Optimizing a Tableau Reasoner and its Implementation in Prolog

Riccardo Zese[a,*], Giuseppe Cota[b]

[a]Dipartimento di Ingegneria, Università di Ferrara, Via Saragat 1, 44122, Ferrara, Italy
[b]Exabyte S.r.l., Via Sant'Agostino 10, 45100, Rovigo, Italy

ARTICLE INFO

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

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 sometimes achieve 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. The first one consists into identifying the fragment of the KB that allows to perform inference without losing the completeness. The second one modifies which tableau rule to apply and their order of application, in order to reduce the number of operations. Experimental results show the effectiveness of the introduced optimizations.

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].

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, 5, 6, 7, 8]. Among them, DISPONTE [9, 10] is a semantics for probabilistic DLs that 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 latter is a framework, written in Prolog, which contains three reasoners, namely, (i) TRILL [14, 10], able to collect the set of all justifications and compute the probability of queries, (ii) TRILL^P [14, 10], which implements in Prolog the tableau algorithm defined by Baader and Peñaloza [15, 16] for returning the pinpointing formula instead of the set of justifications, and (iii) TORNADO [17], which is similar to TRILL^P but represents the pinpointing formula in a way that can be directly used to compute the probability of the query.

However, to query such KBs and so to compute the probability of a query, it is necessary to collect all the possible justifications for the query. This can be done by means of the *tableau algorithm* [18, 19], which is one of the most used approach to perform inference and it is adopted by most DL reasoners. This algorithm applies some expansion rules on a tableau, a representation of the assertional part of the KB. However, some of these rules are non-deterministic, 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 by Zese *et al.* (2018) [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 gather axioms from the KB that are necessary to build the set of justification/pinpointing formula. In this way, only the useful fragment of the KB is considered, and unnecessary operations are avoided. The second one modifies the application of tableau rules to reduce the number of operations by identifying which rules are necessary and which order to follow. The presented optimizations will be discussed in the paper by means of pseudo-code, in order to concentrate on their functional aspects. Code snippets will be presented in the Appendix to help the reader to enter in the detail of the system and the implementation.

All the probabilistic reasoners in the TRILL framework are available in the TRILL on SWISH web application [20]

*Corresponding author
riccardo.zese@unife.it (R. Zese); g.cota@exabytesrl.it (G. Cota)
ORCID(s): 0000-0001-8352-6304 (R. Zese); 0000-0002-3780-6265 (G. Cota)

at http://trill-sw.eu/.

Moreover, we present an extensive experimental evaluation where we compared the previous version of the TRILL framework (version 5.x.x) with the latest one (version 6.x.x) containing the optimizations. The experimental results show that the introduced extensions can significantly speed-up both regular and probabilistic queries. Finally, we compare the TRILL framework with PuLi [21] on two very large KBs. PuLi is a reasoner able to enumerate justifications by applying resolution calculus with answer literals [22]. The approach used by PuLi allows to answer queries w.r.t. $\mathcal{EL}$ ontologies with hundreds of thousands of axioms in few minutes. The results show that, in some cases, TRILL achieves comparable performances than PuLi, which, however, largely outperforms TRILL systems in most cases. Nevertheless, two considerations must be made. First, even if PuLi outperforms the TRILL framework and much work must be still needed to make it competitive with PuLi, the optimizations presented here allow the TRILL framework to better deal with very large KBs. Second, the TRILL framework allows deal with $\mathcal{SHIQ}$ DLs, whereas PuLi supports languages with lower expressivity, such as $\mathcal{EL}$, and a limited set of queries.

Furthermore, it is clear from the results that the overhead produced by the probabilistic computation slightly reduces the speed-up obtained by the new version. However, it has been demonstrated to be usually negligible compared with the time needed by the reasoning phase [17]. All the KBs used in the experiments and the scripts to run the experiments are available at https://github.com/rzese/docker-trill and 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 its extension to find all the justifications, useful for debugging or probabilistic reasoning, which motivate 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) [23, 24] syntax is based on individuals, representing objects of the domain, concepts, which group individuals sharing the same characteristics, and roles, connecting pairs of individuals. There are many DL languages that differ in the constructs that are allowed for defining concepts and roles. We first briefly describe the DL $\mathcal{SHI}$ and then its extension $\mathcal{SHIQ}$.

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

A *knowledge base* (KB) $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$ consists of a TBox $\mathcal{T}$, an RBox $\mathcal{R}$ and an ABox $\mathcal{A}$. An RBox $\mathcal{R}$ is a finite set of *transitivity axioms* $Trans(R)$ and *role inclusion axioms* $R \sqsubseteq S$, where $R, S \in \mathbf{R} \cup \mathbf{R}^-$. A finite set of role inclusion axioms $\mathcal{R}_h$ is called *role hierarchy*. Given $\mathcal{R}_h$, we can define $\dot{\sqsubseteq}$ as the transitive-reflexive closure of $\sqsubseteq$ over $\mathcal{R} \cup \{R^- \sqsubseteq S^- | R \sqsubseteq S \in \mathcal{R}\}$. A role $R \in \mathbf{R} \cup \mathbf{R}^-$ is *simple* w.r.t. $\mathcal{R}$ if for all $S \dot{\sqsubseteq} R$, S is not transitive. A *TBox* $\mathcal{T}$ is a finite set of *concept inclusion axioms* $C \sqsubseteq D$, where C and D are concepts. An *ABox* $\mathcal{A}$ is a finite set of *concept membership axioms* $a : C$ and *role membership axioms* $(a, b) : R$, where C is a concept, $R \in \mathbf{R}$ and $a, b \in \mathbf{I}$. With respect to $\mathcal{SHI}$, the DL $\mathcal{SHIQ}$ adds new constructs for the definition of qualified number restrictions, i.e., given a concept C and a simple role $R \in \mathbf{R} \cup \mathbf{R}^-$, 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 $\mathcal{SHI}$ or $\mathcal{SHIQ}$ KB is usually assigned a semantics in terms of interpretations $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty *domain* and $\cdot^{\mathcal{I}}$ is the *interpretation function*, which assigns an element in $\Delta^{\mathcal{I}}$ to each $a \in \mathbf{I}$, a subset of $\Delta^{\mathcal{I}}$ to each concept and a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ to each role. A query Q over a KB $\mathcal{K}$ is 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 *[25]:*

$\exists hasAnimal.Pet$	$\sqsubseteq$	$PetOwner$
$Cat \sqsubseteq Pet$		
$fluffy : Cat$		$(kevin, fluffy) : hasAnimal$
$tom : Cat$		$(kevin, tom) : hasAnimal$

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 $Q = kevin : PetOwner$.

2.2. The Tableau Algorithm

Reasoning tasks such as query answering are reduced to consistency checking or satisfiability checking. This is typically realised with the *tableau algorithm* [23]. In its basic definition, a *tableau* is an ABox represented using a tuple $G = (V, E, \mathcal{L}, \dot{\neq})$ 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 $\langle a, b \rangle \in E$ in the graph is labelled with the set of roles $\mathcal{L}(\langle a, b \rangle)$. The binary predicate $\dot{\neq}$ 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 for which $a : C \in \mathcal{K}$,

and an edge $e = \langle a, b \rangle$ 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 and it is fully expanded, the tableau represents a model for the negation of the query.

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 $\mathcal{T}$ of completion graphs is built instead of one. $\mathcal{T}$ is initialized with a single completion graph, say G_0, which is initialized as described above for G. Every application of a rule modifies $\mathcal{T}$. In particular, the application of a deterministic rule to a completion graph G_i returns a new graph G_i' which replaces G_i. In the case of applying 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. Note that, each time the tableau algorithm tries to apply a rule, first it must test if the rule is applicable. If it is so, then it applies the rule, updating the tableau as described above.

For ensuring the termination of the algorithm, a cycle detection technique known as *blocking* [26, 27, 28] is used.

Soundness and *completeness* of the tableau algorithm are proved by Baader *at al.* [23].

2.2.1. Representing the Set of All Justifications

An important problem to solve is finding the set of all justifications for a given query. This non-standard reasoning service is also known as *axiom pinpointing* [29] and it is useful for tracing derivations and debugging ontologies. This problem has been investigated by various authors [30, 29, 31, 32].

The axiom pinpointing problem is also important for probabilistic inference. For example, the DISPONTE semantics [9, 10] requires the set of all justifications to compute the probability of the queries. The axioms of the KB are associated with a probability, therefore, from the set of all justifications it is possible to compute the probability that the query holds w.r.t. the KB.

The set of all 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 $\mathcal{J} \subseteq \mathcal{K}$ is a justification if for all $\mathcal{J}' \subset \mathcal{J}$, $\mathcal{J}' \not\models Q$, i.e., $\mathcal{J}'$ is not an explanation for Q.*

The set of all justifications for the query Q given a KB $\mathcal{K}$ is denoted by $\text{ALL-JUST}(Q, \mathcal{K})$.

The set of all 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.

In the next section we will give a brief overall description of how the tableau algorithm can find $\text{ALL-JUST}(Q, \mathcal{K})$ or the pinpointing formula. Example 2 shows the different representation of the $\text{ALL-JUST}(Q, \mathcal{K})$ and the pinpointing formula to show the output obtainable by the different systems in the TRILL framework.

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

$$E_1 = \exists hasAnimal.Pet \sqsubseteq PetOwner$$
$$E_2 = fluffy : Cat$$
$$E_3 = tom : Cat$$
$$E_4 = Cat \sqsubseteq Pet$$
$$E_5 = (kevin, fluffy) : hasAnimal$$
$$E_6 = (kevin, tom) : hasAnimal$$

Let $Q = kevin : PetOwner$ be the query, then

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

while the pinpointing formula is

$$((E_5 \wedge E_2) \vee (E_6 \wedge E_3)) \wedge E_4 \wedge E_1$$

2.2.2. Axiom Pinpointing with Tableau Algorithm

In order to solve the axiom pinpointing problem, the tableau algorithm has been modified so that each expansion rule updates a *tracing function* τ as well. This function associates each concept (role) in the label of a node (edge) with a justification found so far, either in the form of a set of axioms as defined in Definition 2 [33] or in the form of a pinpointing formula [16], depending on which definition of the tracing function is used.

For the sake of simplicity, in this Section we do not analyse the tracing function and the different ways the reasoners test the applicability of the rules, because the optimizations proposed in the paper are not affected by the choice of the definition of the tracing function. It is sufficient to know that it is used to find the justifications of a query. Basically, the tracing function associates a justification to each label of the tableau. Given a query, once the tableau is fully expanded, to build the justification for the query, the labels that cause clashes are collected. Then, the justifications of these labels are put together to form the justification for the query. We refer the interested reader to Zese (2017) [10] for a detailed overview.

The expansion rules 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, as seen in Example 3.

Example 3. *Consider the following KB and justifications as defined in Definition 2:*

$$a : C$$
$$a : D$$
$$C \sqsubseteq E$$
$$C \sqsubseteq D$$

The initial tableau has a node for the individual a such that $\mathcal{L}(a) = \{C, D\}$*, with justifications initialized as* $\{a : C\}$ *and* $\{a : D\}$*. Given the presence of the axiom* $C \sqsubseteq E$*, a rule may add to* $\mathcal{L}(a)$ *the concept* E *with the justification* $\{a : C, C \sqsubseteq E\}$*.*

In the KB, there is another axiom to consider, $C \sqsubseteq D$*. In this case, the rule should add* D *to* $\mathcal{L}(a)$ *with justification* $\{a : C, C \sqsubseteq D\}$*. However,* D *is already present in* $\mathcal{L}(a)$*, so, the applicability test fails, and the rule will not update the tableau.*

To solve the axiom pinpointing problem, reasoners must explore the entire search space of the possible explanations. Classical tableau-based systems, usually implemented using imperative languages, allow the tableau algorithm to find a new justification in several ways. The most used approach, called *Hitting Set Tree* (HST) [34], basically takes a KB and a query, and run the tableau to get a first justification. If a justification is found, the HST algorithm selects an axiom from the justification, removes it form the KB and runs the tableau on this new KB. If a new justification is found, recursively, it selects an axiom from this new justification, removes it from the KB, and calls the tableau. Otherwise, the removed axiom is re-included in the KB. For instance, given a KB $\mathcal{K}$ and a query Q, if the justification $\mathcal{J}_1 = \{E_1, E_2, E_3\}$ is found, where E_is 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 $\mathcal{J}$, one at a time, i.e., it looks for a new justification w.r.t. $\mathcal{K} \setminus \{E_2\}$ and $\mathcal{K} \setminus \{E_3\}$. Otherwise, if a new justification $\mathcal{J}_2 = \{E_4, E_5\}$ is found, before backtracking, the HST algorithm creates a new choice point and calls the reasoner to find a new justification w.r.t. $\mathcal{K}'' = \mathcal{K}' \setminus \{E_4\}$. 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. The algorithm also maintains in memory the set of axioms removed from the KB in each step to reduce the number of calls to the tableau.

