old/new for TORNADO (TORN.) in Test 3. Columns correspond to $n$ while rows correspond to $m$ and $m$ vary from 10 to 100, step 10.

<table>
<thead>
<tr>
<th>TORN.</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.342</td>
<td>0.326</td>
<td>0.270</td>
<td>0.310</td>
<td>0.273</td>
<td>0.263</td>
<td>0.282</td>
<td>0.311</td>
<td>0.299</td>
<td>0.276</td>
</tr>
<tr>
<td></td>
<td>0.361</td>
<td>0.340</td>
<td>0.333</td>
<td>0.332</td>
<td>0.313</td>
<td>0.327</td>
<td>0.350</td>
<td>0.323</td>
<td>0.306</td>
<td>0.299</td>
</tr>
<tr>
<td></td>
<td>0.408</td>
<td>0.395</td>
<td>0.368</td>
<td>0.384</td>
<td>0.424</td>
<td>0.363</td>
<td>0.377</td>
<td>0.379</td>
<td>0.402</td>
<td>0.425</td>
</tr>
<tr>
<td></td>
<td>0.426</td>
<td>0.421</td>
<td>0.464</td>
<td>0.411</td>
<td>0.427</td>
<td>0.458</td>
<td>0.476</td>
<td>0.403</td>
<td>0.411</td>
<td>0.412</td>
</tr>
<tr>
<td></td>
<td>0.494</td>
<td>0.489</td>
<td>0.545</td>
<td>0.479</td>
<td>0.530</td>
<td>0.547</td>
<td>0.458</td>
<td>0.471</td>
<td>0.508</td>
<td>0.522</td>
</tr>
<tr>
<td></td>
<td>0.568</td>
<td>0.572</td>
<td>0.515</td>
<td>0.572</td>
<td>0.580</td>
<td>0.512</td>
<td>0.539</td>
<td>0.547</td>
<td>0.558</td>
<td>0.560</td>
</tr>
<tr>
<td></td>
<td>0.583</td>
<td>0.603</td>
<td>0.593</td>
<td>0.610</td>
<td>0.541</td>
<td>0.578</td>
<td>0.584</td>
<td>0.600</td>
<td>0.494</td>
<td>0.531</td>
</tr>
<tr>
<td></td>
<td>0.553</td>
<td>0.560</td>
<td>0.627</td>
<td>0.565</td>
<td>0.602</td>
<td>0.610</td>
<td>0.671</td>
<td>0.575</td>
<td>0.618</td>
<td>0.633</td>
</tr>
<tr>
<td></td>
<td>0.643</td>
<td>0.657</td>
<td>0.717</td>
<td>0.670</td>
<td>0.669</td>
<td>0.731</td>
<td>0.653</td>
<td>0.710</td>
<td>0.721</td>
<td>0.716</td>
</tr>
<tr>
<td></td>
<td>0.709</td>
<td>0.749</td>
<td>0.824</td>
<td>0.754</td>
<td>0.795</td>
<td>0.843</td>
<td>0.749</td>
<td>0.742</td>
<td>0.733</td>
<td>0.779</td>
</tr>
</tbody>
</table>
Test 4  This test stresses even further the management of the backtracking by considering artificial KBs of increasing size where, despite the increment of the size is linear in the number of axioms, the number of justifications increases exponentially. Such KBs are of the form

\( (C_{1,i}, Q_i, B_i) \)

with \( 1 \leq i \leq n \) an integer, \( n \geq 1 \). The query \( Q = \bigwedge_{i \in [1,n]} P_i \cap Q_i \)

For \( n = 2 \) for example, we have 4 different explanations, namely

\( \{C_{1,1}, C_{2,1}, C_{1,2}, C_{2,2}\} \)
\( \{C_{1,1}, C_{3,1}, C_{1,2}, C_{2,2}\} \)
\( \{C_{1,1}, C_{2,1}, C_{1,2}, C_{3,2}\} \)
\( \{C_{1,1}, C_{3,1}, C_{1,2}, C_{3,2}\} \)

The corresponding pinpointing formula is \( C_{1,1} \land (C_{2,1} \lor C_{3,1}) \land C_{1,2} \land (C_{2,2} \lor C_{3,2}) \). In general, given \( n \), the formula for this example is

\[
\bigwedge_{i \in [1,n]} C_{1,i} \land \bigwedge_{j \in [1,n]} \bigvee_{z \in \{2,3\}} C_{z,j}
\]

whose size is linear in \( n \).

