Fundamenta Informaticae 57 (2003) 1–33
IOS Press

From Tableaux to Automata for Description Logics*

Franz Baader, Jan Hladik, and Carsten Lutz†
Theoretical Computer Science, TU Dresden,
D-01062 Dresden, Germany
baader,hladik,lutz@tcs.inf.tu-dresden.de

Frank Wolter
Department of Computer Science, University of Liverpool
Liverpool L69 7ZF, U.K.
frank@csc.liv.ac.uk

Abstract. This paper investigates the relationship between automata- and tableau-based inference procedures for description logics. To be more precise, we develop an abstract notion of what a tableau-based algorithm is, and then show, on this abstract level, how tableau-based algorithms can be converted into automata-based algorithms. In particular, this allows us to characterize a large class of tableau-based algorithms that imply an ExpTime upper-bound for reasoning in the description logics for which such an algorithm exists.

1. Introduction

Description logics (DLs) [1] are a family of knowledge representation languages which can be used to represent the terminological knowledge of an application domain in a structured and formally well-understood way. The name *description logics* is motivated by the fact that, on the one hand, the important notions of the domain are described by *concept descriptions*, i.e., expressions that are built from atomic concepts (unary predicates) and atomic roles (binary predicates) using the concept and role constructors provided by the particular DL. On the other hand, DLs differ from their predecessors, such as semantic

*This is an extended and improved version of a paper first published in *Proc. LPAR 2003*, Springer LNCS 2850. The work was partially supported by the German Research Foundation (DFG) under grant GR 1324/3-3.
†Address for correspondence: Theoretical Computer Science, TU Dresden, D-01062 Dresden, Germany

networks and frames [31, 25], in that they are equipped with a formal, *logic*-based semantics, which can, e.g., be given by a translation into first-order predicate logic.

Knowledge representation systems based on description logics (DL systems) [38, 26] provide their users with various inference capabilities (like subsumption and instance checking) that allow them to deduce implicit knowledge from the explicitly represented knowledge. In order to ensure a reasonable and predictable behavior of a DL system, these inference problems should at least be decidable, and preferably of low complexity. Consequently, the expressive power of the DL in question must be restricted in an appropriate way. If the imposed restrictions are too severe, however, then the important notions of the application domain can no longer be expressed. Investigating this trade-off between the expressivity of DLs and the complexity of their inference problems has been one of the most important issues in DL research (see [11] for an overview of complexity results).

The focus of this research has, however, changed in the last 15 years. In the beginning of the 1990ies, DL researchers investigated the border between tractable and intractable DLs [13, 14], and systems that employed so-called structural subsumption algorithms, which first normalize the concept descriptions, and then recursively compare the syntactic structure of the normalized descriptions, were still prevalent [30, 23, 24, 29]. It quickly turned out, however, that structural subsumption algorithms can handle only very inexpressive languages, and that one cannot expect a DL of reasonable expressive power to have tractable inference problems. For expressive DLs, tableau-based inference procedures turned out to be quite useful. After the first such tableau-based subsumption algorithm was developed by Schmidt-Schauß and Smolka [33] for the DL $\mathcal{ALC}$, this approach was extended to various other DLs and also to other inference problems such as the instance problem (see [5] for an overview).

Most of these early tableau-based algorithms for DLs were of optimal worst-case complexity: they treated DLs with a PSpace-complete subsumption problem, and the algorithms needed only polynomial space. Thus, by designing a tableau-based algorithm for such a DL one could solve two problems simultaneously: prove an optimal complexity upper-bound, and describe an algorithm that is easy to implement and optimize [2, 18], thus yielding a practical reasoning system for this DL. Modern tableau-based DL reasoners such as FaCT [17] and RACER [15] are based on very expressive DLs (like $\mathcal{SHIQ}$ [21]), which have an ExpTime-complete subsumption problem. Despite the high worst-case complexity of the underlying logics, the systems FaCT and RACER behave quite well in realistic applications. This is mainly due to the fact that their implementors have developed a great variety of sophisticated optimization techniques for tableau-based algorithms (see [18] for an overview of these techniques). Tableau-based algorithms are, however, notoriously bad at proving ExpTime upper-bounds.[1] In many cases, ExpTime upper-bounds are easily established using automata-based approaches (see, e.g., Section 5.3 in [9]). However, automata-based algorithms are not amenable to the sophisticated optimization techniques that have been developed for (tableau-based) state-of-the-art DL reasoners. Until now, it was thus common practice to devise two different algorithms for every ExpTime-complete DL, an automata-based one for establishing the exact worst-case complexity, and a tableau-based one for the implementation.

This paper investigates the (rather close) relationship between automata- and tableau-based algorithms. To be more precise, we develop an abstract notion of what a tableau-based algorithm is, and then show, on this abstract level, how tableau-based algorithms can be converted into automata-based algorithms. In particular, this allows us to characterize a large class of tableau-based algorithms that imply an

[1]The only such result we know of [12] treats the case of $\mathcal{ALC}$ with general concept inclusions (GCIs), and even in this simple case the algorithm is very complicated.

ExpTime upper-bound for reasoning in the DLs for which such an algorithm exists. We consider this to be a very useful result since, in many cases, it eliminates the need for developing two algorithms for the same DL: one can now design a tableau-based algorithm, use our general result to obtain an ExpTime upper-bound, and then base a practical implementation on the very same algorithm. We illustrate the usefulness of our framework by reproving the known ExpTime upper-bounds for the description logic $\mathcal{ALC}$ with a universal role [34], and for the extension $\mathcal{ALCQI}$ of $\mathcal{ALC}$ by qualified number restrictions and inverse roles [10].

In the next section, we introduce the abstract notion of a tableau system. In order to motivate and illustrate the technical definitions, we first consider the example of a tableau-based algorithm for $\mathcal{ALC}$ with a universal role. In Section 3, we define additional restrictions on tableau systems that ensure an exponential upper-bound on reasoning. This upper-bound is shown via a translation of tableau systems into looping tree automata. In Section 4, we show how tableau systems can directly be used to obtain a tableau-based decision procedure, which can be the basis for an optimized implementation. The main problem to be solved there is to ensure termination of the tableau-based algorithm. In Section 5, we apply the abstract framework to a more complex DL: we design a tableau system for the DL $\mathcal{ALCQI}$, thus giving an alternative proof of the known ExpTime upper-bound for reasoning in this DL (with numbers in number restrictions coded in unary). Finally, in Section 6, we discuss related work. In particular, we will explain how the present version of this article improves over a previous version [3].

2. Formalizing Tableau Algorithms

In this section, we develop an abstract formalization of tableau algorithms. To this end, we first discuss an extension of the standard tableau-based algorithm for the basic description logic $\mathcal{ALC}$ to $\mathcal{ALC}$ with a universal modality ($\mathcal{ALC}^U$), and then use this concrete example as a guide when devising the abstract framework.

2.1. A Tableau Algorithm for $\mathcal{ALC}^U$

We start with introducing the syntax and semantics of $\mathcal{ALC}^U$:

Definition 2.1. **($\mathcal{ALC}^U$ syntax)**
Let $\mathsf{N_C}$ and $\mathsf{N_R}$ be pairwise disjoint and countably infinite sets of *concept names* and *role names*. We assume that $\mathsf{N_R}$ contains a special role u, which is called the universal role. The set of $\mathcal{ALC}^U$-concepts $\mathsf{CON}_{\mathcal{ALC}^U}$ is the smallest set such that

- every concept name is an $\mathcal{ALC}^U$-concept, and
- if C and D are $\mathcal{ALC}^U$-concepts and r is a role name, then the following expressions are also $\mathcal{ALC}^U$-concepts: $\neg C$, $C \sqcap D$, $C \sqcup D$, $\exists r.C$, $\forall r.C$.

A *general concept inclusion (GCI)* is an expression $C \sqsubseteq D$, where both C and D are $\mathcal{ALC}^U$-concepts. A finite set of GCIs is called $\mathcal{ALC}^U$*-TBox*.

As usual, we will use $\top$ as abbreviation for an arbitrary propositional tautology, $\bot$ for $\neg\top$, and $C \rightarrow D$ for $\neg C \sqcup D$.

Note that there exist several different TBox formalisms that vary considerably w.r.t. expressive power (see [4]). The kind of TBoxes adopted here are among the most general ones available. They are supported by modern DL reasoners such as FaCT and RACER. However, in the presence of the universal role, reasoning w.r.t. such complex TBoxes can be reduced to reasoning without a TBox (see below).

Like all DLs, $\mathcal{ALC}^U$ is equipped with a Tarski-style set-theoretic semantics.