Example 4. *Consider again the KB of Example 3 and the query* $Q := a : D$*. In the first call of the tableau, as seen in Example 3, the HST algorithm gets the justification* $\mathcal{J}_1 = \{a : D\}$*. It selects an axiom from* $\mathcal{J}_1$ *and removes it from the KB. In this example,* $\mathcal{J}_1$ *contains only the axiom* $a : D$*, so the new KB becomes as follows:*

$$a : C$$
$$C \sqsubseteq E$$
$$C \sqsubseteq D$$

Then, the HST algorithm calls the tableau algorithm w.r.t. this new KB. The initial tableau has now a node for the individual a such that $\mathcal{L}(a) = \{C\}$*, with its justification initialized as* $\{a : C\}$*. Given the presence of the axiom* $C \sqsubseteq D$*, a rule adds to* $\mathcal{L}(a)$ *the concept* D *with justification* $\{a : C, C \sqsubseteq D\}$*. This is the new justification* $\mathcal{J}_2$ *that the tableau algorithm returns for* Q *to the HST algorithm.*

Next, the HST algorithm selects one axiom from $\mathcal{J}_2$ *and removes it from the above KB. However,* Q *is not entailed by the resulting KB, thus the removed axiom is included back. This operation is done for each axiom of each justification, until every combination of axioms has been removed by the original KB and the query tested on each of the resulting KBs.*

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 [35, 36, 37, 38, 39]. Another possibility is to apply an extension of (disjunctive) ASP [40] or an abductive proof procedure to perform ontological reasoning [41, 42]. We proposed three systems, TRILL [14, 10], TRILL^P [14, 10], and TORNADO [17], that implement the tableau algorithm in Prolog. These reasoners will be briefly presented in Section 3.

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 [14, 10], TRILL^P [14, 10], and TORNADO [17], which use the Prolog language to implement the tableau algorithm and compute the probability of the queries under the DISPONTE semantics.

TRILL solves the ALL-JUST($Q, \mathcal{K}$) problem for $\mathcal{SHIQ}$ DL, it can return the set of all justifications and the probability of the queries computed by means of knowledge compilation using BDDs[1] [17]. TRILL^P modifies TRILL by representing the set of all justifications in a possibly more compact way. This is done by means of a different definition of the tracing function. However, this approach is correct and terminating for the DL $\mathcal{SHI}$ [16]. The set of all the justifications must be translated into a BDD to compute the query probability. Analogously, TORNADO optimizes TRILL^P by directly representing the values of the tracing function as

[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.

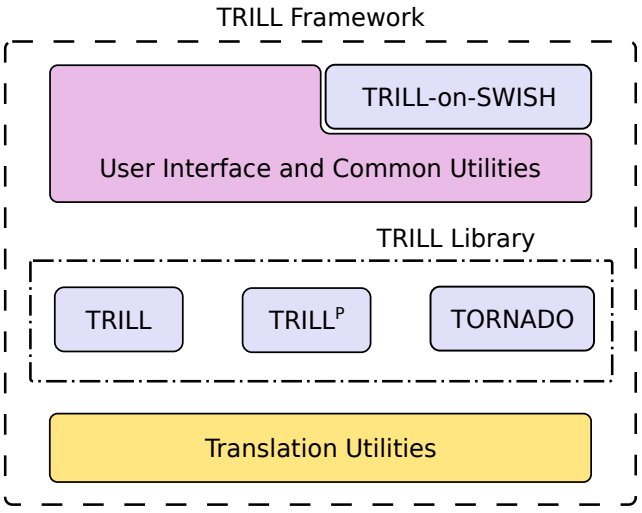

Figure 1: Software architecture of the old version of TRILL.

a BDD, avoiding some exponential blow-ups in the inference process. As TRILL^P, it supports the $\mathcal{SHI}$ DL.

The TRILL framework is implemented in SWI-Prolog [43], the code of all three systems is available at `https://github.com/rzese/trill`.

The TRILL framework forms a layer cake, shown in Figure 1, 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 [44] 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 defines the user interface and the queries that can be asked. In particular, TRILL, TRILL^P, and TORNADO can answer concept and role membership queries, subsumption queries and can test the unsatisfiability of a concept of the KB or the inconsistency 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, class and role assertions with the corresponding value of the tracing function, i.e., a justification for the label. The tracing function stores a fragment of the KB in TRILL, the pinpointing formula in TRILL^P, and the BDD representing the pinpointing formula in TORNADO. T contains the structure of the tableau. 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 tested and, in case the applicability test succeeds, applied in order. This is done by the procedure APPLY_ALL_RULES, first the deterministic ones and then the non-deterministic ones, as shown in Figures 2 and 3.

In our case, $\mathcal{SHIQ}$ DL, the tableau algorithm terminates with any rule application order. However, this is not true for more expressive DLs like $\mathcal{SHOIQ}$, and therefore a rule application strategy must be adopted to assure termination [18]. Considering $\mathcal{SHIQ}$, TRILL systems allow the user to easily change the order of the expansion rules using `set_tableau_expansion_rules(DetRuleList,NonDetRuleList)` predicate. Calling this goal before a query forces TRILL to use the rule defined in the two lists in the order they are reported in the lists. Eventually, TRILL may prune some rules if they are useless for the query.

Expansion rules can only add new assertions into the tableau, therefore if a rule *Rule* is applicable on a tableau T on a certain assertion $a : C$, it is so in every expansion of T obtained by the application of other rules different from *Rule* and/or the application of *Rule* on assertions different to $a : C$. 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. The first type of non-determinism is handled principally by the rule order of application, which is usually fixed, and by the applicability test of each rule, which tests whether the changes the rule would make on the tableau are already present. Another way to manage this type of non-determinism is the use of expansion queues, as discussed in Section 4.2. As for the "don't know" non-determinism, when a non-deterministic rule creates a set of tableaux, the choice of which of them to consider for further expansion opens a choice point. To help TRILL to manage these choice points, each of them is associated with an identifier. These identifiers allow knowing the sequence of choice points that generates every tableau.

Even though for $\mathcal{SHIQ}$ DL the expansion rules application order does not affect the completeness of the algorithm, it can significantly affect the performance [45, 46]. Usually, non-deterministic rules are tried as late as possible in order to avoid performing same expansions more than once.

Example 5. *Consider the following simple KB:*

$$C \sqsubseteq D$$
$$a : C$$
$$a : E \sqcup F$$

Also consider two expansion rules (whose definitions are simplified for the purpose of this example):

→ ***unfold*** *given an individual a, if there is a concept $C \in \mathcal{L}(a)$ and an axiom $C \sqsubseteq D$ in the KB, then if $D \notin \mathcal{L}(a)$ add D to $\mathcal{L}(a)$*

```
1: procedure APPLY_ALL_RULES(TabIn)
   TabIn: the input tableau.
   Returns TabOut: the output tableau created by the application of the rules.
2:     DetRules ← the list of deterministic rules
3:     NonDetRules ← the list of non-deterministic rules
4:     do
5:         TabOut ←APPLY_ONE_RULE(DetRules, NonDetRules, TabIn)
6:         if TabOut ≠ null then
7:             TabIn ← TabOut
8:         end if
9:     while TabOut ≠ null
10:    return TabIn
11: end procedure
```

Figure 2: Application of the expansion rules by means of the procedures APPLY_ALL_RULES. The lists *DetRules* and *NonDetRules* contain the available rules and is different in TRILL, TRILL^P and TORNADO.

```
1: procedure APPLY_ONE_RULE(DetRules, NonDetRules, TabIn)
   TabIn: the input tableau.
   DetRules: the list of deterministic expansion rules.
   NonDetRules: the list of non-deterministic expansion rules.
   Returns TabOut: the output tableau created by the application of one rules.
2:     TabOut ←APPLY_DET_RULES(DetRules, TabIn)
3:     if TabOut = null then          ▷ This is necessary to apply non-det. rules only when det. rules are not applicable.
4:         TabOut ←APPLY_NONDET_RULES(NonDetRules, TabIn)
5:     end ifreturn TabOut
6: end procedure

7: procedure APPLY_DET_RULES(Rules, TabIn)
   Rules: the list of rules to try to apply.
   TabIn: the input tableau.
   Returns TabOut: the output tableau created by the application of the rules.
8:     for all Rule ∈ Rules do
9:         if Rule is applicable then
10:            TabOut ← Rule(TabIn)                                  ▷ Applies the rule Rule.
11:            return TabOut
12:        end if
13:    end for
14:    return null                                                   ▷ If det. rules are not applicable.
15: end procedure

16: procedure APPLY_NONDET_RULES(Rules, TabIn)
   Rules: the list of rules to try to apply.
   TabIn: the input tableau.
   Returns TabOut: the output tableau created by the application of one rule.
17:    for all Rule ∈ Rules do
18:        if Rule is applicable then
19:            TabOutL ← Rule(TabIn)                                 ▷ The rule returns a list of tableaux.
20:            Create a choice point and choose a TabOut ∈ TabOutL
21:            return TabOut
22:        end if
23:    end for
24:    return null                                                   ▷ If non-det. rules are not applicable.
25: end procedure
```

Figure 3: Application of one expansion rule, chosen among deterministic and non-deterministic ones, by means of the procedures APPLY_ONE_RULE, which exploits APPLY_DET_RULES and APPLY_NONDET_RULES. The lists *DetRules* and *NonDetRules* contain the available rules and is different in TRILL, TRILL^P and TORNADO.

→ ⊔ *given an individual a, if there is a concept $C \sqcup D \in \mathcal{L}(a)$, i.e., a belongs to either C or D, then if $\{C, D\} \cap \mathcal{L}(a) = \emptyset$, create two new tableau and add C to $\mathcal{L}(a)$ in the first tableau and D to $\mathcal{L}(a)$ in the second tableau.*

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

Otherwise, if we apply the → ⊔ rule first, we obtain two tableaux with $\mathcal{L}(x) = \{C, E \sqcup F, E\}$ and $\mathcal{L}(x) = \{C, E \sqcup F, F\}$ respectively. At this point, the → 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_ALL_RULES (lines 1-11, Figure 2) repeatedly calls the procedure APPLY_ONE_RULE (line 5, Figure 2), which takes the list of deterministic and non-deterministic rules and tries to apply one of them. If one of the rules is applicable, it returns the resulting tableau, otherwise, it returns *null*. Thus, if APPLY_ONE_RULE returns a tableau, the do-while loop continues, APPLY_ALL_RULES assigns *TabOut* to *TabIn* in order to maintain the last tableau computed (line 6), and calls APPLY_ONE_RULE again. Otherwise, if APPLY_ONE_RULE returns *null*, APPLY_ALL_RULES quits the loop and returns the input tableau *TabIn* (line 10, Figure 2).

To apply the rules, APPLY_ONE_RULE (whose pseudo-code is shown in Figure 3, lines 1-6) calls first the procedure APPLY_DET_RULES (lines 7-15, Figure 3) and then the procedure APPLY_NONDET_RULES (lines 16-25, Figure 3). The first 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 deterministic rules or, if no rules can be applied, it returns *null*. The procedure APPLY_NONDET_RULES 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. If a non-deterministic rule is applicable, the list of tableaux obtained by its application is returned by the corresponding predicate and a tableau from the list is non-deterministically chosen by APPLY_NONDET_RULES and returned to APPLY_ONE_RULE (line 21). Otherwise, if no rule is applicable, APPLY_NONDET_RULES returns *null* (line 24).

Finally, thanks to the backtracking, all the possible choices opened by the procedure APPLY_NONDET_RULES will be explored. In our case, TRILL systems collect the list of all the possible tableaux by exploiting the Prolog's backtracking facilities, not shown in the pseudo-code. Note that other systems implemented using different languages must implements an algorithm to explore all the choice points, such as the aforementioned HST algorithm. Once collected all the tableaux, TRILL looks 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.

4. Improving TRILL

The improvements implemented in the new version of the TRILL framework are basically twofold: an extraction of the fragment of the KB relevant to the query, which will be considered during the tableau expansion, and an optimization of the application order of the tableau expansion rules, which also determines which rule will be considered during the expansion of the tableau. As we will see in the following, the first extension is used to avoid the expansion of irrelevant information and thus the application of unnecessary expansion rules. For the sake of simplicity, we consider only consistent KBs. However, this procedure may help making the reasoner able to also reason w.r.t. inconsistent KBs, because, with some minor changes, it applies an approach similar to those implemented by a reasoner following the Repair semantics[2].

On the other hand, the second is used 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 to see an improvement of the performance for general KBs, while in the case of very simple KBs, the two versions should perform in a similar way.

4.1. Extraction of the Relevant KB Fragment

The first improvement allows TRILL to collect useful information about the query before starting the initialization of the tableau. In particular, the TRILL systems extract from the KB a collection of individuals that are related to those relevant for the query.

To describe this optimization, we first need to define the concept of *related* individuals. Formally, given a KB $\mathcal{K} = (\mathcal{T}, \mathcal{R}, \mathcal{A})$, two individuals $a, b \in \mathcal{A}$ are *directly related* if there exists at least a role $R \in \mathcal{R}$ such that there is either a role membership axiom $(a, b) : R$ or $(b, a) : R$ in $\mathcal{K}$. The concept of *indirectly related* is defined as the transitive closure. If two individuals are directly or indirectly related, we can simply say that they are related.

The idea behind this optimization is simple: given a query Q, if it asks for information about certain individuals, then it is sufficient to expand the tableau only for these individuals and for those that are directly and indirectly related to them. In other words, if we need to know whether *kevin* is a *PetOwner* we should concentrate on him and the individuals related to him, while we can forget all other individu-

[2] A *repair* is a consistent subset of axioms of a possibly inconsistent KB. There may be many different repairs for a KB. Moreover, there are different repair semantics, defining how a query can be answered w.r.t. the built repairs [47, 48].

als (completely unrelated to *kevin*) because they cannot give useful information about *kevin*.

Once collected the set of the individuals related to those of the query, the tableau can be initialized using only the individuals in this set. Therefore, TRILL considers only the fragment of the ABox that is relevant for the query. Clearly, the TBox remains unchanged to avoid losing useful information about the taxonomy of the KB.