The value of \( n \) was increased from 2 to 10 in steps of 2. We performed the query \( Q \) 50 times for each KB to compute the average running time, shown in Table 12. The cells with “–” indicate that the inference time exceeded 4 hours and 30 minutes.

Results show that the new version of the TRILL framework outperforms the old version. These results clearly show that the optimizations implemented in the systems can sensibly improve the performances.

Test 5  The last test was performed by considering versions of BRCA of increasing size generated following the method presented in [52]. In that paper, the authors generated KBs by starting from the original (probabilistic) TBox and adding a number of (probabilistic) sub-class axioms varying from 10 to 15. The choice of the axioms to add were made following a metrics they defined in the paper, called connectivity, indicating the number of relations among concepts, e.g., subsumption and disjointness. Following this metric, each axiom added to the original KB were chosen in order to significantly increase the complexity of the reasoning w.r.t. the created KBs. By following this method, we created KBs of increasing size where the number of the added probabilistic axioms was varied from 9 to 26, and, for each number, 100 different consistent ontologies were created. After the addition of the subclass-of axioms, for each KB an individual were added and randomly assigned to each simple class that appears in the added axioms with probability 0.6, where complex classes were split into their components, e.g., the complex class 

\[
\text{PostmenopausalWomanTakingTestosterone}\]

was divided into 

\[
\text{PostmenopausalWoman} \quad \text{and} \quad \text{WomanTakingTestosterone}.
\]

Finally, for each KB we ran 100 probabilistic queries of the form \( \alpha : C \) where \( \alpha \) is the added individual and \( C \) is a class randomly selected among those that represent women under increased and lifetime risk such as 

\[
\text{WomanUnderLifetimeBRCRisk} \quad \text{and} \quad \text{WomanUnderStronglyIncreasedBRCRisk}.
\]

Table 13 shows the execution time averaged over the 100 probabilistic queries with the varying of the number of probabilistic axioms (“NPA” column) for the reasoners TRILL, TRILL\(^p\), and TORNADO respectively. Moreover, the ratio old/new is shown for each system and each size of the KBs. The averaged ratio ± standard deviation is also shown for each system.

Overall, the new version of the systems always outperforms the old one. In particular, the system that is the most affected by the optimizations is TRILL\(^p\), as shown also by the previous tests, followed by TORNADO, which demonstrates to be always the best system among those included in the TRILL framework.
Average CPU time (in seconds) for computing the probability of queries with the reasoners TRILL, trillp, and TORNADO in their two versions in Test 5. The column “ratio” shows the ratio old/new for the execution time. Their average ± the standard deviation is also shown.