Definition 2.2. **($\mathcal{ALC}^U$ semantics)**
An *interpretation* $\mathcal{I}$ is a pair $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, where $\Delta^{\mathcal{I}}$ is a non-empty set, called the *domain*, and $\cdot^{\mathcal{I}}$ is the *interpretation function*. The interpretation function maps each concept name A to a subset $A^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ and each role name r to a subset $r^{\mathcal{I}}$ of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. It interprets the universal role u as the universal relation $u^{\mathcal{I}} := \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and is extended to arbitrary $\mathcal{ALC}^U$-concepts as follows:

$$\begin{aligned}
(\neg C)^{\mathcal{I}} &:= \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C \sqcap D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
(C \sqcup D)^{\mathcal{I}} &:= C^{\mathcal{I}} \cup D^{\mathcal{I}} \\
(\exists r.C)^{\mathcal{I}} &:= \{d \in \Delta^{\mathcal{I}} \mid \text{There is } e \in \Delta^{\mathcal{I}} \text{ with } (d,e) \in r^{\mathcal{I}} \text{ and } e \in C^{\mathcal{I}}\} \\
(\forall r.C)^{\mathcal{I}} &:= \{d \in \Delta^{\mathcal{I}} \mid \text{For all } e \in \Delta^{\mathcal{I}}, \text{ if } (d,e) \in r^{\mathcal{I}}, \text{ then } e \in C^{\mathcal{I}}\}
\end{aligned}$$

The interpretation $\mathcal{I}$ is a *model* of the $\mathcal{ALC}^U$-concept C iff $C^{\mathcal{I}} \neq \emptyset$, and it is a model of the TBox $\mathcal{T}$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds for all $C \sqsubseteq D \in \mathcal{T}$.

The main inference problems related to a TBox are satisfiability and subsumption of concepts.

Definition 2.3. **($\mathcal{ALC}^U$ inference problems)**
The $\mathcal{ALC}^U$-concept C is *satisfiable w.r.t. the TBox* $\mathcal{T}$ iff C and $\mathcal{T}$ have a common model, and C is *subsumed by* the $\mathcal{ALC}^U$-concept D *w.r.t. the TBox* $\mathcal{T}$ (written $C \sqsubseteq_{\mathcal{T}} D$) iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ holds for all models $\mathcal{I}$ of $\mathcal{T}$. If the TBox is empty, then we say that C is satisfiable rather than that C is satisfiable w.r.t. $\emptyset$, and that C is subsumed by D ($C \sqsubseteq D$) rather than that C is subsumed by D w.r.t. $\emptyset$ ($C \sqsubseteq_{\emptyset} D$).

Since $C \sqsubseteq_{\mathcal{T}} D$ iff $C \sqcap \neg D$ is unsatisfiable w.r.t. $\mathcal{T}$, it is sufficient to design a satisfiability algorithm. In addition, in the presence of the universal modality, satisfiability w.r.t. a TBox $\mathcal{T}$ can be reduced to satisfiability w.r.t. the empty TBox (i.e., satisfiability): it is easy to see that C_0 is satisfiable w.r.t. $\mathcal{T} = \{C_1 \sqsubseteq D_1, \ldots, C_n \sqsubseteq D_n\}$ iff $C_0 \sqcap \forall u.(C_1 \rightarrow D_1) \sqcap \ldots \sqcap \forall u.(C_n \rightarrow D_n)$ is satisfiable. Consequently, we will restrict our attention to the satisfiability problem (i.e., w.r.t. the empty TBox).

We now discuss an extension of the standard tableau-based satisfiability algorithm for $\mathcal{ALC}$ that handles the universal modality. In the context of DLs, the standard algorithm for $\mathcal{ALC}$ has first been described in [33]; more modern accounts can, e.g., be found in [5]. It can rightfully be viewed as the ancestor from which all state-of-the-art tableau-based algorithms for description logics are descended. Such algorithms are nowadays the standard approach for reasoning in DLs, and they underlie modern and efficient reasoning systems such as FaCT and RACER, which are based on DLs that are much more expressive than $\mathcal{ALC}$.

Tableau algorithms are characterized by an underlying data structure, a set of completion rules, and a number of so-called clash-triggers. To decide the satisfiability of an input concept C, the algorithm starts with an initial instance of the data structure constructed from C, and repeatedly applies completion rules to it. This rule application can be viewed as an attempt to construct a model for the input, or as making implicit knowledge explicit. Rule application continues until either one of the clash-triggers applies, which means that the attempt to construct a model has failed, or all implicit knowledge has been made explicit without encountering a clash-trigger. In the latter case, the algorithm has succeeded to construct (a representation of) a model. To be more precise, the tableau algorithms considered in this paper may be non-deterministic, i.e., there may exist completion rules that yield more than one possible outcome. In this case, the algorithm returns "satisfiable" iff there exists at least *one* way to apply the non-deterministic rules such that a model of the input is obtained. Note that only the choice of the outcome of non-deterministic rules is true "don't know" non-determinism (and thus requires backtracking), whereas the order of rule applications is basically "don't care" non-determinism.

Before we can define the data structure underlying the $\mathcal{ALC}^U$ tableau algorithm, so-called completion trees, we must introduce some notation. Given an $\mathcal{ALC}^U$-concept C, its *negation normal form (NNF)* is an equivalent[2] concept such that negation occurs only in front of concept names. Such a concept can easily be computed by pushing negation as far as possible into concepts, using de Morgan's rules and the usual duality rules for quantifiers. In the following, we w.l.o.g. assume that the input concepts to the $\mathcal{ALC}^U$ tableau algorithm are in negation normal form. If C is an $\mathcal{ALC}^U$-concept then we use $\mathsf{sub}(C)$ to denote the set of all subconcepts of C.

Definition 2.4. (Completion trees)
Let C be an $\mathcal{ALC}^U$-concept in NNF. A *completion tree* for C is a labeled tree[3] $t = (V, E, \mathcal{N}, \mathcal{E})$ of finite out-degree such that (V, E) is a tree, each node $a \in V$ is labeled with a subset $\mathcal{N}(a)$ of $\mathsf{sub}(C)$ and each edge $(a, b) \in E$ is labeled with a role name $\mathcal{E}(a, b)$ occurring in C.

The completion rules are given in Figure 1, where $\mathsf{R}\sqcup$ is the only non-deterministic rule. To decide satisfiability of a concept C in NNF, the $\mathcal{ALC}^U$ tableau algorithm starts with the initial completion tree

$$T_C := (\{x\}, \emptyset, \{x \mapsto \{C\}, \emptyset)$$

and repeatedly applies completion rules. Rule application stops in one of the following two cases:

1. the obtained completion tree $t = (V, E, \mathcal{N}, \mathcal{E})$ *contains a clash*, i.e. there is a node $a \in V$ and a concept name A such that $\{A, \neg A\} \subseteq \mathcal{N}(a)$;

2. t is *saturated*, i.e. no more completion rules are applicable to t.

If we consider only $\mathcal{ALC}$-concepts (i.e., $\mathcal{ALC}^U$-concepts not containing the universal role), then we can drop the $\mathsf{R}U$ rule. In this case, the described algorithm terminates for any input and any sequence of rule applications. Things are not so simple if we admit the universal role: because of the $\mathsf{R}U$ rule, the algorithm need not terminate, both on satisfiable and on unsatisfiable inputs. For example, rule application to the concept $\forall u.\exists r.\top$ continues indefinitely. However, the algorithm then computes an infinite "increasing" sequence of completion trees: in each step, the tree and its node labels may only grow but never shrink. In case of non-termination, there thus exists a unique completion tree computed by

[2]Two concepts are equivalent iff they subsume each other w.r.t. the empty TBox.

[3]Here and in the following, a tree is an acyclic directed graph (V, E) with a unique root where every node other than the root is reachable from the root and has exactly one predecessor. The edge relation E is a sub*set* of $V \times V$, and thus the successors of a given node are not ordered.

R$\sqcap$	if $C_1 \sqcap C_2 \in \mathcal{N}(a)$ and $\{C_1, C_2\} \not\subseteq \mathcal{N}(a)$ then $\mathcal{N}(a) := \mathcal{N}(a) \cup \{C_1, C_2\}$
R$\sqcup$	if $C_1 \sqcup C_2 \in \mathcal{N}(a)$ and $\{C_1, C_2\} \cap \mathcal{N}(a) = \emptyset$ then $\mathcal{N}(a) := \mathcal{N}(a) \cup \{C\}$ for some $C \in \{C_1, C_2\}$
R$\exists$	if $\exists r.C \in \mathcal{N}(a)$ and there is no r-successor b of a with $C \in \mathcal{N}(b)$, then generate a new successor b of a, and set $\mathcal{E}(a, b) := r$ and $\mathcal{N}(b) := \{C\}$
R$\forall$	if $\forall r.C \in \mathcal{N}(a)$ and b is an r-successor of a with $C \notin \mathcal{N}(b)$ then set $\mathcal{N}(b) := \mathcal{N}(b) \cup \{C\}$
RU	if $\forall u.C \in \mathcal{N}(a)$ and $C \notin \mathcal{N}(b)$ then set $\mathcal{N}(b) := \mathcal{N}(b) \cup \{C\}$

Figure 1. Completion rules for $\mathcal{ALC}^U$.

this run of the algorithm "in the limit". Thus, both terminating and non-terminating runs of the algorithm "compute" a unique completion tree. This (possibly infinite) completion tree is called *saturated* iff no more completion rules are applicable to it.