Note that these definitions are correct unless nominals are considered. A nominal is a concept defined as a set of individuals. In this case, the definition of *related* individuals must also consider the individuals in the nominal concept. If during the gathering of the related individuals we encounter an individual that belongs to such a concept, also the individuals contained in the definition of the concept must be collected. Moreover, in presence of nominals, the set of considered individuals must be carefully updated also during the expansion of the tableau, because in this scenario there could be combinations of axioms forcing the merge of individuals not related. However, we do not consider this eventuality as we are considering the DLs $\mathcal{SHI}$ and $\mathcal{SHIQ}$, which does not consider nominals.

It is worth noting that some expansion rules for $\mathcal{SHI}$ and $\mathcal{SHIQ}$ DLs create new individuals not present in the KB, called *anonymous* or *fresh*. Moreover, there is a rule for $\mathcal{SHIQ}$ DL, which allows the use of maximum cardinality restrictions, that tries to merge individuals already present in the KB [18].

In the first case, if an expansion rule adds new individuals, they are added to the tableau, and so considered during future expansions. Moreover, these rules add new individuals that are related to the individual of the assertion that triggered the expansion rule.

On the other hand, the rule that merges individuals considers for merge only individuals directly related to that present in the assertion triggering the expansion rule itself. The directly related individuals can either come from the KBs, and so, they are surely added to the tableau because considered during the collection of the related individuals, or be anonymous individuals created by a previously applied expansion rule, and so, present in the tableau.

There are no rules for $\mathcal{SHIQ}$ DL that can add role assertions between two individuals present in the KB that are not related (directly or indirectly). They can only add role assertions relating newly created anonymous individuals.

This optimization is particularly effective in Prolog because of the unification mechanism. If the axioms are loaded in memory as facts, unification allows a fast extraction of directly related individuals.

Let us consider, e.g., the axiom $(kevin, tom) : hasAnimal$. By following the definitions of the Thea2 library, TRILL represents role membership axioms by means of predicate `propertyAssertion/3`, i.e., the axiom above is translated to `propertyAssertion(hasAnimal,kevin,tom)`. Thus, when the reasoner has an individual *a*, using Prolog, it simply must unify the predicate `propertyAssertion(_,a,IndRelated)` or the predicate `propertyAssertion(_,IndRelated,a)` to extract the related individual, which is the value taken by the variable `IndRelated`. The symbol _ represents that we do not care about the role defined by the axiom, but only the individuals. The unification allows a fast search within the axioms of the KB, requiring only an access to the KB for each unification, which is in main memory. Therefore, the use of the Prolog language is intertwined with the implementation of this specific optimization which exploits a distinctive characteristic of the language.

4.1.1. Pseudo-code

The pseudo-code of this procedure, which is called COLLECT_INDIVIDUALS, is shown in Figure 4. Basically, TRILL starts from the individuals of Q that are collected by the procedure EXTRACT_FROM_Q (line 2). Note that, if Q is a class assertion or a property assertion, the procedure returns a set containing one and two individuals respectively, if Q is a different kind of query, for example a subclass-of axiom, then the set *IndividualsSet* will be empty.

Then, for each individual *Individual* in the set of individuals to check *IndividualsToCheck*, if not already checked, the procedure collects all the individuals directly related to it by means of GATHER_DIRECTLY_RELATED_INDIVIDUALS procedure (lines 17-25) and removes *Individual* from the set of individuals that must be still considered (lines 6-7). The individual is also included in a second set, *IndividualsChecked*, in order to avoid considering the same individuals more than one time (lines 8-13).

The procedure must consider the two sets *IndividualsToCheck* and *IndividualsChecked* to avoid considering the same individual more than once. For example, if the KB contains the following axioms $\{(a,b) : R, (b,c) : R, (c,a) : R\}$, if COLLECT_INDIVIDUALS did not maintain the two sets, it would enter in an infinite loop. By maintaining the two sets, this problem is avoided.

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 allow 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 6. *Suppose that the tableau contains k assertions* $\{a : C_1, \ldots, a : C_k\}$ *and that the KB contains only the axiom* $C_k \sqsubseteq C_{k+1}$. *In this example, the only rule that will modify the tableau is the* $\rightarrow$ *unfold rule and will be applicable only once. In the tableau algorithm, there is a sequence of expansion rule, called following a certain order usually defined in the specific system. As seen in Section 2.2, each rule, when called, performs a set of tests to check whether the rule can be applied or not. As can be seen in the rules in Example 5, these tests can be, e.g., the presence in the label*

```
1: procedure COLLECT_INDIVIDUALS(Q, K)
   Q: the query.
   K: the KB considered to answer the query Q.
   Returns IndividualsSet: set of all the individuals related with those in Q.
2:     IndividualsSet ← EXTRACT_FROM_Q(Q)                ▷ Returns the set of individuals in the query.
3:     IndividualsToCheck ← IndividualsSet
4:     IndividualsChecked ← ∅
5:     while IndividualsToCheck ≠ ∅ do
6:         Individual ← EXTRACT(IndividualsToCheck)      ▷ Select one individual from the set IndividualsToCheck.
7:         IndividualsToCheck ← IndividualsToCheck \ Individual
8:         if Individual ∉ IndividualsChecked then
9:             IndividualsChecked ← IndividualsChecked ∪ Individual
10:            NewIndividualsToCheck ← GATHER_DIRECTLY_RELATED_INDIVIDUALS(Individual, K)
11:            IndividualsSet ← IndividualsSet ∪ NewIndividualsToCheck
12:            IndividualsToCheck ← IndividualsToCheck ∪ NewIndividualsToCheck
13:        end if
14:    end while
15:    return IndividualsSet
16: end procedure

17: procedure GATHER_DIRECTLY_RELATED_INDIVIDUALS(Individual, K)
   Individual: the individual for which we want to collect all the related individuals.
   K: the KB considered to answer the query Q.
   Returns RelatedIndividuals: set of all the individuals related to Individual.
18:    for all (Individual, RelatedIndividuals) : R ∈ K do
19:        RelatedIndividuals ← RelatedIndividuals ∪ {RelatedIndividuals}
20:    end for
21:    for all (RelatedIndividuals, Individual) : R ∈ K do
22:        RelatedIndividuals ← RelatedIndividuals ∪ {RelatedIndividuals}
23:    end for
24:    return RelatedIndividuals
25: end procedure
```

Figure 4: Extraction of the individuals related to those of the query.

of a node of a certain concept. If the tests succeed, the rule is applicable, thus the tableau algorithm applies it updating the tableau.

In this example, to fully expand the tableau, all the rules are tested until → unfold rule, which is applicable, so its applicability test succeeds, and it is applied to the tableau. After this application, the tableau is fully expanded. Then, a second round of rule application is performed, but since no more rules are applicable, they are all tested, but none is applied. At this point the expansion stops.

Therefore, the rules before the → unfold rule are tested twice. The → unfold rule is tested and applied in the first round and only tested in the second round. The remaining rules, those tested after the → unfold rule, are tested only once, in the second round.

For the sake of simplicity, we assume that every rule performs one single test, which is done by an atomic instruction. Although, actually, the complexity of the applicability test heavily depends on the expansion rule itself and the definition of the tracing function.

Let us suppose that there are n rules and the → unfold rule is the third in the order, suppose also that the assertions 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 tested on all the assertions and, finally, 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 on each of the $k+1$ assertions, but no one can modify the tableau, which is fully expanded. At the end of the process $3k+n(k+1)$ tests have been performed to decide whether a rule can be applied or not, that are 3 rules on k assertions in the first round, when the third rule tested (→ unfold) adds a new assertion, so in the second round each of the n rules are tested on $k+1$ assertions.

In the worst case, where the → unfold is the last rule applied, the number of tests is $nk + n(k+1) = 2nk + n$

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 justification has not been changed by now), it does not lead to further expansions of the tableau and, hence, it will not until the end of the expansion.

Following this observation, reasoners using the tableau algorithm implement systems, usually based on queues or heuristics, to guide the rule applications, avoiding the prob-

lem described above. For these reasons, we changed the way TRILL, TRILL^P and TORNADO apply the expansion rules in two ways. The first is straightforward, when the query is added to the tableau, TRILL systems check the tableau, the KB, and the query to determine which rules surely cannot be applied during the inference process. For example, if the tableau and the KB do not contain cardinality restrictions, the expansion rules geared to them cannot apply. Thus, these rules are pruned and never tested during the expansion. As regards the latter, in each round of rules application, instead of trying every rule w.r.t. every assertion, TRILL systems keep updated a structure called *ExpansionPairQ*.

It is also worth noting that the order of the expansion rules can be easily changed by the user by means of the `set_tableau_expansion_rules(DetRuleList,NonDetRuleList)` predicate. Calling this goal before a query forces TRILL to use the rule defined in the two lists in the order they are reported in the lists. Eventually, TRILL may prune some rules if they are useless for the query.

The structure *ExpansionPairQ* is composed of a pair of expansion queues: *DetQueue* and *NonDetQueue*. The first one contains every assertion triggering only deterministic rules contained in the tableau, while the second contains assertions triggering non-deterministic rules, i.e., disjunctions of concepts ($\sqcup$) and maximum cardinality restrictions ($\leq$). Each assertion can be included in only one queue and, in this queue, it can appear only once. Both 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 expansion rules.

During the expansion, an assertion is dequeued from *ExpansionPairQ*, first the assertions from *DetQueue* are extracted and only when *DetQueue* is empty they are extracted from *NonDetQueue*. The dequeued assertion is used to try to apply only the expansion rules that match the assertion. If a rule can be applied to a tableau, it also updates the queues in *ExpansionPairQ*. The rule adds the assertions corresponding to the labels it updated to the correct queue. If the rule adds an assertion that is already present in *ExpansionPairQ*, this assertion is first deleted by *ExpansionPairQ*, in order to avoid duplicates.

4.2.1. Pseudo-code

The procedure APPLY_ALL_RULES of Figure 2 is replaced by the EXPAND_QUEUES procedure, shown in Figure 5, which calls a slightly different version of APPLY_ALL_RULES which now takes 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* by means of procedure DEQUEUE (lines 12-21, Figure 5). This procedure takes the pair of queues *ExpansionPairQ* and first tries to pop an assertion *AssToExpand* from *DetQueue*, i.e., the queue containing assertions triggering only deterministic rules, if the queue is not empty. If *DetQueue* is empty, then, it tries to pop *AssToExpand* from *NonDetQueue*, i.e., the queue containing those triggering non-deterministic rules. If the queue *NonDetQueue* is empty as well, *AssToExpand* is set to *null*. Finally, DEQUEUE returns *AssToExpand*. Note that the operation POP updates the queue *Queue* it takes as input by removing the assertion *Ass* that it returns, i.e., $Queue \leftarrow Queue \setminus \{Ass\}$.

The EXPAND_QUEUES procedure stops when there are no more assertions to expand. The APPLY_ALL_RULES procedure used in Figure 5 and shown in Figure 6 differs from that shown in Figure 2 because it also takes as input the assertion *AssToExpand*. Then, this assertion is passed to APPLY_ONE_RULE (line 5, Figure 6). Then, this procedure will pass *AssToExpand* to the expansion rules that try to expand the tableau considering the assertion (lines 10 and 19, Figure 7). The Prolog code implementing these procedures can be found in Section A.3 in the Appendix.

Passing *AssToExpand* simplifies the process of rule application. Without considering *AssToExpand*, each expansion rule should explore the tableau looking for a possible label matching its precondition, for example searching for nodes containing a certain type of concept. By considering *AssToExpand* this search is avoided because the rule already knows which label must consider. Second, it simplifies the choice of which rule to try. In fact, without considering *AssToExpand* and/or expansion queues, the original version of the tableau algorithm should call every expansion rule. Now, only the rules that match the assertion *AssToExpand* are called.

Example 7. *Consider the case depicted in Example 6, 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 expansion rules of which the* $\rightarrow$ *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 tests. Let us 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 Example 6, in the first round TRILL tests all the assertions. This is because, during the initialization of the tableau, every assertion added to the tableau has been added in DetQueue. The main difference in this case is in the second round, where DetQueue contains only the new assertion $a : C_{k+1}$*, therefore, the number of tests in this case is* $nk + n$*, which corresponds with the worst case, i.e., the case in which* $\rightarrow$ *unfold is the last rule tested and every rule matches every assertion, 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 assertions, e.g., the $\rightarrow \sqcup$ *matches only with* `unionOf` *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

```
1: procedure EXPAND_QUEUES(TabExIn)
   TabExIn: the input tableau associated with its pair of expansion queues.
   Returns TabExOut: the final tableau.
2:     ExpansionPairQ ← GETEXPANSIONPAIRQ(TabExIn)
3:     AssToExpand ← DEQUEUE(ExpansionPairQ)                ▷ Also updates ExpansionPairQ.
4:     if AssToExpand is not null then
5:         TabEx ← REPLACEEXPANSIONPAIRQ(TabExIn, ExpansionPairQ)
6:         TabExOut ← APPLY_ALL_RULES(TabEx,AssToExpand)
7:         return EXPAND_QUEUES(TabExOut)
8:     else
9:         return TabExIn
10:    end if
11: end procedure

12: procedure DEQUEUE(ExpansionQ)
    ExpansionQ: the expansion queue.
    Returns AssToExpand: the assertion to expand if there are any or null.
    Updates ExpansionQ: removes from ExpansionQ the assertion AssToExpand.
    Uses POP(Queue) that extracts and returns the first assertion Ass and updates Queue ← Queue \ {Ass}.
13:    if ExpansionQ.DetQueue not empty then
14:        AssToExpand ← POP(ExpansionQ.DetQueue)
15:    else if ExpansionQ.NonDetQueue not empty then
16:        AssToExpand ← POP(ExpansionQ.NonDetQueue)
17:    else                                                 ▷ ExpansionQ is empty.
18:        AssToExpand ← null
19:    end if
20:    return AssToExpand
21: end procedure
```

Figure 5: Application of the expansion rules by means of the procedure EXPAND_QUEUES, which in turns calls APPLY_ALL_RULES to expand, if possible, the assertion extracted from the queue.

```
1: procedure APPLY_ALL_RULES(TabEx, AssToExpand)
   TabIn: the input tableau.
   AssToExpand: the assertion to expand.
   Returns TabExOut: the output tableau created by the application of the rules.
2:     DetRules ← the list of deterministic rules
3:     NonDetRules ← the list of non-deterministic rules
4:     do
5:         TabExOut ←APPLY_ONE_RULE(DetRules, NonDetRules, AssToExpand, TabEx)
6:         if TabExOut ≠ null then
7:             TabEx ← TabExOut
8:         end if
9:     while TabExOut ≠ null
10:    return TabEx
11: end procedure
```

Figure 6: Application of the expansion rules on the assertion *AssToExpand* by means of the procedures APPLY_ALL_RULES. In bold, the differences with the pseudocode of Figure 2.

updated. However, as the number of rule applications grows up, this overhead is expected to become negligible and the systems should become faster.