<table>
<thead>
<tr>
<th>Res.</th>
<th>NPA</th>
<th>old</th>
<th>new</th>
<th>ratio</th>
<th>Res.</th>
<th>old</th>
<th>new</th>
<th>ratio</th>
<th>Res.</th>
<th>old</th>
<th>new</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>30.508</td>
<td>1.875</td>
<td>16.3</td>
<td></td>
<td>260.925</td>
<td>2.078</td>
<td>125.5</td>
<td></td>
<td>1.204</td>
<td>0.071</td>
<td>17.0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>27.114</td>
<td>1.310</td>
<td>20.7</td>
<td></td>
<td>237.148</td>
<td>2.087</td>
<td>113.7</td>
<td></td>
<td>1.044</td>
<td>0.067</td>
<td>15.6</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>83.032</td>
<td>31.333</td>
<td>2.7</td>
<td></td>
<td>311.544</td>
<td>7.801</td>
<td>39.9</td>
<td></td>
<td>1.515</td>
<td>0.085</td>
<td>17.8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>78.865</td>
<td>31.277</td>
<td>2.5</td>
<td></td>
<td>352.997</td>
<td>6.260</td>
<td>56.1</td>
<td></td>
<td>1.573</td>
<td>0.086</td>
<td>18.2</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>74.402</td>
<td>10.889</td>
<td>6.8</td>
<td></td>
<td>341.674</td>
<td>5.188</td>
<td>65.9</td>
<td></td>
<td>1.604</td>
<td>0.087</td>
<td>18.4</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>107.875</td>
<td>31.080</td>
<td>3.5</td>
<td></td>
<td>321.559</td>
<td>8.260</td>
<td>38.9</td>
<td></td>
<td>1.624</td>
<td>0.084</td>
<td>19.4</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>99.876</td>
<td>18.617</td>
<td>5.4</td>
<td></td>
<td>350.364</td>
<td>7.020</td>
<td>49.9</td>
<td></td>
<td>1.645</td>
<td>0.088</td>
<td>18.7</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>70.720</td>
<td>16.261</td>
<td>4.3</td>
<td></td>
<td>339.781</td>
<td>6.261</td>
<td>54.3</td>
<td></td>
<td>1.632</td>
<td>0.085</td>
<td>19.1</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>174.825</td>
<td>73.561</td>
<td>2.4</td>
<td></td>
<td>461.894</td>
<td>27.983</td>
<td>16.5</td>
<td></td>
<td>2.126</td>
<td>0.109</td>
<td>19.5</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>161.380</td>
<td>65.569</td>
<td>2.5</td>
<td></td>
<td>342.495</td>
<td>24.239</td>
<td>14.1</td>
<td></td>
<td>1.896</td>
<td>0.101</td>
<td>18.8</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>156.791</td>
<td>65.263</td>
<td>2.4</td>
<td></td>
<td>436.081</td>
<td>36.541</td>
<td>11.9</td>
<td></td>
<td>2.040</td>
<td>0.103</td>
<td>19.8</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>214.143</td>
<td>63.378</td>
<td>3.4</td>
<td></td>
<td>464.981</td>
<td>31.602</td>
<td>14.7</td>
<td></td>
<td>2.237</td>
<td>0.113</td>
<td>19.9</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>159.106</td>
<td>65.987</td>
<td>2.4</td>
<td></td>
<td>455.583</td>
<td>27.010</td>
<td>16.9</td>
<td></td>
<td>2.116</td>
<td>0.107</td>
<td>19.7</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>166.372</td>
<td>66.271</td>
<td>2.5</td>
<td></td>
<td>404.276</td>
<td>24.321</td>
<td>16.6</td>
<td></td>
<td>1.929</td>
<td>0.100</td>
<td>19.2</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>225.816</td>
<td>92.194</td>
<td>2.4</td>
<td></td>
<td>469.687</td>
<td>56.111</td>
<td>8.4</td>
<td></td>
<td>2.210</td>
<td>0.114</td>
<td>19.4</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>234.661</td>
<td>123.920</td>
<td>1.9</td>
<td></td>
<td>457.938</td>
<td>91.312</td>
<td>5.0</td>
<td></td>
<td>2.161</td>
<td>0.111</td>
<td>19.4</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>176.237</td>
<td>66.980</td>
<td>2.6</td>
<td></td>
<td>455.783</td>
<td>48.476</td>
<td>9.4</td>
<td></td>
<td>2.118</td>
<td>0.108</td>
<td>19.7</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>295.107</td>
<td>194.457</td>
<td>1.5</td>
<td></td>
<td>570.789</td>
<td>128.392</td>
<td>4.4</td>
<td></td>
<td>2.812</td>
<td>0.136</td>
<td>20.7</td>
<td></td>
</tr>
</tbody>
</table>

Avg. 4.8 | Avg. 36.8 | Avg. 18.9 | ± 5.20 | ± 35.93 | ± 1.19

7. Conclusions and Future Directions

In this paper we presented two extensions implemented in the TRILL framework, which contains three systems for reasoning on DISPONTE KBs: TRILL, able to collect the set of all the justifications and compute the probability of queries, TRILL\(^p\), which implements in Prolog the tableau algorithm defined in [15, 16] for returning the pinpointing formula instead of the set of justifications, and TORNADO, which is similar to TRILL\(^p\) but instead of building a pinpointing formula and translating it to a BDD in two different phases, it builds the BDD while building the tableau.