The tableau algorithm for $\mathcal{ALC}^U$ is sound and complete in the following sense:

- *Soundness*. If the algorithm computes a saturated and clash-free completion tree for the input C, then C is satisfiable.
- *Completeness*. If the input C is satisfiable, then there is a run of the algorithm that computes a saturated and clash-free completion tree for this input.

Given these notions of soundness and completeness, it should be clear that we want our algorithm to compute saturated completion trees. Obviously, any terminating run of the algorithm yields a saturated completion tree. For this reason, the order of rule applications is in this case "don't care" non-deterministic. For a non-terminating run, this is only true if we require completion rules to be applied in a *fair*[4] manner. Ensuring fairness is a simple task: we can, e.g., always apply completion rules to those nodes in the tree that are as close to the root as possible. This yields a fair strategy since the out-degree of completion trees constructed for an input C is bounded by the cardinality of the set $\mathsf{sub}(C)$.

Although the procedure as described until now does not necessarily terminate and thus is no decision procedure for satisfiability, quite surprisingly we will see that it already provides us with enough information to deduce an ExpTime upper-bound for $\mathcal{ALC}^U$-concept satisfiability (and thus, in particular, with a decidability result). This will be shown by a translation into a tree automaton, which basically accepts saturated and clash-free completion trees for the input. We view this as a rather convenient feature of our framework: to obtain an ExpTime decision procedure, it is sufficient to design a sound and complete

[4]Intuitively, fairness means that rules are applied such that every applicable rule will eventually be applied unless it is made inapplicable by the application of other rules.

tableau algorithm and not even bother to prove termination, a usually hard task (see Section 3 for details). Moreover, we will show in Section 4 that a given non-terminating sound and complete tableau procedure can always be turned into a terminating sound and complete procedure. This yields a tableau-based *decision* procedure, which is, however, not necessarily of ExpTime complexity.

2.2. The General Framework

We now develop a general notion of tableau algorithms. It is in the nature of this endeavor that our formalism will be a rather abstract one. We start with defining the core notion: tableau systems. Intuitively, the purpose of a tableau system is to capture all the details of a tableau algorithm such as the one for $\mathcal{ALC}^U$ discussed in the previous section. The set $\mathfrak{I}$ of inputs used in the following definition can be thought of as consisting of all possible concepts C (or pairs $(C, \mathcal{T})$ of concepts C and TBoxes $\mathcal{T}$) of the DL under consideration.

Definition 2.5. (Tableau system)
Let $\mathfrak{I}$ be a set of *inputs*. A *tableau system for* $\mathfrak{I}$ is a tuple

$$S = (\mathsf{NLE}, \mathsf{GME}, \mathsf{EL}, k, \cdot^S, \mathcal{R}, \mathcal{C}),$$

where NLE, GME, and EL are sets of *node label elements*, *global memory elements*, and *edge labels*, respectively, k is a natural number (the *pattern depth*), and $\cdot^S$ is a function mapping each input $\Gamma \in \mathfrak{I}$ to a tuple

$$\Gamma^S = (\mathsf{nle}, \mathsf{gme}, \mathsf{el}, \mathsf{ini})$$

such that

- $\mathsf{nle} \subseteq \mathsf{NLE}$, $\mathsf{gme} \subseteq \mathsf{GME}$, and $\mathsf{el} \subseteq \mathsf{EL}$ are finite;
- ini is a subset of $\wp(\mathsf{nle}) \times \wp(\mathsf{gme})$, where $\wp(\cdot)$ denotes powerset.

The definitions of $\mathcal{R}$ and $\mathcal{C}$ depend on the notion of an S*-pattern*. Such a pattern is a pair (t, μ) consisting of a a finite labeled tree

$$t = (V, E, n, \ell),$$

of depth at most k with $n : V \to \wp(\mathsf{NLE})$ and $\ell : E \to \mathsf{EL}$ node and edge labeling functions, and a subset μ of GME.

- $\mathcal{R}$, the collection of *completion rules*, is a function mapping each S-pattern to a finite set of non-empty finite sets of S-patterns;
- $\mathcal{C}$, the collection of *clash-triggers*, is a set of S-patterns.

To illustrate tableau systems, we now define a tableau system $S_{\mathcal{ALC}^U}$ that describes the $\mathcal{ALC}^U$ tableau algorithm discussed in the previous section. As the set of inputs $\mathfrak{I}$ for $S_{\mathcal{ALC}^U}$, we simply use the set of all $\mathcal{ALC}^U$-concepts in NNF. Now for the tableau system itself. Intuitively, NLE is the set of elements that may appear in node labels of completion trees, *independently* of the input. In the case of $\mathcal{ALC}^U$, NLE is thus simply $\mathsf{CON}_{\mathcal{ALC}^U}$. Similarly, EL is the set of edge labels, also independently of the input. In the case of $\mathcal{ALC}^U$, EL is thus the set of role names $\mathsf{N_R}$. The rôle of the global memory component can be illustrated by the RU rule. In contrast to the other rules, which are local in the sense that they are concerned with a single node of the completion tree or a single node and its successor nodes, the RU rule is global: it considers two *arbitrary* nodes a and b in the completion tree. The global memory component contains information relevant for such global rules. For the RU rule, it is important to know which concepts C must be propagated to all nodes since $\forall u.C$ is contained in some node. Thus, the global memory component also contains concepts, which means that, in the case of $\mathcal{ALC}^U$, GME is also $\mathsf{CON}_{\mathcal{ALC}^U}$. The number k restricts the size of the trees in patterns. We will consider it in more detail when describing the rules and clash-triggers.

The function $\cdot^S$ describes the impact of the input on the form of the constructed completion trees. More precisely, nle fixes the node label elements that may be used in a completion tree for a particular input, and el fixes the edge labels. Similarly, gme fixes the possible elements of the global memory component for a particular input. Finally, ini describes the possible initial node labels of the root of the completion tree as well as the initial value of the global memory component. Note that the initial root label and the initial value of the global memory component are not necessarily unique, but rather there can be many choices—a possible source of (don't know) non-determinism that does not show up in the $\mathcal{ALC}^U$ algorithm. To illustrate the function $\cdot^S$, let us define it for the tableau system $S_{\mathcal{ALC}^U}$. For simplicity, we write $\mathsf{nle}_{S_{\mathcal{ALC}^U}}(C)$ to refer to the first element of the tuple $C^{S_{\mathcal{ALC}^U}}$, $\mathsf{gme}_{S_{\mathcal{ALC}^U}}(C)$ to refer to the second element of the tuple $C^{S_{\mathcal{ALC}^U}}$, and so forth. For each input $C \in \mathsf{CON}_{\mathcal{ALC}^U}$, we have

$$
\begin{aligned}
\mathsf{nle}_{S_{\mathcal{ALC}^U}}(C) &= \mathsf{sub}(C);\\
\mathsf{gme}_{S_{\mathcal{ALC}^U}}(C) &= \mathsf{sub}(C);\\
\mathsf{el}_{S_{\mathcal{ALC}^U}}(C) &= \{r \in \mathsf{N_R} \mid r \text{ appears in C}\};\\
\mathsf{ini}_{S_{\mathcal{ALC}^U}}(C) &= \{(\{C\}, \emptyset)\}.
\end{aligned}
$$

It remains to formalize the completion rules and clash-triggers. First observe that, in the $\mathcal{ALC}^U$ tableau, every clash-trigger as well as every rule premise and consequence (except for the RU rule) concerns only a single node either alone or together with its successors in the completion tree. For this reason, we can restrict the depth of the trees in patterns to $k = 1$. The global RU rule is handled through the global memory component (see the description of the rules below).

The collection of completion rules $\mathcal{R}$ maps patterns to finite sets of finite sets of patterns. Intuitively, if P is a pattern and $\{P_1, \ldots, P_m\} \in \mathcal{R}(P)$, then this means that a rule of the collection can be applied to all completion trees "matching" the pattern P. For this, the tree part of the pattern must match a subtree of the completion tree, and the global memory component of the pattern must coincide with the global memory component of the completion tree. If a rule matches a completion tree in this sense, then it non-deterministically replaces the matched subtree of the completion tree with a subtree matching the tree part of one of the patterns $P_1, \ldots, P_m$ (we will give a formal definition of this later on). In addition, the global memory component of the completion tree is replaced by the global memory component of the right-hand side pattern. If $\{P_1, \ldots, P_m\} \in \mathcal{R}(P)$, then we will usually write

$$P \rightarrow_{\mathcal{R}} \{P_1, \ldots, P_m\}$$

to indicate the rule induced by this element of $\mathcal{R}(P)$. Similar to the application of such a rule, a completion tree contains a clash if this completion tree matches a pattern in $\mathcal{C}$.