Example 8. *Consider a slightly different version of the tableau and the KB of Examples 6 and 7 where the tableau contains only the assertion* $\{a : C_1\}$ *and the KB contains a set of axioms* $C_i \sqsubseteq C_{i+1}$ *with i from* 1 *to k. With the previous version of TRILL, the number of tests to perform in the worst case, i.e.,* → *unfold is the last rule tested and every rule matches every assertion, are* $n(1 + \ldots + (k + 1))$*. This number comes from the fact that, at the beginning, the tableau contains one assertion* $(a : C_1)$*. In the first round n rules are tested and the last rule adds the new assertion* $a : C_2$*. In the second round of rule application, n rules are tested on the two assertions. For the sake of simplic-*

```
1: procedure APPLY_ONE_RULE(DetRules, NonDetRules, AssToExpand, TabIn)
   DetRules: the list of deterministic expansion rules.
   NonDetRules: the list of non-deterministic expansion rules.
   AssToExpand: the assertion to expand.
   TabIn: the input tableau.
   Returns TabOut: the output tableau created by the application of one rules.
2:     TabOut ←APPLY_DET_RULES(DetRules, AssToExpand, TabIn)
3:     if TabOut = null then          ▷ This is necessary to apply non-det. rules only when det. rules are not applicable.
4:         TabOut ←APPLY_NONDET_RULES(NonDetRules, AssToExpand, TabIn)
5:     end ifreturn TabOut
6: end procedure

7: procedure APPLY_DET_RULES(Rules, AssToExpand, TabIn)
   Rules: the list of rules to try to apply.
   AssToExpand: the assertion to expand.
   TabIn: the input tableau.
   Returns TabOut: the output tableau created by the application of the rules.
8:     for all Rule ∈ Rules do
9:         if Rule is applicable then
10:            TabOut ← Rule(AssToExpand, TabIn)                    ▷ Applies the rule Rule.
11:            return TabOut
12:        end if
13:    end for
14:    return null                                                  ▷ If det. rules are not applicable.
15: end procedure

16: procedure APPLY_NONDET_RULES(Rules, AssToExpand, TabIn)
   Rules: the list of rules to try to apply.
   AssToExpand: the assertion to expand.
   TabIn: the input tableau.
   Returns TabOut: the output tableau created by the application of one rule.
17:    for all Rule ∈ Rules do
18:        if Rule is applicable then
19:            TabOut ← Rule(AssToExpand, TabIn)                    ▷ The rule returns a list of tableaux.
20:            Create a choice point and choose a TabOut ∈ TabOutL
21:            return TabOut
22:        end if
23:    end for
24:    return null                                                  ▷ If non-det. rules are not applicable.
25: end procedure
```

Figure 7: Application of one expansion rule, chosen among deterministic and non-deterministic ones and on the basis of the assertion to expand *AssToExpand*, by means of the procedures APPLY_ONE_RULE, which exploits APPLY_DET_RULES and APPLY_NONDET_RULES. In bold, the differences with the pseudocode of Figure 3.

ity, we consider the assertion in order: $a : C_1, a : C_2, \ldots$. *Note that, if we consider a different order for the assertions, we should also consider the backtracking, however we are reasoning on the worst case in terms of number of tests performed, so this simplification is acceptable. When the last rule considers the last assertion* $a : C_2$, *it updates the tableau adding a third assertion. This will be repeated until the assertion* $a : C_{k+1}$ *is added. So, the number of tests is* $n + 2n + \ldots + (k+1)n$. *On the other hand, the new version needs only* $n(k+1)$ *tests, i.e., in each round only one assertion is considered and n rules are tested for each new assertion. For* $k = 1$ *the difference is* $3 : 2$ *that increases to* $6 : 1$ *when* $k = 10$.

5. Related Work

As regards the extraction of the relevant fragment of the ABox for the query, a possible similar idea is the extraction of KB modules, for which there are many different proposals, for example [49, 50, 51, 52]. These are subsets of axioms of a KB that contain all the knowledge about a specified set of terms. However, they need to impose limitations on the expressivity of the language used, usually spanning from $\mathcal{EL}$ to $\mathcal{HL}$, and on the structure, for example by requiring acyclic terminologies. These modules can be computed on-demand, every time a query is asked, or during the loading of the KB. Usually, the first option is preferred because there may be exponential modules, making their extraction

once for all unfeasible. A similar approach is called atomic decomposition [53]. This approach extracts *atoms*, minimal sets of axioms contained in a KB such that each module either contains all axioms in the atom or none of them, or it is a union of atoms. In this way, the same atom can be used to represent many modules, making them easier to find than modules. Both the definitions of modules and atoms rely on the definition of a notion of locality, which in turns affects the number of modules and the difficulty of their extraction.

The difference with what proposed in this paper is that modules and atoms contain axioms while TRILL systems extract a set of individuals. There are no needs to also select the terminological axioms that will be considered, because this is demanded to the management of the expansion queue on the one hand and to the unification mechanism of Prolog on the other hand, i.e., picking up an axiom from the Prolog's database is a matter of unification, that is almost instantaneous. Moreover, in case of more axioms unifying with what the tableau algorithm is looking for, the backtracking mechanism will automatically manage it. Using other approaches, such as Java OWL API, to find an axiom there is the need to iterate on collections of classes, which is more time and memory consuming.

Another option is to rely on ABox Absorption [54, 55], where the assertional axioms are translated into terminological axioms using nominals, i.e., concepts defined by a set of individuals. In this way, reasoning on the ABox axioms is avoided turning an instance-of query into a subsumption query. However, this forces the reasoner to pre-process the entire KB and the reasoner must consider the new KB, which is as big as the original one. Therefore, usually, ABox absorption is used together with TBox Absorption [56, 57, 58] techniques which simplifies the terminological axioms under certain conditions depending on the language used. Differently, the approach presented here does not involve a preprocess of the KB but a simple extraction of a set of individuals that is then considered during the inference.

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 algorithm takes a pair from the queues following their priority 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 application 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, similarly to FaCT++, TRILL systems manage two queues. Moreover, the management of the assertions is slightly different because once an assertion is considered, every applicable 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 included in *NonDetQueue* (that with "lower priority") and therefore considered later. This means that assertions matching non-deterministic rules are considered only when the *DetQueue* is empty.

6. Experiments

To test the effectiveness of the optimizations introduced in the new version of the TRILL framework, we compared the system TRILL with its old version in four different tests. We report only the results of TRILL, except when differently stated, because they are similar to those of the other systems, namely TRILL^P and TORNADO. In particular, TRILL^P is the reasoner that presents the highest speed-up among the two versions, while for TORNADO the speed-up is slightly lower than that achieved by TRILL. Therefore, TRILL is a good indicator of the results. To facilitate reading, in the following we refer to the two versions of the reasoner using their version number, in particular, the version implementing the optimization presented in this paper is the 6.0.2 while the previous version is the 5.2.2. We will indicate the two versions with *v6* and *v5*, respectively.

The experiments presented here are done by partly following the tests performed Zese *et al.* (2018) [17], where *v5* TRILL systems have been compared with the probabilistic reasoners BORN [59], BUNDLE [13, 60], and PRONTO [61], and with the non-probabilistic reasoners Pellet [1], Konclude[3] [62], 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 *v5* version of TRILL systems, can achieve better results than state-of-the-art (non-)probabilistic reasoners. We refer the reader to the work of Zese *et al.* (2018) [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 Zese *et al.* (2018) [17], in this paper, Test 2 and Test 3 consider only the different implementations of the TRILL framework. To help the reader who wants to compare the results presented in the two articles, below we briefly discuss the similarities and differences with the orig-

[3] `http://derivo.de/produkte/konclude/`
[4] `http://jfact.sourceforge.net/`

Table 1
Information about the KBs used in Test 1.

KB	N. of axioms	N. of Individuals	N. of concepts
BioPAX	926	0	69
BRCA	290	5	153
DBpedia	267	0	118
Vicodi	403	19	168

inal tests [17]. To carry out the comparison among the two versions of the framework, we re-ran Tests 1, 2, and 3, from Zese *et al.* (2018) [17], whose results are discussed in Test 1, Test 2, and Test 3, respectively.

In particular, Test 1 is extended by also answering queries that have no justifications. Moreover, it considers also probabilistic query answering as in Test 4 of Zese *et al.* (2018) [17]. As shown by the results presented by Zese *et al.* (2018) [17], the probability computation does not affect the inference performance, therefore, the speed-up achieved in the two cases should be similar.

We have also added two new tests, namely Test 4, experimenting with the scalability of the *v5* and *v6* version of the TRILL framework, and Test 5, comparing TRILL and TORNADO with PuLi [21].

All tests were run on a Linux machine with Intel® Core™ i7-8565U CPU @ 1.80GHz and 16 GB of RAM.

Test 1 We used four real-world KBs:

- BioPAX level 3[5], which models metabolic pathways.
- BRCA [63], which models the risk factors of breast cancer depending on many factors such as age and drugs taken.
- An extract of DBpedia ontology[6], containing structured information of Wikipedia, usually contained in the information box on the right-hand side of a page.
- Vicodi [64], which contains information on European history and models historical events and important personalities.

The details about the KBs are shown in Table 1. In particular, in two KBs the ABox is absent. In this way we can show how much the implementation of the expansion queue is effective to reduce the inference time. This test is composed of three different tasks. We tested the performance of TRILL when answering (1) queries with at least one justification; (2) queries that have no justifications; and (3) probabilistic queries that have at least one justification. For every task, we ran 50 sub-class queries for each KB, except for Vicodi for which we ran 50 instance-of queries, and we collected the running time.

For task 1, the sub-class queries were created by randomly selecting a class C_1 from the KB. Each class has the same probability to be picked up, hence we sample classes from a uniform probability distribution. Once, selected the first class, a second class C_2 is randomly selected such that the query $Q = C_1 \sqsubseteq C_2$ has at least a justification. The sampling of C_2 is made from the uniform probability distribution $P(C|\textit{at least one justification for } Q)$. To create the instance-of queries, first an individual i was randomly selected from a uniform probability distribution on all the individuals. Then, as for sub-class queries, a class is sampled from the probability distribution $P(C|\textit{at least one justification for } i{:}C)$, which is uniform [17]. In this way, we ensured that each query had at least one explanation, and that no biases are introduced during the query creation. For each query, we asked TRILL to find one justification and the covering set of justifications.

Table 2
Average speed-up for answering queries in Test 1 with the reasoner TRILL in its two versions, *v6* and *v5*. Column "No Just." reports the results considering queries that have no justifications, while the columns "1 Just." and "All Just." report the time that TRILL takes to return one single justification and all the justifications.

Dataset	No Just.	1 Just.	All Just.
BioPAX	2.268	2.243	2.175
BRCA	1578.705	1456.817	1657.876
DBpedia	0.914	0.733	0.892
Vicodi	13.980	12.763	14.960

For task 2, we created queries that have no justifications. To do so, we followed the same steps done for task 1, but in this case, we selected classes that are not super-class or that the individual does not belongs to. In this case, the choice was made randomly as well. Having queries with no justifications, the reasoner must explore the entire search space similarly to the case of finding all the justifications to ensure that there are no ways to prove the query (and so, to find a justification). The average speed-up achieved by TRILL are reported in Table 2, 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 speed-up is computed by the ratio $\frac{\textit{v5-CPU-time}}{\textit{v6-CPU-time}}$. To find the average speed-up, for each query the ratio *v5-CPU-time v6-CPU-time* has been computed.