The first extension allows the reasoners to collect information about the KB and build the hierarchy of the concepts in order to quickly find connections between them during the expansion of the tableau. The second optimization avoids testing assertions that have not been changed since the last check during the expansion of the tableau, in order to not perform useless tests on assertions.

The extensive experimentation performed shows that the presented optimizations can significantly speed up both regular and probabilistic queries in many cases. The results show that in exchange for a small overhead, visible especially w.r.t. simpler KBs or when the time needed for the inference is under one second, the optimizations allows to achieve a speed up of up to more than 125:1.

We are also studying the possibility of selectively expand the tableau when possible. In fact, given a query involving an individual, for example an instance-of query, the justifications only depend on individuals connected to those specified by the query. Thus, the idea is to collect the set of all the individuals that are connected directly or indirectly to those of the query and expand only these individuals. This should allow TRILL systems to scale even further, as the resolution of the query would consider only a subset of the axioms of the considered KB. Another interesting direction would be made TRILL systems incremental and following a more pay-as-you-go model.

The optimizations presented here, and these future directions are aimed at the realization of a Semantic Web framework where several reasoners are made easily interchangeable because all bundled in a single rea-
soner, BUNDLE [12]. The reasons are used to collect justifications while BUNDLE oversees the computation of the probability of the query from them.

At the moment, TRILL is included only partly, while FaCT++, JFact, Pellet, HermiT can be used to collect justifications. Other reasoners following the OWL-API can be easily added while we are also studying the possibility to include abductive frameworks, such as [38, 39] which translate the KB into Datalog± or by implementing the abductive tableaux presented in [53]. The framework will also contain a crawler for linked data, called KRaider [54], able to follow links in the linked data cloud and automatically collect RDF triples.

Moreover, as already said before, the order of the expansion rules in TRILL is fixed. However, it is easy to implement the possibility to set a different order. This could be another possible future work.

References


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


Appendix A. Comparison of the implementations

In this section we show how the optimizations described in this paper are implemented in TRILL, comparing the code of the two versions of the framework. We will show a comparison between the implementation of the tableau expansion rules and how the systems implement the expansion of the tableau.

A.1. Non-deterministic Rules

As seen in Section 3, in the old version non-deterministic rules are implemented following the interface rule_name(\text{TabIn}, \text{TabList}), while, in the new version, as seen in Section 4.2, they follow the interface rule_name(\text{TabIn}, \text{AxToExpand}, \text{TabList}). Figure 8 shows the code of the non-deterministic rule → ∪ contained in the old version, in the upper frame, and in the new version, in the lower frame, of the framework. The predicate or_rule searches in the tableau TabIn = (A0,T0) for an individual to which the rule can be applied and unifies L with the list of new tableaux created by scan_or_list. Figure 9 shows the code of scan_or_list/8 implemented in the old version of the TRILL systems. The predicate findClassAssertion/4 implements the search for a class assertion in A. The difference among the two versions is in the fact that in the old implementation the predicate findClassAssertion/4 looks for any assertion matching the concept unionOf, while in the new implementation it looks for the specific assertion passed to the rule.

The predicates get_choice_point_id/1, add_choice_point/3 and create_choice_point/4 are used to correctly build the justifications in case of non-deterministic rules. Basically, when a non-deterministic rule is applied, more than one new tableaux are created. In this case, there can be justifications that do not depend on the application of non-deterministic rules (i.e., justifications that can be found in every tableau) and therefore that can be directly turned by the algorithm. On the other hand, there are justifications that depend on non-deterministic rules. In this case, there could be a different clash in each tableau generated by the non-deterministic rule, therefore the explanations built during the inference process must consider also the choice points occurred during their construction. For a detailed description of these aspects we refer to [55].