To illustrate this, let us again consider the case of $\mathcal{ALC}^U$. For $\mathcal{ALC}^U$, the set of clash-triggers $\mathcal{C}$ consists of all patterns whose tree has a root label containing both A and $\neg A$ for some concept name A. The effect of this is that a completion tree contains a clash iff one of its node labels contains A and $\neg A$ for some concept name A.

With two exceptions, the collection of completion rules is defined by a straightforward translation of the rules in Figure 1. For each pattern $P = (t, \mu)$ with $t = (V, E, n, \ell)$ a tree of depth ≤ 1 with root v_0, $\mathcal{R}(P)$ is the smallest set of finite sets of patterns such that the following holds:

R$\sqcap$ if the root label $n(v_0)$ contains the concept $C \sqcap D$ and $\{C, D\} \not\subseteq n(v_0)$, then $\mathcal{R}(P)$ contains the singleton set $\{((V, E, n', \ell), \mu)\}$, where $n'(v) = n(v)$ for all $v \in V \setminus \{v_0\}$ and $n'(v_0) = n(v_0) \cup \{C, D\}$;

R$\sqcup$ if the root label $n(v_0)$ contains the concept $C \sqcup D$ and $\{C, D\} \cap n(v_0) = \emptyset$, then $\mathcal{R}(P)$ contains the set $\{((V, E, n', \ell), \mu), ((V, E, n'', \ell), \mu)\}$, where $n'(v) = n''(v) = n(v)$ for all $v \in V \setminus \{v_0\}$ and $n'(v_0) = n(v_0) \cup \{C\}$ and $n''(v_0) = n(v_0) \cup \{D\}$;

R$\exists$ if the root label $n(v_0)$ contains the concept $\exists r.C$, $u_1, \ldots, u_m$ are all the sons of v_0 with $\ell(v_0, u_i) = r$, and $C \notin n(u_i)$ for all $i, 1 \leq i \leq m$, then $\mathcal{R}(P)$ contains the set $\{P_0, P_1, \ldots, P_m\}$, where

- $P_0 = ((V_0, E_0, n_0, \ell_0), \mu)$, where u_0 is a node not contained in V, $V_0 = V \cup \{u_0\}$, $E' = E \cup \{(v_0, u_0)\}$, $n_0 = n \cup \{u_0 \mapsto \{C\}\}$, $\ell' = \ell \cup \{(v_0, u_0) \mapsto r\}$,
- for $i = 1, \ldots, m$, $P_i = ((V, E, n_i, \ell), \mu)$, where $n_i(v) = n(v)$ for all $v \in V \setminus \{u_i\}$ and $n_i(u_i) = n(u_i) \cup \{C\}$;

R$\forall$ if $n(v_0)$ contains the concept $\forall r.C$, $\ell(v_0, v_1) = r$ for some $v_1 \in V$, and $C \notin n(v_1)$, then $\mathcal{R}(P)$ contains $\{((V, E, n', \ell), \mu)\}$, where $n'(v) = n(v)$ for all $v \in V \setminus \{v_1\}$ and $n'(v_1) = n(v_1) \cup \{C\}$;

R$U1$ if $\forall u.C \in n(v_0)$ and $C \notin \mu$, then $\mathcal{R}(P)$ contains the set $\{(t, \mu \cup \{C\})\}$.

R$U2$ if μ contains the concept C, and $C \notin n(v_0)$, then $\mathcal{R}(P)$ contains $\{((V, E, n', \ell), \mu)\}$, where $n'(v) = n(v)$ for all $v \in V \setminus \{v_0\}$ and $n'(v_0) = n(v_0) \cup \{C\}$.

The first exception is the RU rule, which is now split into two rules. The first rule stores the information about which concepts must hold at all nodes in the global memory component, and the second then propagates the concepts stored in this component to all other nodes.

The second exception is the treatment of existential restrictions. The rule in Figure 1 is deterministic: it always generates a *new* r-successor of the given node. In contrast, the rule handling existential restrictions introduced above (don't know) non-deterministically chooses between generating a new successor or re-using one of the old ones. Basically, this is the price we have to pay for having a very general framework. The reason why one can always create a new individual when treating existential restrictions in $\mathcal{ALC}^U$ is that $\mathcal{ALC}^U$ is invariant under bisimulation [7], and thus one can duplicate successors in models without changing validity. We could have tailored our framework such that the deterministic rule for $\mathcal{ALC}^U$ can be used, but then we basically would have restricted its applicability to DLs invariant under bisimulation, a property that is violated by other DLs such as those providing for number restrictions (see Section 6 in [3] for a more detailed discussion of this issue).

Let us now continue with the general definitions. Tableau systems are a rather general notion. In fact, as described until now they are too general to be useful for our purposes. For example, tableau algorithms described by such tableau systems need not be monotonic: completion rules could repeatedly (even indefinitely) add and remove the same piece of information. To prevent such pathologic behavior, we now formulate a number of conditions that "well-behaved" tableau systems are supposed to satisfy. For the following definitions, fix a set of inputs $\mathfrak{I}$ and a tableau system $S = (\mathsf{NLE}, \mathsf{GME}, \mathsf{EL}, k, \cdot^S, \mathcal{R}, \mathcal{C})$ for $\mathfrak{I}$. Before we can define admissibility of tableau systems, we must introduce an "inclusion relation" between patterns.

Definition 2.6. Let $P = (t, \mu)$ and $P' = (t', \mu')$ with $t = (V, E, n, \ell)$ and $t' = (V', E', n', \ell')$ be S-patterns. We write $P \precsim P'$ iff the following conditions are satisfied: $\mu \subseteq \mu'$ and there is an injection $\pi : V \to V'$ that maps the root of t to the root of t' and satisfies the following conditions:

- for all $x \in V$, we have $n(x) \subseteq n'(\pi(x))$;
- for all $x, y \in V$, if $(x, y) \in E$, then $(\pi(x), \pi(y)) \in E'$ and $\ell(x, y) = \ell'(\pi(x), \pi(y))$.

If π is the identity on V (and thus $V \subseteq V'$), then we write $P \preceq P'$ (and $P \prec P'$ if, additionally, $P \neq P'$). If $\mu = \mu'$, π is a bijection, and $n(x) = n'(\pi(x))$ for all $x \in V$, then we write $P \sim P'$. To make the injection (bijection) π explicit, we sometimes write $P \precsim_\pi P'$ ($P \sim_\pi P'$).

Let $\Gamma \in \mathfrak{I}$ be an input. We say that $P = (t, \mu)$ is a *pattern for* Γ iff μ is a subset of $\mathsf{gme}_S(\Gamma)$, the labels of all nodes in t are subsets of $\mathsf{nle}_S(\Gamma)$, and the labels of all edges in t belong to $\mathsf{el}_S(\Gamma)$. The pattern P is *saturated* iff $\mathcal{R}(P) = \emptyset$.

Definition 2.7. (Admissible)
The tableau system S is called *admissible* iff it satisfies, for all S-patterns P and P', the following conditions:

1. If $P \to_\mathcal{R} \{P_1, \ldots, P_m\}$, then $P \prec P_i$ for all $i, 1 \leq i \leq m$.
2. If $P \to_\mathcal{R} \{P_1, \ldots, P_m\}$, P' is saturated, and $P \precsim P'$, then there exists an $i, 1 \leq i \leq m$, such that $P_i \precsim P'$.
3. For all inputs $\Gamma \in \mathfrak{I}$, if P is a pattern for Γ and $P \to_\mathcal{R} \{P_1, \ldots, P_m\}$, then the patterns P_i are patterns for Γ.
4. If $P \in \mathcal{C}$ and $P \precsim P'$, then $P' \in \mathcal{C}$.

It is in order to discuss the intuition underlying the above conditions. Condition 1 basically says that rule application always adds nodes, elements of node labels, or elements of the global memory component. Condition 2 can be understood as follows. Assume that a (non-deterministic) rule is applicable to P and that P' is a "superpattern" of P that is saturated (i.e., all applicable rules have already been applied). Then the non-deterministic rule can be applied in such a way that the obtained new pattern is still a subpattern of P'. Intuitively, this condition can be used to reach P' from P by repeated rule application. Condition 3 says that, by applying completion rules for some input Γ, we stay within the limits given by the values of the $\cdot^S$ function. Condition 4 states that applicability of clash-triggers is monotonic, i.e., if a pattern triggers a clash, all its "superpatterns" also trigger a clash.