The results show that the *v6* version of the systems seems to present a little overhead for simple KBs probably due to the pruning of the expansion rules. DBpedia has no individuals and represents a taxonomy, i.e., a hierarchy of simple concepts and complex concepts of the form $C_0 \sqcap \cdots \sqcap C_n$ with C_i simple concepts, $1 \leq n \leq 4$. Therefore, answering query is trivial, with an average running time of magnitude 10^{-3} seconds for both versions. However, when the running time increases, this overhead is compensated by a faster management of the tableau expansion, as shown for BioPAX, and BRCA. These KBs are more complex than DBpedia. In particular, BRCA contains many complex concepts, TRILL *v6*

[5] http://www.biopax.org/
[6] http://dbpedia.org/

Table 3
Average running time for answering queries in Test 1 with the reasoner TRILL in its two versions, *v6* and *v5* ± the standard deviation/minimum ÷ maximum time for answering a query. All times are reported in ms. Section "No Just." reports the results considering queries that have no justifications, while the sections "1 Just." and "All Just." report the time that TRILL takes to return one single justification and all the justifications.

	v5	*v6*
Dataset	No Just.	
BioPAX	$14,780 \pm 11,113/9,980 \div 82,390$	$7,097 \pm 5,616/3,022 \div 32,514$
BRCA	$10696,803 \pm 829,614/10158,338 \div 14032,208$	$13,438 \pm 18,526/3,697 \div 81,401$
DBpedia	$2,690 \pm 0,094/2,529 \div 2,941$	$2,944 \pm 0,101/2,901 \div 3,636$
Vicodi	$50,237 \pm 0,557/49,481 \div 53,344$	$3,625 \pm 0,343/3,097 \div 4,444$
	1 Just.	
BioPAX	$15,009 \pm 11,773/9,750 \div 81,988$	$7,121 \pm 5,670/3,507 \div 32,416$
BRCA	$10695,678 \pm 892,824/10214,796 \div 13726,005$	$15,211 \pm 20,494/4,074 \div 81,071$
DBpedia	$2,403 \pm 0,075/2,268 \div 2,560$	$3,288 \pm 0,161/2,653 \div 3,504$
Vicodi	$50,662 \pm 0,365/49,939 \div 51,454$	$3,994 \pm 0,323/3,520 \div 4,892$
	All Just.	
BioPAX	$14,785 \pm 8,796/9,876 \div 53,916$	$7,664 \pm 6,737/3,175 \div 40,824$
BRCA	$10702,400 \pm 1001,480/10196,159 \div 14737,977$	$14,100 \pm 20,149/3,518 \div 101,518$
DBpedia	$2,423 \pm 0,077/2,299 \div 2,617$	$2,714 \pm 0,015/2,686 \div 2,759$
Vicodi	$50,198 \pm 0,834/49,347 \div 55,215$	$3,373 \pm 0,313/2,896 \div 4,223$

takes 10^{-2} seconds on average, while *v5* needs 10^{1} seconds on average. Differently, Vicodi's TBox is similar to that of DBpedia, containing only a taxonomy of simple concepts, but presents several individuals, which are not all related. The extraction of the fragment of ABox significant for the query makes TRILL *v6* one order of magnitude faster than *v5*.

To ensure that the results reported in Table 2 are significant, we report in Table 3 the average running time in milliseconds (ms) for TRILL *v5* and *v6* with their standard deviation, and the minimum and maximum values for the running time. As one can see, the variability is significant but for the KBs where *v6* achieves higher speed-up (BRCA and Vicodi), the maximum time needed by *v6* is lower than the minimum for *v5*. For BioPAX and DBpedia, for which the speed-up is low, this strong separation between the runtimes is absent. However, in these cases, the average running time is very low while the overhead added by the optimization is more significant, as also shown in Table 2.

Finally, Table 4 shows the minimum and maximum speed-up for the four KBs. The results from Tables 2, 3, and 4 shows that the optimization implemented are more effective for larger KBs but are independent by the query, as the minimum and maximum speed-up are similar to the average.

For task 3, since TRILL performs also probabilistic query answering, we extended this test to this setting. To make the KBs probabilistic, we annotated all the axioms of each KB with a probability value randomly generated. Then we ran the two versions of TRILL to compute the probability of the same queries generated for the "All Just." column of Table 2 in task 1 but asking TRILL to compute the probability of the query together with the set of justifications. Finally, we collected the running times, computed the average speed-up, and compared the results obtained performing probabilistic query answering with those reported in "All Just." column of Table 2. We expect here a smaller speed-up since the time for computing the probability is the same for both versions, as it is not affected by the optimization introduced. However, the highest the running time the highest the speed-up, because as of the running time increases, the computation of the probability become less significant.

Table 4
Minimum ÷ maximum speed-up for answering queries in Test 1 with the reasoner TRILL in its two versions, *v6* and *v5*. Section "No Just." reports the results considering queries that have no justifications, while the sections "1 Just." and "All Just." report the time that TRILL takes to return one single justification and all the justifications.

Dataset	No Just.
BioPAX	$1.357 \div 3.534$
BRCA	$169.282 \div 2776.355$
DBpedia	$0.721 \div 1.012$
Vicodi	$11.254 \div 16.416$
	1 Just.
BioPAX	$1.419 \div 3.071$
BRCA	$168.707 \div 2511.701$
DBpedia	$0.675 \div 0.917$
Vicodi	$10.421 \div 14.330$
	All Just.
BioPAX	$1.321 \div 3.319$
BRCA	$140.575 \div 2901.117$
DBpedia	$0.843 \div 0.970$
Vicodi	$11.841 \div 17.321$

Table 5 shows the results as the average ratio between the CPU running time taken by the two versions of the system for performing probabilistic inference (grey column) and non-probabilistic inference. The final ratio is computed by find-

Table 5
Average speed-up for answering non-probabilistic queries and probabilistic queries with the reasoner TRILL in its two versions, w.r.t. the 4 different KBs of Test 1. The values of the last column are taken from Table 2.

Dataset	Prob. Query Answering	Non-prob. Query Answering
BioPAX	2.187	2.175
BRCA	1522.360	1657.876
DBpedia	0.942	0.892
Vicodi	12.923	14.960

Table 6
Average speed-up for answering non-probabilistic queries and probabilistic queries with the reasoners TRILL^P and TORNADO in their two versions, w.r.t. the 4 different KBs of Test 1.

	TRILL^P		TORNADO	
Dataset	Prob.	Non-prob.	Prob.	Non-prob.
BioPAX	1.995	1.954	2.170	1.886
BRCA	976.970	1107.231	456.723	517.879
DBpedia	0.818	0.689	0.781	0.814
Vicodi	12.190	14.226	16.712	17.147

ing the ration $\frac{v5\text{-}CPU\text{-}time}{v6\text{-}CPU\text{-}time}$ for each query and average them. As the results show, the extra time for the probability computation reduces the speed-up. However, it is worth noting that this time is negligible (of the order of 10^{-3} seconds or even 10^{-4} seconds depending on the KB). Clearly, the optimization presented in this paper does not affect the computation of the probability, as also proved by these results.

For this test, we decided to report the results of TRILL^P and TORNADO because it considers KBs modelling real world domains, therefore they are the most representative of real use case scenarios. Table 6 reports the average speed-up of TRILL^P and TORNADO for probabilistic and non-probabilistic query answering. In particular, TRILL^P and TORNADO directly returns the set of all justifications represented as a Boolean formula or a BDD, therefore, the non-probabilistic setting for these reasoners corresponds with "Non-prob. Query Answering" setting of Table 5 for TRILL.

From Tables 5 and 6 it is possible to see that the speed-up presents similar values for all the three reasoners. TRILL^P achieves the highest speed-up overall w.r.t. the BRCA KB, which is the more difficult to query. This is due to the fact that TRILL^P relies on a SAT-solver [14] for checking the applicability of a tableau expansion rule, therefore, the optimizations presented in this paper result to be more effective.

Test 2 In this second test we consider the artificial KB shown in the following:

$$C_{1,1} \sqsubseteq C_{1,2} \sqsubseteq ... \sqsubseteq C_{1,n} \sqsubseteq C_{n+1}$$
$$C_{1,1} \sqsubseteq C_{2,2} \sqsubseteq ... \sqsubseteq C_{2,n} \sqsubseteq C_{n+1}$$
$$...$$
$$C_{1,1} \sqsubseteq C_{m,2} \sqsubseteq ... \sqsubseteq C_{m,n} \sqsubseteq C_{n+1}$$
$$a : C_{1,1}$$

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

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 m justifications containing $n + 1$ axioms each: n subclass-of axioms and 1 assertion axiom. The choice of this query is made to compute the average speed-up in the worst case, i.e., when answering the most difficult query. Moreover, during the expansion of this KB the differences between the rule calls made by the *v5* and the *v6* 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 *v5* 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 *v6* version can cut a small number of rule tests. Moreover, in these KBs there is only one individual, which is also used in the query. Therefore, the extraction of the relevant fragment of the KB for the query returns the KB itself, further reducing the difference between the execution of the two versions and possibly adding some overhead that may slow down the reasoner. In this test, we want to compute a sort of baseline for the speed-up introduced by the optimizations presented in this paper, in order to also explore the cases where the optimizations increase the running time.

We executed the query Q 100 times to compute the average speed-up of running time that each system in each version needs to compute all the justifications ($\frac{v5\text{-}CPU\text{-}time}{v6\text{-}CPU\text{-}time}$). We varied m and n between 1 to 7. Table 7 reports the results for TRILL. 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 and ranges from 10^{-3} and 10^{-2} seconds. Therefore, in this case the overhead is negligible, as a different load of the CPU may easily change the value of the speed-up.

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. Table 8 shows the ratio *v5-CPU-time*:*v6-CPU-time* for TRILL for increasing m and n.

As expected, the overhead becomes negligible as the size of the KBs increases. Indeed, the larger the KB the higher the speed-up. It is important to notice that the more difficult the application of the rule, the higher the ratio. However, given the fact that this is a hard scenario for both the

Table 7
Average speed-up achieved for computing all the explanations with the reasoner TRILL in its two versions in Test 2. Columns correspond to n while rows correspond to m. n and m vary from 1 to 7 in step of 1.

	1	2	3	4	5	6	7
1	0.623	1.267	1.249	1.153	1.124	0.523	0.519
2	0.279	0.137	0.153	0.170	0.178	0.186	0.195
3	0.122	0.135	0.152	0.164	0.244	0.264	0.206
4	0.119	0.128	0.149	0.219	0.175	0.180	0.182
5	0.114	0.119	0.206	0.157	0.161	0.205	0.206
6	0.102	0.128	0.145	0.145	0.177	0.173	0.174
7	0.093	0.158	0.128	0.179	0.149	0.154	0.159

Table 8
Average speed-up for TRILL in Test 2. Columns correspond to n while rows correspond to m. n and m vary from 10 to 100 in step of 10.

	10	20	30	40	50	60	70	80	90	100
10	0.959	0.972	0.996	0.981	1.035	1.058	1.077	1.078	1.112	1.127
20	1.104	1.120	1.150	1.167	1.188	1.216	1.230	1.241	1.271	1.300
30	1.257	1.283	1.287	1.327	1.322	1.347	1.377	1.400	1.435	1.442
40	1.393	1.416	1.441	1.451	1.488	1.510	1.526	1.555	1.584	1.602
50	1.529	1.567	1.574	1.614	1.642	1.664	1.687	1.715	1.734	1.784
60	1.688	1.721	1.756	1.776	1.786	1.820	1.832	1.867	1.893	1.924
70	1.807	1.854	1.872	1.912	1.939	1.977	1.999	2.025	2.053	2.083
80	1.948	2.003	2.009	2.056	2.081	2.127	2.157	2.187	2.219	2.238
90	2.073	2.127	2.144	2.202	2.243	2.285	2.307	2.343	2.348	2.385
100	2.208	2.263	2.302	2.359	2.387	2.432	2.459	2.481	2.506	2.532

optimizations (one is not able to reduce the size of the tableau, while the other removes only few rule applications), it is clear that TRILL benefits from optimizations, increasing more and more the speed-up compared to the previous version. For example, it reduces the running time from ≈ 508 seconds to ≈ 210 seconds for the KB with $n = m = 100$.

Test 3 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

$$(C_{1,i})\ B_{i-1} \sqsubseteq P_i \sqcap Q_i$$
$$(C_{2,i})\ P_i \sqsubseteq B_i$$
$$(C_{3,i})\ Q_i \sqsubseteq B_i$$

with $1 \leq i \leq n$, n an integer, $n \geq 1$. The query $Q = B_0 \sqsubseteq B_n$ has 2^n explanations, even if the KB has a size that is linear in n. 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 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 speed-up (computed as $\frac{v5\text{-}CPU\text{-}time}{v6\text{-}CPU\text{-}time}$), shown in Table 9. As in Test 6, we used the most difficult query to answer, i.e., that with the highest number of justifications.

Table 9
Average speed-up for answering queries with the reasoner TRILL in its two versions for the KBs of Test 3 with increasing n.

n	Speed-up
2	1.200
4	2.181
6	4.654
8	5.893
10	6.149

The results show that the *v6* version of the TRILL framework outperforms the *v5* version. These results clearly show that the optimizations implemented in the systems, and especially the management of the expansion rules, can sensibly improve the performances. For the sake of completeness, TRILL *v5*'s running time with $n = 10$ was around 1853 seconds ($\approx$ 30 minutes), while the *v6* needed around 316 seconds ($\approx$ 5 minutes).

Test 4 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)[7]. Similarly to Test

[7] http://si.washington.edu/projects/fma

Table 10
Average speed-up for answering queries with TRILL, TRILL^P and TORNADO in their two versions, w.r.t. KBs of increasing size based on FMA (Test 4). The first two columns show respectively the number of individuals and the number of axioms contained in each KB.

N. Ind	N. axioms	Speed-up TRILL	TRILL^P	TORNADO
10	4788	5.102	9.756	5.219
20	4830	24.395	42.332	27.279
30	4895	81.983	189.893	86.758
40	4967	62.448	103.434	61.925
50	5020	93.454	185.739	96.058
60	5082	206.092	344.537	207.968
70	5124	289.742	376.721	283.732
80	5220	747.405	1314.447	785.136
90	5244	736.156	1225.007	708.866
100	5354	752.150	1262.428	793.740
200	6008	9930.380	14329.290	10275.189
300	6650	28029.388	33471.564	26616.672

1, in this test we consider TRILL^P and TORNADO as well, because also in this case we are using a KB modelling a real-world domain. 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 a number of individuals varying from 10 to 100 with step of 10, 200 and 300, each of them described with 1 to 11 assertions. Each assertion was created by randomly selecting sub-classes of the concept "Organ_zone", which is the super-class of the most concepts in the KB. We ran 10 queries w.r.t. each KB looking for the justifications about the individual belonging to "Organ_zone" and collected the CPU time and computed the average speed-up, reported in Table 10. This test is specifically made for testing mainly the optimization regarding the extraction of the relevant fragment of the KB. From the results, we can see that this optimization can dramatically improve the performance of the reasoners. This is due to the fact the standard tableau, implemented in TRILL *v5*, initializes the tableau with every individual of the KB. However, as discussed in Section 4.1, this is detrimental for query answering as it is useless considering individuals not interested by the query.