Figure 9 shows the code of the scan_or_list/8 predicate. It calls modify_ABox/5, which checks if the class assertion axiom with the associated explanation is already present in A0. The variable A0 represents thus the ABox and corresponds with the list of labels of the completion graph, it is the list of assertions contained in the tableau mapping each assertion with the value of its tracing function. If the assertion is already present in the ABox, the predicate modify_ABox/5 checks the applicability of the expansion rule. A rule can be applied if the tracing func-
or_rule((A0,T0),TabList):-
  findClassAssertion(unionOf(LC),Ind,Expl,A0),
  
+ indirectly_blocked(Ind,(A0,T0)),
  get_choice_point_id(ID),
  scan_or_list(LC,0,ID,Ind,Expl,A0,T0,TabList),
  dif(L,[]),
  create_choice_point(Ind,or,unionOf(LC),LC),!.

or_rule((A0,T)-ExpansionPairQ0,(unionOf(LC),Ind),TabList):-
  findClassAssertion(unionOf(LC),Ind,Expl,A),
  
+ indirectly_blocked(Ind,(A,T)),
  get_choice_point_id(ID),
  scan_or_list(LC,0,ID,Ind,Expl,A0,T,ExpansionPairQ0,TabList),
  dif(L,[]),
  create_choice_point(M,Ind,or,unionOf(LC),LC),!.

Fig. 8. Code of the \(\to \sqcup\) rule in its old version (top frame) and new version (bottom frame). It unifies the list \(L\) with all the tableaux resulting by the application of the rule. The predicate \(\text{scan\_or\_list}/8\) called in the upper frame, shown in Figure 9, creates the list of the tableaux or fails. Similarly, the predicate \(\text{scan\_or\_list}/9\) also updated the expansion queues.

A.2. Deterministic Rules

In the old version of the TRILL framework, deterministic rules are defined following the interface \(\text{rule\_name}(\text{TabIn}, \text{TabOut})\) that, given the current tableau \(\text{TabIn}\), returns the tableau \(\text{TabOut}\) obtained by the application of the rule to \(\text{TabIn}\). In the new version, the rules also take the assertion to expand. Figure 10 shows part of the code of the deterministic rule \(\to \text{unfold}\). As before, in the top frame the code of the original implementation is shown while in the bottom frame the code of the original implementation is shown while the new implementation is shown in the bottom frame. The predicate \(\text{unfold\_rule}/2\) (top) searches in \(\text{TabIn} = (A0,T)\) for an individual to which the rule can be applied. On the other hand, the predicate \(\text{unfold\_rule}/3\) (bottom) extracts exactly the assertion taken in input. Then the predicate \(\text{find\_sub\_sup\_class}/3\) is called in order to find the class to be added to the label of the individual. In this part of the code it is possible to note how the hierarchy created during the KB loading is used to easily and quickly apply the rule. In the original version, the search for a super-class \(D\) is done by looking for one possible axiom in the KB. This axiom is then added to the original explanation to update the value of the trac-
Fig. 9. The predicate `scan_or_list/8` as implemented in the old version of the framework. It adds each concept \( C \) contained in the list \( \text{LC} \) with the explanation \( \text{Expl} \) to the tableau \( A_0 \) creating a new tableau for each concept in the conjunction. The new version differs only in the fact that it takes the pair of expansion queues and gives it in input to the `modify_ABox` predicate.

```prolog
scan_or_list([],_,_,_,_,_,_,[]):- !.
scan_or_list([C|CT],N0,CP,Ind,Expl0,A0,T0,[(A,T0)|L]):-
    add_choice_point(cpp(CP,N0),Expl0,Expl),
    modify_ABox(A0,C,Ind,Expl,A),
    N is N0 + 1,
    scan_or_list(CT,N,CP,Ind,Expl0,A0,T0,L).
```