It is easy to see that these conditions are satisfied by the tableau system $S_{\mathcal{ALC}^U}$ for $\mathcal{ALC}^U$. For Condition 1, this is obvious since the rules only add nodes, elements of node labels, or elements of the global memory component, but never remove them. Condition 3 holds since rules only add subconcepts of existing concepts to the node label or the global memory component. Condition 4 is also clear: if $P = (t, \mu)$ and the label of the root of t contains A and $\neg A$, then the label of the root of the tree of every superpattern of P also contains A and $\neg A$.

The most interesting condition is Condition 2. We illustrate it by considering the treatment of disjunction and of existential restrictions in $S_{\mathcal{ALC}^U}$. First, assume that $P \rightarrow_{\mathcal{R}} \{P_1, P_2\}$ where the root label of the tree t of P contains $C \sqcup D$ and the root labels of the trees of P_1 and P_2 are obtained from the root label of t by respectively adding C and D. If $P \precsim P'$, then the root label of the tree of P' also contains $C \sqcup D$. If, in addition, P' is saturated, then this root label already contains C or D. In the first case, $P' \precsim P_1$ and in the second $P' \precsim P_2$.

Second, consider the rules handling existential restrictions. Thus, let $P \precsim P'$, and assume that the root label of the tree t of P contains the existential restriction $\exists r.C$ and that the root of t has m r-successors $u_1, \ldots, u_m$. Then the existential restriction $\exists r.C$ induces the rule $P \rightarrow_{\mathcal{R}} \{P_0, \ldots, P_m\}$ where the patterns $P_0, \ldots, P_m$ are as defined above. If, in addition, P' is saturated, then the root of its tree has an r-successor whose label contains C. If this is a "new" r-successor (i.e., one not in the range of the injection π that ensures $P \precsim P'$), then $P_0 \precsim P'$.[5] Otherwise, there is an r-successor u_i of the root of t such that the label of $\pi(u_i)$ in the tree of P' contains C. In this case, $P_i \precsim P'$.

We now introduce S-trees, the abstract counterpart of completion trees, and define what it means for a pattern to match into an S-tree.

Definition 2.8. (S-tree, matching)
An *S-tree* is a pair $T = (t, \mu)$ where $\mu \subseteq \mathsf{GME}$ and $t = (V, E, n, \ell)$ is a labeled tree with finite out-degree, a countable set of nodes V, and the node and edge labeling functions $n : V \rightarrow \wp(\mathsf{NLE})$ and $\ell : E \rightarrow \mathsf{EL}$. Any node $x \in V$ defines a pattern T, x, the *k-neighborhood* of x in T, as follows: $T, x := ((V', E', n', \ell'), \mu)$ where

- $V' = \{x\} \cup \{y \in V \mid \text{there is a path from } x \text{ to } y \text{ of length} \leq k \text{ in } t\}$;
- E', n', ℓ' are the restrictions of E, n, ℓ to V'.

The tree (V', E', n', ℓ') of T, x is denoted by t, x. If P is an arbitrary S-pattern and $x \in V$, then we say that P *matches* x in T iff $P \sim T, x$ (see Definition 2.6).

For the tableau system for $\mathcal{ALC}^U$ introduced above, $S_{\mathcal{ALC}^U}$-trees are basically the completion trees defined in Section 2. The only difference is that $S_{\mathcal{ALC}^U}$-trees have an additional global memory component μ.

Later on, we need sub-tree relations between S-trees in analogy to the inclusion relations "$\precsim$" and "$\preceq$" between patterns introduced in Definition 2.6. These relations are defined on trees exactly as for patterns, and we also use the same relation symbols for them.

We are now ready to describe rule application on an abstract level. Intuitively, the rule $P \rightarrow_{\mathcal{R}} \{P_1, \ldots, P_m\}$ can be applied to the node x in the S-tree T if $P \sim T, x$, and its application yields the

[5]This shows that we cannot replace $\precsim$ by $\preceq$ in the statement of Condition 2. In fact, we cannot be sure that the new successor introduced in P_0 has the same name as the new successor in P'.

new tree T', which is obtained from T by adding new nodes to T, x and/or extending labels of nodes from T, x and/or extending the global memory component, as indicated by some P_i. This intuition is formalized in the following definition.

Definition 2.9. (Rule application)
Let S be an admissible tableau system, $T = (t, \mu)$ be an S-tree, and $P \rightarrow_{\mathcal{R}} \{P_1, \ldots, P_m\}$ be a rule of S. The S-tree $T' = (t', \mu')$ is obtained from T by application of this rule to a node x of t iff the following conditions hold:

1. $P \sim_\pi T, x$ for some bijection π.

2. There is an $i, 1 \leq i \leq m$ such that T' is obtained from T by replacing T, x by P_i.
   To be more precise, let $t = (V, E, n, \ell)$, $P = (t_0, \mu_0)$ where $t_0 = (V_0, E_0, n_0, \ell_0)$, and $P_i = (t_i, \mu_i)$ where $t_i = (V_i, E_i, n_i, \ell_i)$, and assume (without loss of generality) that $V \cap V_i = \emptyset$. Let π' be the extension of π to V_i that is the identity on $V_i \setminus V_0$. Then $\mu' = \mu_i$ and $t' = (V', E', n', \ell')$, where

   (a) $V' = V \cup (V_i \setminus V_0)$;
   (b) $E' = E \cup \{(\pi'(y), \pi'(z)) \mid (y, z) \in E_i\}$;
   (c) $n'(y') = n(y')$ if $y' \notin \mathsf{ran}(\pi')$ and $n'(y') = n_i(y)$ if $y' = \pi'(y)$ for some $y \in V_i$;
   (d) $\ell'(y, z) = \ell(y, z)$ for all $(y, z) \in E$, and
   $\ell'(y', z') = \ell_i(y, z)$ if $y' = \pi'(y)$, $z' = \pi'(z)$, and $(y, z) \in E_i$.

For a fixed rule $P \rightarrow_{\mathcal{R}} \{P_1, \ldots, P_m\}$, a fixed choice of P_i, and a fixed node x in T, the results of the rule application is unique. It is easy to check that, in the case of $S_{\mathcal{ALC}}$, rule application as defined above captures precisely the intuitive understanding of rule application employed in Section 2.

To finish our abstract definition of tableau algorithms, we need some way to describe the set of S-trees that can be obtained by starting with an initial S-tree for an input Γ, and then repeatedly applying completion rules. This leads to the notion of S-trees for Γ.

Definition 2.10. (S-tree for Γ)
Let S be an admissible tableau system, and let Γ be an input for S. The set of *S-trees for* Γ is the smallest set of S-trees such that

1. All *initial S-trees for* Γ belong to this set, where an initial S-tree for Γ is of the form

   $$((\{v_0\}, \emptyset, \{v_0 \mapsto \Lambda\}, \emptyset), \mu)$$

   where v_0 is a node and $(\Lambda, \mu) \in \mathsf{ini}_S(\Gamma)$.

2. If T is an S-tree for Γ and T' can be obtained from T by the application of a completion rule, then T' is an S-tree for Γ.

3. If $T_0, T_1, \ldots$ is an infinite sequence of S-trees for Γ with $T_i = ((V_i, E_i, n_i, \ell_i), \mu_i)$ such that

   (a) T_0 is an initial S-tree for Γ and
   (b) for all $i \geq 0$, T_{i+1} can be obtained from T_i by the application of a completion rule,

then the S-tree $T^\omega = ((V, E, n, \ell), \mu)$ is also an S-tree for Γ, where

- $V = \bigcup_{i \geq 0} V_i$,
- $E = \bigcup_{i \geq 0} E_i$,
- $n = \bigcup_{i \geq 0} n_i$,
- $\ell = \bigcup_{i \geq 0} \ell_i$, and
- $\mu = \bigcup_{i \geq 0} \mu_i$.

Rule application may terminate after finitely many steps or continue forever. The last case of Definition 2.10 deals with such infinite sequences of rule applications. The S-tree T^ω can be viewed as the limit of the sequence of S-trees $T_0, T_1, \ldots$ This limit exists since admissibility of S implies that rule application is monotonic w.r.t. the sub-tree relationship "$\preceq$", i.e., it extends S-trees by new nodes or by additional elements in node labels, but it never removes nodes or elements of node labels.

Let us now define when an S-tree is saturated and clash-free.

Definition 2.11. (Saturated, clash-free)
Let S be an admissible tableau system. We say that the S-tree T is

- *saturated* if, for every node x in T and every pattern P, $P \sim T, x$ implies $\mathcal{R}(P) = \emptyset$;

- *clash-free* if, for every node x in T and every $P \in \mathcal{C}$, we have $P \not\sim T, x$.