Test 5 This test aims at comparing TRILL and TORNADO with the reasoner PuLi [21], which achieved impressive performance in finding the set of justifications in very large $\mathcal{EL}$ KBs, such as GO-PLUS[8], GALEN version 7 of OpenGALEN[9] and SNOMED CT version 2015-01-31[10].

The tableau algorithm is at a disadvantage compared to the strategy used by PuLi to find the justifications in terms of complexity. However, PuLi supports $\mathcal{EL}$ DL, language adopted by the KBs considered in this test and can answer only subsumption queries. On the other hand, the tableau algorithm implemented in TRILL framework consider DLs significantly more expressive, forcing the implementation of sophisticated mechanism to check the applicability of the expansion rules. Moreover, the reasoners in the TRILL framework can answer a larger number of queries, spanning from instance check to find justification for two individuals being related by a certain role. Therefore, to create a fairer condition, we use TORNADO, which applies the tableau expansion rules specific to $\mathcal{SHI}$ DL, which is more expressive than $\mathcal{EL}$, but less then $\mathcal{SHIQ}$ DL considered by TRILL.

This test is divided in two tasks comparing TRILL and TORNADO with PuLi w.r.t.: (1) GO-PLUS and GALEN, and (2) the KBs from Test 2 and Test 3.

For task 1, we run TRILL and TORNADO under the same settings imposed to test PuLi. As described by its authors [21], PuLi have been compared with SAT-based methods, running three different strategies, called *Bottom-up*, *Top-Down*, and *Threshold* paired with the reasoner ELK [65] to extract inferences. To carry on the test, the three KBs have been pre-processed to remove non-$\mathcal{EL}$ axioms, in order to allow PuLi to manage them. After this pre-process, the set of entailed direct subsumptions between atomic concepts have been extracted and used as queries. The test aimed at checking for how many queries the set of all justifications were computed within 1 hour global timeout and imposing a timeout of 60 seconds for each single query.

In this paper, we consider only GO-PLUS and GALEN because we have not been able to get a copy of SNOMED CT. GO-PLUS, after the pre-process described above, contains 105557 axioms and 90443 queries have been collected. As for GALEN, it contains 44475 axioms and 91332 queries have been extracted.

Table 11 shows the results in terms of number of queries attempted in 1 hour, the number of queries that timed out, the percentage of queries attempted on the total number of possible queries, and the percentage of timed-out queries on the number of those attempted within 1 hour achieved by TRILL and TORNADO *v6*, and PuLi using its three searching strategies, *Bottom-Up*, *Top-Down*, and *Threshold* with the reasoner ELK [65] (respectively PBU, PTD and PT in Table 11). As can be seen, TRILL and TORNADO achieve similar results and are outperformed by PuLi, showing the limitations of the tableau algorithms with respect to resolution with answer literals [22] used by PuLi. In this test we do not report the results achieved by TRILL and TORNADO *v5* because they were not able to solve any query within the time out of 60 seconds.

It is also important to note that, by using the library Thea2 to load the KBs, the systems in TRILL framework are very slow. The loading of the KBs takes a dozen of minutes. This is clearly a problem that will be addressed in the next optimizations.

For task 2, we run PuLi on the KBs of Test 2 with $m = 100$ and of Test 3, using the same settings described in the

[8] http://geneontology.org/page/download-ontology
[9] https://www.opengalen.org/sources/sources.html
[10] https://www.snomed.org/

Table 11
Number of queries attempted in 1h / number of 60s timeouts / % of attempted queries in the number of all queries / % of 60s timeouts in the number of queries attempted in 1h of TRILL and TORNADO (TORN.) *v6*, and PuLi *Bottom-Up* (PBT), *Top-Down* (PTD), and *Threshold* (PT) for Test 5.

System	GO-PLUS	GALEN
PBU	3537/43/3.91/1.22	10865/25/11.90/0.23
PTD	20335/46/22.48/0.23	5096/36/5.58/0.71
PT	23147/38/25.59/0.16	42395/23/46.42/0.05
TORN.	90/54/0.10/60.00	68/60/0.07/88.41
TRILL	87/54/0.10/62.07	65/60/0.07/92.31

Table 12
Average running time in s for answering queries with the reasoners TRILL and TORNADO (TORN.) *v6*, and PuLi *Bottom-Up* (PBT), *Top-Down* (PTD), and *Threshold* (PT) for the KBs of Test 2 with $m = 100$ and increasing n (Test 5).

n	TORN.	TRILL	PBU	PTD	PT
10	0.902	0.898	0.056	0.083	0.449
20	3.273	3.242	0.226	0.079	0.130
30	7.266	7.246	0.375	0.163	0.245
40	13.398	13.844	0.180	0.318	0.336
50	19.707	19.797	0.196	0.261	0.172
60	28.469	28.656	0.385	0.376	0.256
70	38.789	39.195	0.437	0.473	0.394
80	53.449	50.887	1.650	1.021	1.528
90	69.167	71.465	1.225	1.204	2.739
100	79.609	86.260	0.893	2.026	4.066

two tests. We consider only the KBs with $m = 100$ from Test 2 because they are those for which the running time is higher. Moreover, all the instance-of queries of Test 2 have been replaced with sub-class queries $Q = C_{1,1} \sqsubseteq C_{n+1}$ because PuLi cannot answer instance-of queries. Table 12 reports the average running time in s for TRILL and TORNADO *v6*, and PuLi on the KBs of Test 2, while Table 13 the average running time in ms w.r.t. the KBs of Test 3. The results show that in some cases TORNADO is competitive even with systems applying resolution calculus with answer literals on KBs defined following DLs languages with lower expressivity, such as PuLi. As regards this task, TRILL performs worse than TORNADO with the increase of the number of justifications, showing that the use of the BDDs can significantly improve the performances of the reasoner [17]. Moreover, TORNADO can scale better than PuLi equipped with ELK [65] in some cases, showing its effectiveness in certain scenarios. These results are probably due to the importance of the management of non-deterministic rules for the KBs of Test 3, since the number of choice points increases exponentially in this KB.

Table 13
Average running time in ms for answering queries with the reasoners TRILL and TORNADO (TORN.) *v6*, and PuLi *Bottom-Up* (PBT), *Top-Down* (PTD), and *Threshold* (PT) for the KBs of Test 3 with increasing n (Test 5).

n	TORN.	TRILL	PBU	PTD	PT
2	15.625	6.250	5.803	7.414	5.808
4	14.063	1.563	4.127	4.573	4.850
6	10.938	60.938	11.968	19.290	19.439
8	21.875	2773.438	34.347	53.318	52.566
10	15.625	175421.875	178.265	345.538	168.204

7. Conclusions and Future Directions

In this paper we present two extensions implemented in the TRILL framework, which contains three systems for reasoning on DISPONTE KBs: (i) TRILL, which is able to collect the set of all justifications and compute the probability of queries, (ii) TRILL^P, which implements in Prolog the tableau algorithm [15, 16] for returning the pinpointing formula instead of the set of justifications, and (iii) TORNADO [17], 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 individuals that are meaningful for the execution of the query. The second optimization first selects only the expansion rules that are applicable on the tableau built for answering the query. Then, it 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 experimentation performed shows that the presented optimizations can significantly speed up both regular and probabilistic queries in many cases. The results show that the optimizations presented introduce a small overhead, visible only w.r.t. simpler KBs, usually with no individuals or with all the individuals related, or when the time needed for the inference is far under one second. However, in exchange for this overhead, the optimizations allow to achieve a speed-up of up to more than 26000:1 in favourable cases and almost 6:1 in unfavourable cases.

We are also studying the possibility of 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 reasoner, BUNDLE [12]. The reasoners 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 [41, 42] which translate the KB into Datalog± or by implementing the abductive tableau presented by Cucala *et al.* (2000) [66]. The framework will also contain a crawler for linked data, called KRaider [67], 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 can be changed by the user, some rules may be removed by the initial pruning, but this operation does not change their order. However, it would be easy to implement the possibility for TRILL to automatically set a different order depending on some heuristics. This could be another possible future work.

References

[1] E. Sirin, B. Parsia, B. Cuenca-Grau, A. Kalyanpur, Y. Katz, Pellet: A practical OWL-DL reasoner, J. Web Semant. 5 (2) (2007) 51–53.

[2] R. Shearer, B. Motik, I. Horrocks, HermiT: A highly-efficient OWL reasoner, in: C. Dolbear, A. Ruttenberg, U. Sattler (Eds.), Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions, collocated with the 7th International Semantic Web Conference (ISWC-2008), Karlsruhe, Germany, October 26-27, 2008, Vol. 432 of CEUR Workshop Proceedings, CEUR-WS.org, 2008.

[3] D. Tsarkov, I. Horrocks, FaCT++ description logic reasoner: System description, in: U. Furbach, N. Shankar (Eds.), Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, Vol. 4130 of Lecture Notes in Computer Science, Springer, 2006, pp. 292–297. doi:10.1007/11814771_26.
URL https://doi.org/10.1007/11814771_26

[4] C. Lutz, L. Schröder, Probabilistic Description Logics for subjective uncertainty, in: F. Lin, U. Sattler, M. Truszczynski (Eds.), 12th International Conference on Principles of Knowledge Representation and Reasoning (KR 2010), AAAI Press, Menlo Park, CA, USA, 2010, pp. 393–403.

[5] M. Jaeger, Probabilistic reasoning in terminological logics, in: J. Doyle, E. Sandewall, P. Torasso (Eds.), 4th International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufmann, 1994, pp. 305–316.

[6] D. Koller, A. Y. Levy, A. Pfeffer, P-CLASSIC: A tractable probabilistic description logic, in: B. Kuipers, B. L. Webber (Eds.), Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island., AAAI Press / The MIT Press, 1997, pp. 390–397.

[7] Z. Ding, Y. Peng, A probabilistic extension to ontology language OWL, in: R. H. Sprague, Jr. (Ed.), 37th Hawaii International Conference on System Sciences (HICSS-37 2004), CD-ROM / Abstracts Proceedings, 5-8 January 2004, Big Island, HI, USA, IEEE Computer Society, 2004, pp. 1–10.

[8] T. Lukasiewicz, Expressive probabilistic description logics, Artif. Intell. 172 (6-7) (2008) 852–883.

[9] F. Riguzzi, E. Bellodi, E. Lamma, R. Zese, Probabilistic description logics under the distribution semantics, Semant. Web 6 (5) (2015) 447–501. doi:10.3233/SW-140154.

[10] R. Zese, Probabilistic Semantic Web: Reasoning and Learning, Vol. 28 of Studies on the Semantic Web, IOS Press, Amsterdam, 2017. doi:10.3233/978-1-61499-734-4-i.
URL http://ebooks.iospress.nl/volume/probabilistic-semantic-web-reasoning-and-learning

[11] T. Sato, A statistical learning method for logic programs with distribution semantics, in: L. Sterling (Ed.), Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995, MIT Press, 1995, pp. 715–729.

[12] G. Cota, F. Riguzzi, R. Zese, E. Bellodi, E. Lamma, A modular inference system for probabilistic description logics, in: D. Ciucci, G. Pasi, B. Vantaggi (Eds.), Scalable Uncertainty Management 12th International Conference, SUM 2018, Milan, Italy, October 3-5, 2018, Proceedings, Vol. 11142 of Lecture Notes in Computer Science, Springer, Heidelberg, Germany, 2018, pp. 78–92. doi:10.1007/978-3-030-00461-3_6.
URL http://mcs.unife.it/~friguzzi/Papers/CotRigZes-SUM18.pdf

[13] F. Riguzzi, E. Lamma, E. Bellodi, R. Zese, BUNDLE: A reasoner for probabilistic ontologies, in: W. Faber, D. Lembo (Eds.), 7th International Conference on Web Reasoning and Rule Systems (RR 2013), Mannheim, Germany, Vol. 7994 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 183–197. doi:10.1007/978-3-642-39666-3_14.

[14] R. Zese, E. Bellodi, F. Riguzzi, G. Cota, E. Lamma, Tableau reasoning for description logics and its extension to probabilities, Ann. Math. Artif. Intell. 82 (1-3) (2018) 101–130. doi:10.1007/s10472-016-9529-3.
URL http://dx.doi.org/10.1007/s10472-016-9529-3f

[15] F. Baader, R. Peñaloza, Automata-based axiom pinpointing, J. Autom. Reasoning 45 (2) (2010) 91–129.

[16] F. Baader, R. Peñaloza, Axiom pinpointing in general tableaux, J. Logic Comput. 20 (1) (2010) 5–34.

[17] R. Zese, E. Bellodi, G. Cota, F. Riguzzi, E. Lamma, Probabilistic DL reasoning with pinpointing formulas: A Prolog-based approach, Theor. Pract. Log. Prog. (2018) 1–28doi:10.1017/S1471068418000480.

[18] I. Horrocks, U. Sattler, A tableau decision procedure for $\mathcal{SHOIQ}$, J. Autom. Reasoning 39 (3) (2007) 249–276.