Fig. 10. Code of the \( \rightarrow \text{unfold} \) rule implemented in the old (top) and new (bottom) version of the systems. Given a class \( C \) from the input tableau, it looks for a class \( D \) which is a super-class or an equivalent class of \( C \). It builds the explanation \( (\text{AxL}) \) for the new class assertion found (by using \( \text{and}_f/3 \)) and adds it to the tableau to update it. The predicate \( \text{and}_f/3 \) performs a conjunction of two explanations.

```prolog
unfold_rule((A0,T),{(A,T)}):-
    findClassAssertion(C,Ind,Expl0,A0),
    find_sub_sup_class(C,D,Ax),
    and_f(Ax,Expl0,Expl),
    modify_ABox(A0,D,Ind,Expl,A).

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

find_sub_sup_class(C,D,Ex):-
    hierarchy(H),
    get_subclass(C,H,D),
    get_subclass_explanation(C,D,H,Ex).
```

The function by means of \( \text{and}_f/3 \) predicate, which conjoins the explanation \( \text{Exp}10 \) with the axiom \( \text{Ax} \). In the new version, instead of looking for axioms in the KB, the system queries the hierarchy for collecting the information needed. In this case, if there are more axioms connecting concepts \( C \) and \( D \), they are all returned at the same time. In this case, the predicate \( \text{and}_f/3 \) conjoins the explanation \( \text{Exp}10 \) with possibly a set of explanations, reducing the number of applications of the rule on the considered assertion. Moreover, it is also possible to note that the `modify_ABox` predicate now has two more arguments, as already discussed in Section A.1.

A.3. Application of the expansion rules

Expansion rules are applied in order by the procedure `APPLY_ALL_RULES`, first the deterministic ones.
and then the non-deterministic ones, as shown in Figure 3.

Figure 11 shows the Prolog code of the corresponding predicate apply_all_rules/2, which calls apply_det_rules/3 to expand the tableau. The predicate apply_det_rules/3 takes as input the list of deterministic rules and the current tableau and returns a new tableau obtained by the application of one of the rules or the same tableau if no more rules can be applied. After the application of a deterministic rule, a cut avoids backtracking to other possible choices for the deterministic rules.

If no more deterministic rules can be applied, non-deterministic rules are tried sequentially with the predicate apply_nondet_rules/3, whose implementation is shown in Figure 11. It takes as input the list of non-deterministic rules and the current tableau and returns a tableau obtained with the application of one of the rules. 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 rules are applicable, the input tableau is returned and the rule application stops, otherwise a new round of rule application is performed. Finally, the findall/3 predicate is used on the set of the built tableaux for finding all the clashes contained in them in order to collect all the possible explanations.

In the new version, the expansion of the tableaux is guided by the procedure EXPAND_QUEUES, which is implemented by the predicate expand_queues/2 shown in Figure 12.

As seen in Section 4.2, every tableau has its own pair of expansion queues depending on the rules applied on it. In the new version of the TRILL framework, the tableau is represented as a pair containing the tableau (the same used in the old version) and the structure containing the two expansion queues, therefore TabExIn of Figure 7 is represented by Tab-ExpansionPairQ. This representation allows to easily manage the tableau, avoiding the need of implementing procedures such as GET_EXPANSION_PAIRQ and REPLACE_EXPANSION_PAIRQ used in Figure 7 (line 2 and line 8).

Therefore, the predicate expand_queues/2 (corresponding with procedure EXPAND_QUEUES of Figure 7) replaces the predicate apply_all_rules/2 of Figure 11, extracts the next assertion that must be expanded and calls apply_all_rules/3. The predicate expand_queues/2 stops when there are no more assertions to expand. The implementation of apply_all_rules/3 is very similar to that of apply_all_rules/2. The new argument taken as input is the assertion AssToExpand which is passed to the expansion rules which are able to manage it. The first definition of the predicate, i.e., expand_queues(Tab-[[],[ ]], Tab), is considered only when both the expansion queues are empty, thus the tableau is fully expanded. In this case, the expansion queues are no longer associated with the tableau, which is returned in order to collect the justifications.

The pair of expansion queues ExpansionPairQ is implemented in ExpansionPairQ as a list containing two lists initialized together with the initial tableau. The first list contains every assertion contained in the tableau not triggering non-deterministic rules, while the second contains assertion triggering non-deterministic rules, i.e., conjunctions of concepts (⊔) and maximum cardinality restrictions (⩽).