Saturatedness says that no completion rule is applicable to the S-tree, and an S-tree is clash-free if no clash-trigger can be applied to any of its nodes.

Finally, we define soundness and completeness of tableau systems w.r.t. a certain property of its set of inputs. If the inputs are concepts (pairs consisting of a concept and a TBox), the property is usually satisfiability of the concept (w.r.t. the TBox).

Definition 2.12. (Sound, complete)
Let $\mathcal{P} \subseteq \mathfrak{I}$ be a property. The tableau system S is called

- *sound for* $\mathcal{P}$ iff, for any $\Gamma \in \mathfrak{I}$, the existence of a saturated and clash-free S-tree for Γ implies that $\Gamma \in \mathcal{P}$;

- *complete for* $\mathcal{P}$ iff, for any $\Gamma \in \mathcal{P}$, there exists a saturated and clash-free S-tree for Γ.

It should be noted that the algorithmic treatment of tableau systems requires a stronger notion of completeness: an additional condition is needed to ensure that the out-degree of S-trees is appropriately bounded (see Definition 3.1 and Definition 4.2 below).

Taking into account the known soundness and completeness results for the $\mathcal{ALC}^U$ tableau algorithm described in Figure 1, it is straightforward to check that the tableau system $S_{\mathcal{ALC}^U}$ is sound and complete w.r.t. satisfiability of concepts. Note, in particular, that saturated S-trees for an input Γ are precisely those S-trees for Γ that can be obtained by exhaustive or infinite and *fair* rule application.

3. ExpTime Automata-based Decision Procedures from Tableau Systems

In this section, we define the class of "ExpTime-admissible" tableau systems. If such a tableau system is sound and complete for a property $\mathcal{P}$, then it gives rise to an ExpTime algorithm for deciding $\mathcal{P}$.[6] In the case where $\mathcal{P}$ is satisfiability of description logic concepts (w.r.t. a TBox), this means that the mere existence of an ExpTime-admissible tableau system for the DL implies an ExpTime upper-bound for concept satisfiability (w.r.t. TBoxes) in this DL. The ExpTime upper-bound is shown via a translation of the inputs of the ExpTime-admissible tableau system into certain automata working on *infinite* trees. For this reason, ExpTime-admissible tableau systems need *not* deal with the issue of termination. Indeed, non-terminating tableau algorithms such as the one for $\mathcal{ALC}^U$ introduced in Section 2.1 may yield ExpTime-admissible tableau systems.

Throughout this section, we consider a fixed set of inputs $\mathfrak{I}$ and a fixed tableau system $S = (\mathsf{NL}, \mathsf{GME}, \mathsf{EL}, k, \cdot^S, \mathcal{R}, \mathcal{C})$ for $\mathfrak{I}$, which is sound and complete w.r.t. some property $\mathcal{P}$.[6] As usual, the exponential upper-bound of deciding $\mathcal{P}$ is assumed to be in the "size" of the input $\Gamma \in \mathfrak{I}$. Thus, we assume that the set of inputs is equipped with a size function, which assigns to an input $\Gamma \in \mathfrak{I}$ a natural number, its size $|\Gamma|$.

3.1. Basic Notions

Recall that a tableau system S is sound and complete for a property $\mathcal{P}$ if, for any input Γ, we have $\Gamma \in \mathcal{P}$ iff there exists a (potentially infinite) saturated and clash-free S-tree for Γ. The fundamental idea for obtaining an ExpTime upper-bound for deciding $\mathcal{P}$ is to use automata on infinite trees to check for the existence of a clash-free and saturated S-tree for a given input Γ. More precisely, each input Γ is converted into a tree automaton $\mathcal{A}_\Gamma$ such that there exists a clash-free and saturated S-tree for Γ iff $\mathcal{A}_\Gamma$ accepts a non-empty language. Since tree automata work on trees of some fixed out-degree, this approach only works if the (size of the) input determines such a fixed out-degree for the S-trees to be considered. This motivates the following definition.

Definition 3.1. (p-complete)
Let p be a polynomial. The tableau system S is called *p-complete for $\mathcal{P}$* iff, for any $\Gamma \in \mathcal{P}$, there exists a saturated and clash-free S-tree for Γ with out-degree bounded by $p(|\Gamma|)$.

Throughout this section, we assume that there exists a polynomial p such that the fixed tableau system S is p-complete w.r.t. the property $\mathcal{P}$ under consideration.

The tableau system $S_{\mathcal{ALC}^U}$ defined in Section 2 is easily proved to be i-complete, with i being the identity function on the natural numbers: using the formulation of the rules, it is easily proved that the out-degree of every $S_{\mathcal{ALC}^U}$-tree for the input C is bounded by the number of concepts of the form $\exists r.D$ in $\mathsf{sub}(C)$ and thus also by the length $|C|$ of the concept C.

It should be noted that most standard description logic tableau algorithms [5] also exploit p-completeness of the underlying logic: although this is not made explicit in the formulation of the algorithm itself, it is usually one of the central arguments in termination proofs.[7] The intuition that p-completeness is *not* an artefact of using an automata-based approach is reinforced by the fact that a similar strengthening of

[6] More precisely, we must demand a slightly stronger version of completeness, as introduced in Definition 3.1 below.
[7] An exception are algorithms that treat qualifying number restrictions with numbers coded in binary in a naive way [16, 35]

completeness is needed in Section 4, where we construct tableau-based decision procedures from tableau systems.

To ensure that the automaton $\mathcal{A}_\Gamma$ can be computed and tested for emptiness in exponential time, we require the function $\cdot^S$ of the tableau system S and the rules of S to exhibit an "acceptable" computational behavior. This is captured by the following definition. In this definition, we assume that all patterns are appropriately encoded in some finite alphabet, and thus can be the input for a decision procedure. The *size of a pattern* P is the sum of the sizes of its global memory component and its node and edge labels, where the size of a node label (global memory component) is the sum of the sizes of its node label elements (global memory elements).

Definition 3.2. (ExpTime-admissible)
The tableau system S is called *ExpTime-admissible* iff the following conditions are satisfied:

1. S is admissible (see Definition 2.7);
2. $\mathsf{ini}_S(\Gamma)$ and $\mathsf{el}_S(\Gamma)$ can be computed in time exponential in $|\Gamma|$, and the size of each edge label in $\mathsf{el}_S(\Gamma)$ is polynomial in $|\Gamma|$;
3. the cardinality of $\mathsf{nle}_S(\Gamma)$ and the size of each global memory element in $\mathsf{nle}_S(\Gamma)$ is polynomial in $|\Gamma|$, and $\mathsf{nle}_S(\Gamma)$ can be computed in time exponential in $|\Gamma|$;
4. the cardinality of $\mathsf{gme}_S(\Gamma)$ and the size of each node label element in $\mathsf{gme}_S(\Gamma)$ is polynomial in $|\Gamma|$, and $\mathsf{gme}_S(\Gamma)$ can be computed in time exponential in $|\Gamma|$;
5. for each pattern P it can be checked in time exponential in the size of P whether, for all patterns P', $P' \sim P$ implies $\mathcal{R}(P') = \emptyset$;
6. for each pattern P it can be checked in time exponential in the size of P whether there is a clash-trigger $P' \in \mathcal{C}$ such that $P' \sim P$.

Note that Point 2 of ExpTime-admissibility implies that, for each $\Gamma \in \mathfrak{I}$, the cardinality of the sets $\mathsf{ini}_S(\Gamma)$ and $\mathsf{el}_S(\Gamma)$ are at most exponential in $|\Gamma|$. The cardinality of the set of node label elements $\mathsf{nle}_S(\Gamma)$ is explicitly required (in Point 3) to be polynomial. For the actual set of node labels (which are sets of node label elements), this yields an exponential upper-bound on its cardinality, but the size of each node label is polynomial in $|\Gamma|$. The same is true for the gobal memory component (Point 4). ExpTime-admissibility ensures that the size of each k-neighborhood T, x is polynomial in $|\Gamma|$ since

- p-completeness implies that we consider only S-trees T of out-degree bounded by $p(|\Gamma|)$, and thus the out-degree of each k-neighborhood is polynomial in $|\Gamma|$;
- k-neighborhoods have constant depth k (not depending on the input);
- the sizes of the global memory component and of edge and node labels are polynomial in $|\Gamma|$.

Thus, the fifth point ensures that the saturatedness condition can be checked in time exponential in $|\Gamma|$ for a given neighborhood T, x of T. The sixth point yields the same for clash-freeness.