[19] I. Horrocks, O. Kutz, U. Sattler, The even more irresistible SROIQ, in: P. Doherty, J. Mylopoulos, C. A. Welty (Eds.), Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning, Lake District of the United Kingdom, June 2-5, 2006, AAAI Press, 2006, pp. 57–67.
URL http://dl.acm.org/citation.cfm?id=3029947.3029959

[20] E. Bellodi, E. Lamma, F. Riguzzi, R. Zese, G. Cota, A web system for reasoning with probabilistic OWL, Softw. Pract. Exper. 47 (1) (2017) 125–142.

[21] Y. Kazakov, P. Skocovský, Enumerating justifications using resolution, in: D. Galmiche, S. Schulz, R. Sebastiani (Eds.), Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Vol. 10900 of Lecture Notes in Computer Science, Springer, 2018, pp. 609–626. doi:10.1007/978-3-319-94205-6_40.
URL https://doi.org/10.1007/978-3-319-94205-6_40

[22] C. Green, Theorem proving by resolution as a basis for question-answering systems, Mach. Intell. 4 (1969) 183–205.

[23] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider (Eds.), The Description Logic Handbook: Theory, Implementation, and Applications, Cambridge University Press, New York, NY, USA, 2003.

[24] F. Baader, I. Horrocks, U. Sattler, Description Logics, Elsevier, Amsterdam, 2008, Ch. 3, pp. 135–179.

[25] S. Bechhofer, I. Horrocks, F. Patel-Schneider, P, Tutorial on OWL (2003).
URL http://www.cs.man.ac.uk/~horrocks/ISWC2003/Tutorial/

[26] I. Horrocks, U. Sattler, A description logic with transitive and inverse roles and role hierarchies, J. Logic Comput. 9 (3) (1999) 385–410.

[27] I. Horrocks, U. Sattler, S. Tobies, Practical reasoning for expressive description logics, CoRR cs.LO/0005014 (2000).
URL http://arxiv.org/abs/cs.LO/0005014

[28] I. Horrocks, U. Sattler, S. Tobies, Practical reasoning for expressive description logics, in: Logic for Programming and Automated Reasoning, Vol. 99, Springer, 1999, pp. 161–180.

[29] S. Schlobach, R. Cornet, Non-standard reasoning services for the debugging of description logic terminologies, in: G. Gottlob, T. Walsh (Eds.), IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003, Morgan Kaufmann Publishers Inc., San Francisco, CA,

USA, 2003, pp. 355–362.

[30] A. Kalyanpur, Debugging and repair of OWL ontologies, Ph.D. thesis, The Graduate School of the University of Maryland (2006).

[31] F. Baader, R. Peñaloza, B. Suntisrivaraporn, Pinpointing in the description logic $\mathcal{EL}^+$, Appl. Artif. Intell. (2007) 52.

[32] M. Horridge, B. Parsia, U. Sattler, The OWL explanation workbench: A toolkit for working with justifications for entailments in OWL ontologies (2009).

[33] C. Halaschek-Wiener, A. Kalyanpur, B. Parsia, Extending tableau tracing for ABox updates, Tech. rep., University of Maryland (2006).

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

[35] B. Beckert, J. Posegga, leanTAP: Lean tableau-based deduction, J. Autom. Reasoning 15 (3) (1995) 339–358.

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

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

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

[39] R. Zese, E. Bellodi, E. Lamma, F. Riguzzi, A description logics tableau reasoner in prolog, in: D. Cantone, M. N. Asmundo (Eds.), Proceedings of the 28th Italian Conference on Computational Logic, Catania, Italy, Vol. 1068 of CEUR Workshop Proceedings, CEUR-WS.org, 2013, pp. 33–47.
URL http://ceur-ws.org/Vol-1068/paper-l02.pdf

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

[41] M. Gavanelli, E. Lamma, F. Riguzzi, E. Bellodi, R. Zese, G. Cota, An abductive framework for datalog± ontologies, in: M. D. Vos, T. Eiter, Y. Lierler, F. Toni (Eds.), Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Vol. 1433 of CEUR Workshop Proceedings, CEUR-WS.org, 2015.

[42] M. Gavanelli, E. Lamma, F. Riguzzi, E. Bellodi, R. Zese, G. Cota, Abductive logic programming for Datalog± ontologies, in: D. Ancona, M. Maratea, V. Mascardi (Eds.), Proceedings of the 30th Italian Conference on Computational Logic (CILC2015), Genova, Italy, 1-3 July 2015, no. 1459 in CEUR Workshop Proceedings, Sun SITE Central Europe, Aachen, Germany, 2015, pp. 128–143.
URL http://ceur-ws.org/Vol-1459/paper21.pdf

[43] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, SWI-Prolog, Theor. Pract. Log. Prog. 12 (1-2) (2012) 67–96. doi:10.1017/S1471068411000494.

[44] V. Vassiliadis, J. Wielemaker, C. Mungall, Processing OWL2 ontologies using thea: An application of logic programming, in: Proceedings of the 6th International Workshop on OWL: Experiences and Directions, Vol. 529 of CEUR Workshop Proceedings, CEUR-WS.org, 2009.

[45] F. Baader, U. Sattler, An overview of tableau algorithms for description logics, Studia Logica 69 (1) (2001) 5–40. doi:10.1023/A:1013882326814.
URL https://doi.org/10.1023/A:1013882326814

[46] D. Tsarkov, I. Horrocks, Ordering heuristics for description logic reasoning, in: L. P. Kaelbling, A. Saffiotti (Eds.), 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), Professional Book Center, 2005, pp. 609–614.
URL http://ijcai.org/Proceedings/05/Papers/1352.pdf

[47] D. Lembo, M. Lenzerini, R. Rosati, M. Ruzzi, D. F. Savo, Inconsistency-tolerant semantics for description logics, in: P. Hitzler, T. Lukasiewicz (Eds.), Web Reasoning and Rule Systems - Fourth International Conference, RR 2010, Bressanone/Brixen, Italy, September 22-24, 2010. Proceedings, Vol. 6333 of Lecture Notes in Computer Science, Springer, 2010, pp. 103–117. doi:10.1007/978-3-642-15918-3_9.
URL https://doi.org/10.1007/978-3-642-15918-3_9

[48] M. Bienvenu, C. Bourgaux, Inconsistency-tolerant querying of description logic knowledge bases, in: J. Z. Pan, D. Calvanese, T. Eiter, I. Horrocks, M. Kifer, F. Lin, Y. Zhao (Eds.), Reasoning Web: Logical Foundation of Knowledge Graph Construction and Query Answering - 12th International Summer School 2016, Aberdeen, UK, September 5-9, 2016, Tutorial Lectures, Vol. 9885 of Lecture Notes in Computer Science, Springer, 2016, pp. 156–202. doi:10.1007/978-3-319-49493-7_5.
URL https://doi.org/10.1007/978-3-319-49493-7_5

[49] B. C. Grau, I. Horrocks, Y. Kazakov, U. Sattler, Modular reuse of ontologies: Theory and practice, J. Artif. Intell. Res. 31 (2008) 273–318. doi:10.1613/jair.2375.
URL https://doi.org/10.1613/jair.2375

[50] B. Konev, C. Lutz, D. Walther, F. Wolter, Semantic modularity and module extraction in description logics, in: M. Ghallab, C. D. Spyropoulos, N. Fakotakis, N. M. Avouris (Eds.), ECAI 2008 - 18th European Conference on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proceedings, Vol. 178 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2008, pp. 55–59. doi:10.3233/978-1-58603-891-5-55.
URL https://doi.org/10.3233/978-1-58603-891-5-55

[51] P. Doran, V. A. M. Tamma, L. Iannone, Ontology module extraction for ontology reuse: an ontology engineering perspective, in: M. J. Silva, A. H. F. Laender, R. A. Baeza-Yates, D. L. McGuinness, B. Olstad, Ø. H. Olsen, A. O. Falcão (Eds.), Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007, ACM Press, 2007, pp. 61–70. doi:10.1145/1321440.1321451.
URL https://doi.org/10.1145/1321440.1321451

[52] B. C. Grau, I. Horrocks, Y. Kazakov, U. Sattler, Just the right amount: extracting modules from ontologies, in: C. L. Williamson, M. E. Zurko, P. F. Patel-Schneider, P. J. Shenoy (Eds.), Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007, ACM Press, 2007, pp. 717–726. doi:10.1145/1242572.1242669.
URL https://doi.org/10.1145/1242572.1242669

[53] C. D. Vescovo, B. Parsia, U. Sattler, T. Schneider, The modular structure of an ontology: Atomic decomposition, in: T. Walsh (Ed.), 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011), AAAI Press/IJCAI, 2011, pp. 2232–2237. doi:10.5591/978-1-57735-516-8/IJCAI11-372.
URL https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-372

[54] J. Wu, A. K. Hudek, D. Toman, G. E. Weddell, Absorption for aboxes, in: Y. Kazakov, D. Lembo, F. Wolter (Eds.), Proceedings of the 2012 International Workshop on Description Logics (DL2012), Rome, Italy, June 7-10, 2012, Vol. 846 of CEUR Workshop Proceedings, CEUR-WS.org, 2012.
URL http://ceur-ws.org/Vol-846/paper_34.pdf

[55] J. Wu, T. Kinash, D. Toman, G. E. Weddell, Absorption for aboxes and tboxes with general value restrictions, in: B. Pfahringer, J. Renz (Eds.), AI 2015: Advances in Artificial Intelligence - 28th Australasian Joint Conference, Canberra, ACT, Australia, November 30 - December 4, 2015, Proceedings, Vol. 9457 of Lecture Notes in Computer Science, Springer, 2015, pp. 609–622.
URL https://doi.org/10.1007/978-3-319-26350-2_54

[56] I. Horrocks, S. Tobies, Optimisation of terminological reasoning, in: F. Baader, U. Sattler (Eds.), Proceedings of the 2012 International Workshop on Description Logics (DL2012), Rome, Italy, June 7-10, 2012, Vol. 33 of CEUR Workshop Proceedings, CEUR-WS.org, 2000, pp. 183–192.
URL http://ceur-ws.org/Vol-33/Horrocks183-192.ps

[57] I. Horrocks, S. Tobies, Reasoning with axioms: Theory and practice, in: A. G. Cohn, F. Giunchiglia, B. Selman (Eds.), KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000, Morgan Kaufmann, 2000, pp. 285–296.

[58] D. Tsarkov, I. Horrocks, Efficient reasoning with range and domain

constraints, in: V. Haarslev, R. Möller (Eds.), Proceedings of the 2004 International Workshop on Description Logics (DL2004), Whistler, British Columbia, Canada, June 6-8, 2004, Vol. 104 of CEUR Workshop Proceedings, CEUR-WS.org, 2004.

[59] İ. İ. Ceylan, J. Mendez, R. Peñaloza, The bayesian ontology reasoner is born!, in: M. Dumontier, B. Glimm, R. S. Gonçalves, M. Horridge, E. Jiménez-Ruiz, N. Matentzoglu, B. Parsia, G. B. Stamou, G. Stoilos (Eds.), Informal Proceedings of the 4th International Workshop on OWL Reasoner Evaluation (ORE-2015) co-located with the 28th International Workshop on Description Logics (DL 2015), Vol. 1387 of CEUR Workshop Proceedings, CEUR-WS.org, 2015, pp. 8–14.

[60] F. Riguzzi, E. Bellodi, E. Lamma, R. Zese, Reasoning with probabilistic ontologies, in: Q. Yang, M. Wooldridge (Eds.), 24th International Joint Conference on Artificial Intelligence (IJCAI 2015), AAAI Press/International Joint Conferences on Artificial Intelligence, Palo Alto, California USA, 2015, pp. 4310–4316.

[61] P. Klinov, Pronto: A non-monotonic probabilistic description logic reasoner, in: S. Bechhofer, M. Hauswirth, J. Hoffmann, M. Koubarakis (Eds.), The Semantic Web: Research and Applications, 5th European Semantic Web Conference, ESWC 2008, Tenerife, Canary Islands, Spain, June 1-5, 2008, Proceedings, Vol. 5021 of Lecture Notes in Computer Science, Springer, 2008, pp. 822–826.

[62] A. Steigmiller, T. Liebig, B. Glimm, Konclude: System description, J. Web Semant. 27 (2014) 78–85.

[63] P. Klinov, B. Parsia, Optimization and evaluation of reasoning in probabilistic description logic: Towards a systematic approach, in: A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, K. Thirunarayan (Eds.), The Semantic Web - ISWC 2008, 7th International Semantic Web Conference, ISWC 2008, Karlsruhe, Germany, October 26-30, 2008. Proceedings, Vol. 5318 of Lecture Notes in Computer Science, Springer, 2008, pp. 213–228.

[64] G. Nagypál, R. Deswarte, J. Oosthoek, Applying the semantic web: The VICODI experience in creating visual contextualization for history, Lit. Linguist. Comput. 20 (3) (2005) 327–349. doi:10.1093/llc/fqi037.
URL https://doi.org/10.1093/llc/fqi037

[65] Y. Kazakov, P. Klinov, Goal-directed tracing of inferences in EL ontologies, in: P. Mika, T. Tudorache, A. Bernstein, C. Welty, C. A. Knoblock, D. Vrandecic, P. Groth, N. F. Noy, K. Janowicz, C. A. Goble (Eds.), The Semantic Web - ISWC 2014 - 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II, Vol. 8797 of Lecture Notes in Computer Science, Springer, 2014, pp. 196–211. doi:10.1007/978-3-319-11915-1_13.
URL https://doi.org/10.1007/978-3-319-11915-1_13

[66] D. T. Cucala, B. C. Grau, I. Horrocks, Consequence-based reasoning for description logics with disjunction, inverse roles, number restrictions, and nominals, in: J. Lang (Ed.), 27th International Joint Conference on Artificial Intelligence (IJCAI 2018), ijcai.org, 2018, pp. 1970–1976. doi:10.24963/ijcai.2018/272.
URL https://doi.org/10.24963/ijcai.2018/272

[67] G. Cota, F. Riguzzi, R. Zese, E. Lamma, Kraider: a crawler for linked data, in: A. Casagrande, E. G. Omodeo (Eds.), Proceedings of the 34th Italian Conference on Computational Logic, Trieste, Italy, Vol. 2396 of CEUR Workshop Proceedings, CEUR-WS.org, 2019, pp. 202–216.
URL http://ceur-ws.org/Vol-2396/paper35.pdf

[68] A. Kalyanpur, B. Parsia, B. C. Grau, E. Sirin, Justifications for entailments in expressive description logics, Tech. rep., Technical report, University of Maryland Institute for Advanced Computer (2006).