As seen in Figure 7 and then in Figure 12, the first operation of expand_queues/2 is to extract from ExpansionPairQ the next assertion to expand.

The extraction is performed by using the predicate extract_from_expansion_queues/3 shown in Figure 13, which implements the lines 3-6 of the algorithm defined in Figure 7. This predicate takes a pair of expansion queues, removes from one of them the next assertion to expand and returns this assertion and the new pair of expansion queues resulting from the removal of the assertion. First, are extracted assertions triggering only deterministic rules, i.e., from the first list, then those triggering non-deterministic rules, i.e., those in the second list.

Appendix B. Implementation of hierarchy management

In the TRILL systems, the map containing the information about the KB is implemented using the dictionaries of SWI-Prolog, while the management of the hierarchy graph in this map is done by means of the digraphs library of SWI-Prolog.

As seen in Section 4.1, after the initialization of the hierarchy, each concept of the KB is added to the hierarchy by means of the predicate add_class/3, corresponding with the procedure ADD_CLASS (Figure 5). The code of add_class/3 is shown in Figure 14.
apply_all_rules(TabIn,TabOut):-
  setting_trill(det_rules,Rules),
  apply_det_rules(Rules,TabIn,TabTemp),
  (same_ABox(TabIn,TabTemp) ->
    TabOut=TabTemp
  ;
    apply_all_rules(TabTemp,TabOut)
 ).
apply_det_rules([],TabIn,TabOut):-
  setting_trill(nondet_rules,Rules),
  apply_nondet_rules(Rules,TabIn,TabOut).
apply_det_rules([H|_],TabIn,TabOut):-
  call(H,TabIn,TabOut),!.
apply_det_rules([_|T],TabIn,TabOut):-
  apply_det_rules(T,TabIn,TabOut).
apply_nondet_rules([],Tab,Tab).
apply_nondet_rules([H|_],TabIn,TabOut):-
  call(H,TabIn,L),!,
  member(TabOut,L),
  dif(TabIn,TabOut).
apply_nondet_rules([_|T],TabIn,TabOut):-
  apply_nondet_rules(T,TabIn,TabOut).

Fig. 11. Application of the expansion rules by means of the predicates apply_all_rules/2, apply_det_rules/3 and apply_nondet_rules/3. The list Rules contains the available rules and is different in TRILL, TRILL$^p$ and TORNADO.

expand_queues(Tab=([],[]),Tab).
expand_queues(Tab=ExpansionPairQ,TabExOut):-
  extract_from_expansion_queues(ExpansionPairQ,AssToExpand,NewExpansionQueue),!,
  apply_all_rules(Tab-NewExpansionQueue,AssToExpand,TabTemp),
  expand_queues(TabTemp,TabExOut).

Fig. 12. Application of the expansion rules by means of the predicate expand_queues/2, which in turns calls apply_all_rules/3.

extract_from_expansion_queues([],[],[[]],T).
extract_from_expansion_queues([EA|T],NonDetQ),EA,[T,NonDetQ]).

Fig. 13. Code of the extract_from_expansion_queues/3 predicate.
add_class(HierIn, Class, HierOut):-
Classes0 = HierIn.classes,
\+ _ = Classes0.find(Class),
ClassesN = HierIn.classesName,
IDClass = HierIn.nClasses,
NC is IDClass + 1,
Classes = Classes0.put(IDClass, Class),
add_edges(HierIn.hierarchy, [0-IDClass], TreeH),
add_subClass_expl(HierIn.explanations, Class, 'http://www.w3.org/2002/07/owl#Thing', Expls),
KB = HierIn.put([hierarchy=TreeH, nClasses=NC, classes=Classes, explanations=Expls, classesName=[Class|ClassesN]]).

Fig. 14. Implementation of the add_class/3 predicate. It adds a new concept Class to the hierarchy by updating its structure and the list containing information about all the classes of the KB. The predicate find/1 is used to perform the search by value in the dictionary.