Most standard tableau algorithms for ExpTime-complete DLs trivially satisfy the conditions of ExpTime-admissibility. For example, it is easy to show that the tableau system $S_{\mathcal{ALC}^U}$ defined in Section 2 is ExpTime-admissible. We have already shown admissibility of $S_{\mathcal{ALC}^U}$, and Point 2, 3, and 4 are immediate consequences of the definitions of $\mathsf{ini}_{S_{\mathcal{ALC}^U}}$, $\mathsf{nle}_{S_{\mathcal{ALC}^U}}$, $\mathsf{gme}_{S_{\mathcal{ALC}^U}}$, and $\mathsf{el}_{S_{\mathcal{ALC}^U}}$. To see that Points 5 and 6 are satisfied as well, first note that the definition of the rules and clash-triggers in $S_{\mathcal{ALC}^U}$ is invariant under isomorphism of patterns. For this reason, the decision problem in Point 5 reduces to checking whether a given pattern P is saturated (see the definition of this notion below Definition 2.6), and the decision problem in Point 6 reduces to checking whether a given pattern is a clash-trigger. As an example, we consider the rule handling existential restrictions. Let $P = ((V, E, n, \ell), \mu)$ be a pattern whose tree has root v_0, and assume that $\exists r.C \in n(v_0)$. This existential restriction contributes a set of patterns to $\mathcal{R}(P)$ iff $C \notin n(u)$ for all r-successors u of v_0. Obviously, this can be checked in time polynomial in the size of the pattern.

The remainder of the present section is concerned with converting ExpTime-admissible tableau systems into automata-based decision procedures, as outlined above. The major challenge is to bring together the different philosophies underlying tableau algorithms and automata-based approaches for deciding concept satisfiability: tableau algorithm actively try to *construct* a model for the input by applying rules, as reflected in the Definitions 2.9 and 2.10, whereas automata are based on the concept of "acceptance" of a tree, i.e., they verify whether a *given* tree actually describes a model. Of course, the emptiness test for the automaton then again checks whether such a tree exists. Due to these different perspectives, it is not straightforward to construct automata that directly check for the existence of S-trees for an input Γ. To overcome this problem, we first introduce the (less constructive) notion of S-trees *compatible with* Γ, and investigate the relationship of this notion to S-trees *for* Γ, as introduced in Definition 2.10.

Definition 3.3. (S-tree compatible with Γ)
Let Γ be an input and $T = ((V, E, n, \ell), \mu)$ an S-tree with root v_0. Then T is *compatible with* Γ iff it satisfies the following conditions:

1. $\mu \subseteq \wp(\mathsf{gme}_S(\Gamma))$;
2. $n(x) \subseteq \wp(\mathsf{nle}_S(\Gamma))$ for each $x \in V$;
3. $\ell(x, y) \in \mathsf{el}_S(\Gamma)$ for each $(x, y) \in E$;
4. there exists $(\Lambda, \nu) \in \mathsf{ini}_S(\Gamma)$ such that $\Lambda \subseteq n(v_0)$ and $\nu \subseteq \mu$;
5. the out-degree of T is bounded by $p(|\Gamma|)$.

Below, we will show that, given an ExpTime-admissible tableau system S that is sound and p-complete for some property $\mathcal{P}$ and an input Γ for S, we can construct a looping tree automaton of size exponential in the size of Γ that accepts exactly the saturated and clash-free S-trees compatible with Γ. Since the emptiness problem for looping tree automata can be decided in time polynomial (actually, linear) in the size of the automaton, this shows that the existence of saturated and clash-free S-trees compatible with Γ can be decided in exponential time. Since S is sound and p-complete for $\mathcal{P}$, we have $\Gamma \in \mathcal{P}$ iff there is a saturated and clash-free S-tree *for* Γ. Thus, we must investigate the connection between S-trees for Γ and S-trees compatible with Γ. This is done in the next lemma.

Lemma 3.1. There exists a clash-free and saturated S-tree that is *compatible with* Γ iff there exists a clash-free and saturated S-tree *for* Γ.

Proof:
The "if" direction is straightforward: let $T = ((V, E, n, \ell), \mu)$ be a clash-free and saturated S-tree for Γ. Since S is sound and p-complete for $\mathcal{P}$, we can w.l.o.g. assume that the out-degree of the tree of T is bounded by $p(|\Gamma|)$. It is not hard to show that T is compatible with Γ, i.e. satisfies Conditions 1 to 5 of Definition 3.3:

- Each initial S-tree satisfies Conditions 1, 2, and 3 of compatibility, and Condition 3 of admissibility ensures that rule application adds only global memory elements from $\mathsf{gme}_S(\Gamma)$, node label elements from $\mathsf{nle}_S(\Gamma)$, and edge labels from $\mathsf{el}_S(\Gamma)$.
- Each initial S-tree satisfies Condition 4 of compatibility, and rule application cannot delete elements from node labels or from the global memory component.
- Since we assume the out-degree of T to be bounded by $p(|\Gamma|)$, Condition 5 of compatibility is also satisfied.

Now for the "only if" direction. Let $T = (t, \mu)$ be a clash-free and saturated S-tree that is compatible with Γ, and let v_0 be the root of the tree $t = (V, E, n, \ell)$. To construct a clash-free and saturated S-tree for Γ, we first construct a (possibly infinite) sequence

$$T_1 \preceq T_2 \preceq T_3 \preceq \cdots$$

of S-trees for Γ such that $T_i \precsim_{\pi_i} T$ for all $i \geq 1$. The construction will be such that the injections π_i that yield $T_i \precsim T$ also build an increasing chain, i.e., π_{i+1} extends π_i for all $i \geq 1$. In the construction, we use a countably infinite set V' from which the nodes of the S-trees T_i are taken. We fix an arbitrary enumeration $x_0, x_1, \ldots$ of V', and write $x < y$ if $x \in V'$ occurs before $y \in V'$ in this enumeration. We then proceed as follows:

- Since T is compatible with Γ, there exists $(\Lambda, \nu) \in \mathsf{ini}_S(\Gamma)$ such that $\Lambda \subseteq n(v_0)$ and $\nu \subseteq \mu$. Define T_1 to be the initial S-tree $((\{x_0\}, \emptyset, \{x_0 \mapsto \Lambda\}, \emptyset), \nu)$. Obviously, $T_1 \precsim_{\pi_1} T$ for $\pi_1 := \{x_0 \mapsto v_0\}$.
- Now, assume that $T_i \precsim_{\pi_i} T$ is already constructed. If T_i is saturated, then T_i is the last S-tree in the sequence. Otherwise, choose the least node x in the tree of T_i (w.r.t. the fixed ordering $<$ on V') such that $P \sim T_i, x$ for some pattern P that is not saturated, i.e. there exists a rule $P \rightarrow_{\mathcal{R}} \{P_1, \ldots, P_m\}$. Since $T_i \precsim_{\pi_i} T$, we have $P \precsim T, \pi_i(x)$. Since T is saturated, the pattern $T, \pi_i(x)$ is saturated. By Condition 2 of admissibility, we have $P_j \precsim T, \pi_i(x)$ for some j with $1 \leq j \leq m$. We apply the rule $P \rightarrow_{\mathcal{R}} \{P_1, \ldots, P_m\}$ to x in T_i such that $P_j \sim T_{i+1}, x$. If the tree of T_{i+1} contains new nodes, then they are taken without loss of generality from V'. Admissibility yields $T_i \preceq T_{i+1}$ and the fact that $P_j \precsim T, \pi_i(x)$ implies that we can define an injection π_{i+1} extending π_i such that $T_{i+1} \precsim_{\pi_{i+1}} T$.

In the definition of the clash-free and saturated S-tree T^* for Γ, we distinguish two cases:

1. if the constructed sequence is finite and T_n is the last S-tree in the sequence, then set $T^* := T_n$;
2. otherwise, let T^* be the S-tree T^ω obtained from the sequence $T_1, T_2, \ldots$ as in Case 3 of Definition 2.10.

In both cases, T^* is obviously an S-tree for Γ by definition. In addition, we have $T^* \precsim_\pi T$ where π is the injection obtained as the union of the injections π_i for $i \geq 1$.

It remains to be shown that T^* is clash-free and saturated. We concentrate on the second case, where $T^* = T^\omega$, since the first case is similar, but simpler. Clash-freeness is an easy consequence of $T^* \precsim T$. In fact, by Condition 4 of admissibility, clash-freeness of T implies that $T^* \precsim T$ is also clash-free.

To show saturatedness of T^*, we must look at T^* and its relationship to the S-trees T_i in more detail. Since $T_i \preceq T^* \precsim T$ and the out-degree of the tree of T is bounded by $p(|\Gamma|)$, the out-degrees of the trees of T_i and T^* are also bounded by $p(|\Gamma|)$. For a given node x of the tree of T^*, we consider its k-neighborhood T^*, x. Since the rules of S only add nodes or elements of node labels or of the global memory component (see Condition 1 in the definition of admissibility), and since the out-degree of x is bounded by $p(|\Gamma|)$ and the sets $\mathsf{nle}_S(\Gamma)$ and $\mathsf{gme}_S(\Gamma)$ are finite, there is an i such that x is a node of T_i and "the neighborhood of x does not change after step i," i.e., $T_i, x = T_{i+1}, x = \ldots = T^*, x$.