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 $rule\_name(TabIn, TabList)$, while, in the new version, as seen in Section 4.2, they follow the interface $rule\_name(TabIn, AssToExpand, TabList)$. Figure 8 shows the code of the non-deterministic rule $\rightarrow \sqcup$ 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 tableau 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 returned 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 also consider the choice points occurred during their construction. For a detailed description of these aspects, we refer to [68].

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 function value of the assertion present in `A0` does not contain already the justifications introduced by the application of the rule, e.g., there is no justification (which is a set of axioms) that is a subset of the axioms contained in the new justification to add. Otherwise, the predicate fails, and the tableau is not updated. Clearly, if the assertion considered by the rule is not present in the ABox, then the rule is applicable. If the rule can be applied, the predicate `modify_ABox/5` updates the list of assertions `A0` creating `A`. Calling `modify_ABox/5` avoids infinite loops in the

```
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),!.
```

Figure 8: Code of the $\rightarrow \sqcup$ 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 `scan_or_list/8` called in the upper frame, shown in Figure 9, creates the list of the tableaux or fails. Similarly, the predicate `scan_or_list/9` also updated the expansion queues.

```
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).
```

Figure 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 `LC` with the explanation `Expl` to the tableau `A0` creating a new tableau for each concept in the disjunction. 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.

rule application. The predicate implemented in the new version of the framework simply takes as input also the pair of expansion queues `ExpansionPairQ0` (see Figure 8) which is then passed to `modify_ABox/7` predicate. This is similar to `modify_ABox/5` but in addition it also updates the expansion queues by adding the new assertions included in the new ABox.

Basically, the two new arguments considered by this predicate are `ExpansionPairQ0`, the initial pair of expansion queues, and `ExpansionPairQ`, the updated pair of expansion queues. If `modify_ABox` updates the tableau, it collects all the newly added assertions and all the assertions whose tracing function has been updated in `AssToAdd`. Then, it removes from `ExpansionPairQ0` all the assertions in `AssToAdd` creating `ExpansionPairQ1`. Finally, it adds to `ExpansionPairQ1` the assertions in `AssToAdd` each one at the end of the correct queue, the choice of the queue depends on the type of each assertion. This will ensure that every assertion appears once in the pair of the expansion queues (i.e., in one single queue and once in that queue) and delays the expansion of newly added assertions in order to expand the tableau in a bottom-up manner starting from the assertions contained in the KB.

A.2. Deterministic Rules

In the old version of the TRILL framework, deterministic rules are defined following the interface *rule_name*(*TabIn*, *TabOut*) that, given the current tableau *TabIn*, returns the tableau *TabOut* obtained by the application of the rule to *TabIn*. In the new version, the rules also take the assertion to expand. Figure 10 shows part of the code of the deterministic rule $\rightarrow$ *unfold*. As before, in the top frame the code of the original implementation is shown while the new implementation is shown in the bottom frame. The predicate `unfold_rule/2` (top) searches in *TabIn* = `(A0,T)` for an individual to which the rule can be applied. On the other hand, the predicate `unfold_rule/3` (bottom) extracts exactly the assertion taken in input. Then the predicate `find_sub_sup_class/3` is called 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 apply the rule easily and quickly. 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 tracing function by means of `and_f/3` predicate, which conjoins the explanation `Expl0` with the axiom `Ax`. In the new version, instead of looking

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

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

```
unfold_rule((A0,T)-ExpansionPairQ0,(C,Ind),(A,T)-ExpansionPairQ):-
  findClassAssertion(C,Ind,Expl,A0),
  find_sub_sup_class(C,D,AxExpl),
  and_f(AxExpl,Expl,AxL),
  modify_ABox(A0,ExpansionPairQ0,D,Ind,AxL,A,ExpansionPairQ).

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

Figure 10: Code of the → *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 (`AxL`) for the new class assertion found (by using `and_f/3`) and adds it to the tableau to update it. The predicate `and_f/3` performs a conjunction of two explanations.

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 `and_f/3` conjoins the explanation `Expl0` 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 Figures 2 and 3.

Figure 11 shows the Prolog code of the `apply_all_rules/2`. The Prolog implementation of `apply_all_rules/2` replaces the do-while loop reported in Figure 2 with recursion. Note that, this part of Prolog code is slightly different from what shown in the pseudo-code of Figure 3, where `apply_det_rules` and `apply_nondet_rules` are called by APPLY_ONE_RULE. Basically, APPLY_ONE_RULE is replaced by `apply_det_rules` and `apply_nondet_rules` procedures, while APPLY_DET_RULES and APPLY_NONDET_RULES of Figure 3 are replaced with the Prolog's built-in predicate `call`. However, these differences do not change the results, it is just a matter of implementation.

Consider the code in Figure 11, `apply_all_rules/2`, once called, tries to expand the tableau by means of the expansion rules, calling `apply_det_rules/3`. This predicate takes as input the list of deterministic rules and the current tableau, tries to apply an expansion rule by means of procedure `call`, 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.

Procedure `apply_nondet_rules` 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 to

```
apply_all_rules(TabIn,TabOut):-
  setting_trill(det_rules,DetRules),
  apply_det_rules(DetRules,TabIn,TabTemp),
  (is_null(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),!.                    % NOTE: here the deterministic expansion rules are called.

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),!,                         % NOTE: here the non-deterministic expansion rules are called.
  member(TabOut,L),
  dif(TabIn,TabOut).

apply_nondet_rules([_|T],TabIn,TabOut):-
  apply_nondet_rules(T,TabIn,TabOut).
```

Figure 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.

APPLY_ALL_RULES. Thus, if APPLY_ALL_RULES receives from APPLY_DET_RULES a modified tableau, it recursively calls APPLY_ALL_RULES on the new tableau. Otherwise, if it receives the tableau given in input to APPLY_DET_RULES, APPLY_ALL_RULES stops the recursion and the rule application, and returns the final tableau. This will be passed back to the first call to APPLY_ALL_RULES, that who started the recursion.

Finally, the `findall/3` predicate is used on the set of the built tableaux for finding all the clashes contained in them 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 5 is represented by `Tab-ExpansionPairQ`. This representation allows to easily manage the tableau, avoiding the need of implementing procedures such as GETEXPANSIONPAIRQ and REPLACEEXPANSIONPAIRQ used in Figure 5 (line 2 and line 5).

Therefore, the predicate `expand_queues/2` (corresponding with procedure EXPAND_QUEUES of Figure 5) 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, i.e., the assertion queues are empty. In Figure 12, the first definition of the predicate `expand_queues`, 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 to collect the justifications.

The implementation of `apply_all_rules/3` is very similar to that of `apply_all_rules/2`, as can be seen from Figure 13. The new argument taken as input is the assertion `AssToExpand` which is passed to the expansion rules which can manage it. So, the rules now follow a slightly different interface: deterministic tableau expansion rules are of the form *rule_name*(*TabIn*, *AssToExpand*, *TabOut*), while nondeterministic rules are implemented following the interface *rule_name*(*TabIn*, *AssToExpand*, *TabList*). As can be seen from the comments in Figure 13, they take as input the current tableau *TabIn* and the assertion 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 considering the assertion *AssToExpand*.

The pair of expansion queues *ExpansionPairQ* is implemented in `ExpansionPairQ` as a list containing two lists ini-

```
  expand_queues(Tab-[[],[]],Tab).

  expand_queues(Tab-ExpansionPairQ,TabExOut):-
  dequeue(ExpansionPairQ,AssToExpand,NewExpansionQueue),!,
  apply_all_rules(Tab-NewExpansionQueue,AssToExpand,TabTemp),
  expand_queues(TabTemp,TabExOut).
```

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

```
apply_all_rules(TabIn,AssToExpand,TabOut):-
  setting_trill(det_rules,DetRules),
  apply_det_rules(DetRules,TabIn,AssToExpand,TabTemp),
  (is_null(TabTemp) ->
     TabOut=TabTemp
     ;
     apply_all_rules(TabTemp,AssToExpand,TabOut)
  ).

apply_det_rules([],TabIn,AssToExpand,TabOut):-
  setting_trill(nondet_rules,Rules),
  apply_nondet_rules(Rules,TabIn,AssToExpand,TabOut).

apply_det_rules([H|_],TabIn,AssToExpand,TabOut):-
  call(H,TabIn,AssToExpand,TabOut),!.     % NOTE: here the deterministic expansion rules are called.
                                          % They take as input also AssToExpand.

apply_det_rules([_|T],TabIn,AssToExpand,TabOut):-
  apply_det_rules(T,TabIn,AssToExpand,TabOut).

apply_nondet_rules([],Tab,_,Tab).

apply_nondet_rules([H|_],TabIn,AssToExpand,TabOut):-
  call(H,TabIn,AssToExpand,L),!,          % NOTE: here the non-deterministic expansion rules are called.
                                          % They take as input also AssToExpand.
  member(TabOut,L),
  dif(TabIn,TabOut).

apply_nondet_rules([_|T],TabIn,AssToExpand,TabOut):-
  apply_nondet_rules(T,TabIn,AssToExpand,TabOut).
```

Figure 13: Application of the expansion rules by means of the predicates `apply_all_rules/3`, `apply_det_rules/4` and `apply_nondet_rules/4`.

tialized 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., disjunctions of concepts ($\sqcup$) and maximum cardinality restrictions ($\leq$). As seen in Figure 5 and then in Figure 12, the first operation of `expand_queues/2` is the extraction from `ExpansionPairQ` the next assertion to expand. The extraction is performed by using the predicate `dequeue/3` shown in Figure 14, which implements the lines 12-21 of the algorithm defined in Figure 5. 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.

```
dequeue([[],[EA|T]],EA,[[],T]).

dequeue([[EA|T],NonDetQ],EA,[T,NonDetQ]).
```

Figure 14: Code of the `extract_from_expansion_queues/3` predicate.

```
collect_individuals(Q,IndividualsSet):-
  extract_from_Q(Q,IndividualsSet0),
  scan_related_individuals(IndividualsSet0,[],IndividualsSet0,IndividualsSet).

scan_related_individuals([],_,IndividualsSet0,IndividualsSet):-
  sort(IndividualsSet0,IndividualsSet).

scan_related_individuals([H|IndividualsToCheck],IndividualsChecked,IndividualsSet0,IndividualsSet):-
  memberchk(H,IndividualsChecked),!,
  scan_related_individuals(IndividualsToCheck,IndividualsChecked,IndividualsSet0,IndividualsSet).


scan_related_individuals([H|IndividualsToCheck0],IndividualsChecked,IndividualsSet0,IndividualsSet):-
  gather_directly_related_individuals(H,NewIndividualsToCheck),
  append(IndividualsSet0,NewIndividualsToCheck,IndividualsSet1),
  append(IndividualsToCheck0,NewIndividualsToCheck,IndividualsToCheck),
  scan_related_individuals(IndividualsToCheck,[H|IndividualsChecked],IndividualsSet1,IndividualsSet).
```

Figure 15: Implementation of the `collect_individuals/2` predicate. It exploits an internal predicate, `scan_related_individuals/4`, for the gathering of the individuals.

B. Implementation of the KB Fragment Extraction

The code that implements the COLLECT_INDIVIDUALS procedure (Figure 4) is similar to its pseudo-code, as shown in Figure 15. All the sets used by the procedure are implemented using the list construct of Prolog and maintained without duplicates by the `sort/2` built-in predicate.

As shown in Figure 15, the `collect_individuals/2` predicate exploits an internal predicate that implements the while-loop of the COLLECT_INDIVIDUALS procedure shown in Figure 4, lines 5-14. This predicate, shown in Figure 15 and called `scan_related_individuals/4`, takes as input three lists `IndividualSet` and `IndividualsToCheck` both initialized unified with `IndividualSet0`, and `IndividualsCheched`, initialized as the empty list. Then, the predicate cycles on the content of `IndividualsToCheck` to collect all the individuals by calling the predicate `gather_directly_related_individuals/2`, shown in Figure 16. It finally unifies the set of all individuals related to those of the query in the list `IndividualSet`.

This predicate implements the second procedure of Figure 4, GATHER_DIRECTLY_RELATED_INDIVIDUALS, by adding all the individuals that are directly related to the individual `H` in the `RelatedIndividuals` list.

```
gather_directly_related_individuals(Ind,RelatedIndividuals):-
  find_successors(Ind,SuccInds),
  find_predecessors(Ind,PredInds),
  append(SuccInds,PredInds,RelatedIndividuals).

find_successors(Ind,List) :- findall(RelatedInd, (M:propertyAssertion(_,Ind,RelatedInd)), List).
find_predecessors(Ind,List) :- findall(RelatedInd, (M:propertyAssertion(_,RelatedInd,Ind)), List).
```

Figure 16: Implementation of the `gather_directly_related_individuals/2` predicate to select the individuals to include in the fragment of the KB.