Given this structure, called kb, and a concept Class, it looks for it in the map between nodes and concepts classes contained in kb. If the class is already present in the map, it means that the hierarchy already contains such a concept and therefore nothing must be done. Otherwise, the new concept is added to the map classes retrieving the name of the new node to insert in the hierarchy graph \( ID_{class} = IDClass \). This new node is then included in the concept hierarchy and linked to the \( \top \) concept (node 0) by means of add_edges/3 predicate. Moreover, the add_subClass_expl/5 predicate adds to the hierarchy a new axiom explaining the edge connecting the two classes taken in input. In Figure 14 it is used to specify that the axiom that links Class with the \( \top \) concept is of the form \( \text{Class} \sqsubseteq \top \).

After the inclusion of each class in the KB, the hierarchy is updated using subsumption axioms. For each sub-class axiom \( \text{SubClass} \sqsubseteq \text{SupClass} \) a new link is added in the hierarchy by the predicate add_hierarchy_link/4 shown in Figure 15. This predicate implements the procedure ADD_HIERARCHY_LINK of Figure 6.

After checking if the two concepts SubClass and SupClass are mapped to different nodes, it adds the link by looking first if \( \text{SupClass} \sqsubseteq \text{SubClass} \). In this case, \( \text{SupClass}, \text{SubClass} \) and all the classes on the path from \( \text{SupClass} \) to \( \text{SubClass} \) are equals and then must be merged. This operation is made by the predicate merge_classes/4, which, basically, collects the concepts that must be merged and all the edges in the hierarchy graph going inward and outward these concepts. Then it removes the nodes corresponding to the concepts to be merged from the graph except for that corresponding to \( \text{SupClass} \). Then, it updates the dictionary mapping nodes and identifiers by assigning to \( ID_{\text{SupClass}} \) the list of the merged concepts. Finally, updates the graph by adding all the edges collected before so that they all connect with \( ID_{\text{SupClass}} \). Otherwise, if \( \text{SupClass} \nsubseteq \text{SubClass} \) a new link between the nodes corresponding to \( \text{SubClass} \) and \( \text{SupClass} \) is added and the edge going from 0 (\( \top \) concept) to \( \text{SubClass} \), if present, is removed to facilitate the management of the hierarchy graph. On the other hand, if the two concepts \( \text{SubClass} \) and \( \text{SupClass} \) are mapped to the same node, add_hierarchy_link/4 simply adds the new explanation for the edge.

The structure so built could be used to output information about the KB by calling the predicates kb_info/0, which writes the number of concepts, individuals and roles, and prints the hierarchy. To facilitate the reading, the hierarchy graph is represented as a tree having in its root the \( \top \) concept.
add_hierarchy_link(HierIn, SubClass, SupClass, HierOut):-
  Classes0=HierIn.classes,
  IDsSubClass=Classes0.find(SubClass),
  IDsSupClass=Classes0.find(SupClass),!,
  (dif(IDSSubClass, IDSSupClass) ->
    (add_link_id(HierIn, IDSSubClass, IDSSupClass, SubClass, SupClass, HierOut))
    ;
    (add_subClass_expl(HierIn.explanations, SubClass, SupClass, Expls),
    HierOut=HierIn.put(explanations, Expls))
  ).

add_link_id(HierIn, IDSSubClass, IDSSupClass, SubClass, SupClass, HierOut):-
  are_subClasses(HierIn, IDSSubClass, IDSSupClass),!,
  merge_classes(HierIn, IDSSubClass, IDSSupClass, HierTemp),
  add_subClass_expl(HierTemp.explanations, SubClass, SupClass, Expls),
  HierOut=HierTemp.put(explanations, Expls).

add_link_id(HierIn, IDSSubClass, IDSSupClass, SubClass, SupClass, HierOut):-
  delete_edges(HierIn.hierarchy, [0-IDSSubClass], TreeH1),
  add_edges(TreeH1, [IDSSupClass-IDSSubClass], TreeH),
  add_subClass_expl(HierIn.explanations, SubClass, SupClass, Expls),
  HierOut=HierIn.put([hierarchy=TreeH, explanations=Expls]).

Fig. 15. Implementation of the add_hierarchy_link/4 predicate. It modifies the hierarchy by creating a new link between the given concepts SubClass and SupClass.