Now assume that T^* is not saturated, i.e., there exists a node x in the tree of T^* to which a rule applies, i.e., $P \sim T^*, x$ for some pattern P with $\mathcal{R}(P) \neq \emptyset$. Let i be such that $T_i, x = T_{i+1}, x = \ldots = T^*, x$. Thus, for $j \geq i$, a rule applies to the node x in the tree of T_i. In the construction of the sequence $T_1, T_2, T_3, \ldots$, we apply a rule only to the least node to which a rule is applicable. Consequently, from the ith step on, we only apply rules to nodes $y \leq x$. Since there are only finitely many such nodes (see the definition of the order $<$ above), there is one node $y \leq x$ to which rules are applied infinitely often. However, each rule application strictly increases the global memory component, the number of nodes in the k-neighborhood of y, or the label of a node in this k-neighborhood. This contradicts the fact that the out-degree of the trees of the T_i is bounded by $p(|\Gamma|)$, all node labels are subsets of the finite set $\mathsf{nle}_S(\Gamma)$, and all global memory components are subsets of the finite set $\mathsf{gme}_S(\Gamma)$. □

3.2. Accepting Compatible S-trees Using Looping Automata

Recall that we assume our tableau system S to be sound and p-complete w.r.t. a property $\mathcal{P}$. By Lemma 3.1, to check whether an input has property $\mathcal{P}$, it thus suffices to verify the existence of a saturated and clash-free S-tree that is compatible with Γ. In this section, we show how this can be done using an automata-based approach.

As usual, the automata work on d-ary infinite trees (for some fixed natural number d) whose nodes are labeled by elements of a finite label set and whose edges are ordered, i.e., we can talk about the i-th son of a node. To be more precise, let M be a set and $d \geq 1$. A *d-ary infinite M-tree* is a mapping $t : \{1, \ldots, d\}^* \to M$ that labels each node $\alpha \in \{1, \ldots, d\}^*$ with $t(\alpha) \in M$. Intuitively, the node αi is the i-th child of α. We use ϵ to denote the empty word, corresponding to the root of the tree.

Definition 3.4. (Looping tree automata)
A *looping tree automaton* $\mathcal{A} = (Q, M, I, \Delta)$ working on d-ary M-trees consists of a finite set Q of states, a finite alphabet M, a set $I \subseteq Q$ of initial states, and a transition relation $\Delta \subseteq Q \times M \times Q^d$.

A *run* of $\mathcal{A}$ on an M-tree t is a mapping $R : \{1, \ldots, d\}^* \to Q$ such that $R(\epsilon) \in I$ and

$$(R(\alpha), t(\alpha), R(\alpha 1), \ldots, R(\alpha d)) \in \Delta$$

for each $\alpha \in \{1, \ldots, d\}^*$. The *language* of d-ary M-trees *accepted by* $\mathcal{A}$ is

$$L(\mathcal{A}) := \{t \mid \text{there is a run of } \mathcal{A} \text{ on the } d\text{-ary } M\text{-tree } t\}.$$

In contrast to patterns, whose trees can have depth up to k, transitions of looping tree automata consider only subtrees of depth 1. This makes it hard to give a direct translation of an input into a looping automaton that accepts the saturated and clash-free S-trees that are compatible with this input. For this reason, we first introduce a new type of tree automata "with transitions of depth k," and show that they can be translated into looping tree automata.

For a set Q and integers k, d, we denote the set of all (full) d-ary trees of depth k with node labels in Q by $\mathsf{T}_d^k(Q)$. If R is an infinite d-ary Q-tree and x a node in R, then R, x denotes the k-neighborhood of x, i.e., the full d-ary subtree of t of depth k with root x.

Definition 3.5. (Looping tree automata with transitions of depth k)
A *looping tree automaton* $\mathcal{A} = (Q, M, I, \Delta)$ *with transitions of depth* k working on d-ary M-trees consists of a finite set Q of states, a finite alphabet M, a set $I \subseteq \mathsf{T}_d^k(Q)$ of initial trees, and a set of transitions $\Delta \subseteq M \times \mathsf{T}_d^k(Q)$.

A *run* of $\mathcal{A}$ on an M-tree t is a mapping $R : \{1, \ldots, d\}^* \to Q$ (i.e., a d-ary Q-tree) such that $R, \epsilon \in I$ and

$$(t(\alpha), (R, \alpha)) \in \Delta$$

for each node α in $\{1, \ldots, d\}^*$. The *language* of d-ary M-trees *accepted by* $\mathcal{A}$ is

$$L(\mathcal{A}) := \{t \mid \text{there is a run of } \mathcal{A} \text{ on the } d\text{-ary } M\text{-tree } t\}.$$

It is easy to see that *normal* looping tree automata (as introduced in Definition 3.4) basically consitute the special case where the transitions are of depth 1. The following lemma shows that looping tree automata of depth $k > 1$ are *not* more powerful than normal looping tree automata. We define the *size* of a tree automaton $\mathcal{A} = (Q, M, I, \Delta)$ as $|\mathcal{A}| := |Q| + |M| + |I| + |\Delta|$.

Lemma 3.2. Any looping tree automaton $\mathcal{A}$ of depth $k > 1$ working on d-ary M-trees can be reduced in time polynomial in $|\mathcal{A}|^d$ to a normal looping tree automaton that accepts the same language.

Proof:
Let $\mathcal{A} = (Q, M, I, \Delta)$ be a looping tree automaton with transitions of depth k. The normal looping tree automaton $\mathcal{B} = (P, M, J, \Theta)$ is defined as follows:

- $P := \{t \mid (m, t) \in \Delta \text{ for some } m \in M\}$;
- $J := I \cap P$;
- $(t_0, m, t_1, \ldots, t_d) \in \Theta$ iff $(m, t_0) \in \Delta$ and $t_1, \ldots, t_d \in P$ are such that t_i coincides with the subtree of t_0 at node i up to depth $k - 1$.

Clearly, $|P|$ is bounded by $|\Delta|$, $|J|$ is bounded by $|I|$, and $|\Theta|$ is bounded by $|P|^d \cdot |\Delta|$. It is also easy to see that $\mathcal{B}$ can be computed in time polynomial in $|\mathcal{A}|^d$. It remains to be shown that $L(\mathcal{A}) = L(\mathcal{B})$. First, assume that $t \in L(\mathcal{A})$ and that R is a run of $\mathcal{A}$ on t. It is easy to see that the following is a run of $\mathcal{B}$ on t:

$$S : \{1, \ldots, d\}^* \to P : \alpha \mapsto R, \alpha.$$

Second, assume that R' is a run of $\mathcal{B}$ on t. If p is an element of $P \subseteq \mathsf{T}_d^k(Q)$, then we denote the label of its root by $\mathsf{rl}(p)$. We claim that the following is a run of $\mathcal{A}$ on t:

$$S' : \{1, \ldots, d\}^* \to Q : \alpha \mapsto \mathsf{rl}(R'(\alpha)).$$

This is an easy consequence of the fact that $S', \alpha = R'(\alpha)$ holds for all $\alpha \in \{1, \ldots, d\}$. □

The next obstacle on our way towards translating an input into a looping automaton that accepts the saturated and clash-free S-trees that are compatible with this input is that the S-trees introduced in Section 2 are not of a fixed arity d and that their edges are labeled, but not ordered. It is, however, not hard to convert S-trees compatible with a given input into d-ary M-trees for appropriate d and M. This is achieved by (i) "padding" with additional dummy nodes, and (ii) representing edge labels via node labels.

Definition 3.6. (Padding)
Let $\Gamma \in \mathfrak{I}$ be an input and $t = (V, E, n, \ell)$ be the tree component of an S-tree compatible with Γ. Let v_0 denote the root of t. For each $x \in V$, we use $d(x)$ to denote the out-degree of x in t. We assume that the successors of each node $x \in V$ are linearly ordered and that, for each node $x \in V \setminus \{v_0\}$, $s(x) = i$ iff x is the i-th successor of its predecessor. We inductively define a function m from $\{1, \ldots, p(|\Gamma|)\}^*$ to $V \cup \{\sharp\}$ (where $\sharp \notin V$) as follows:[8]

- $m(\epsilon) = v_0$;
- if $m(\alpha) = x \in V$, $(x, y) \in E$, and $s(y) = i$, then $m(\alpha i) = y$;
- if $m(\alpha) = x \in V$ and $d(x) < i$, then $m(\alpha i) = \sharp$;
- if $m(\alpha) = \sharp$, then $m(\alpha i) = \sharp$ for all $i \in \{1, \ldots, p(|\Gamma|)\}$.

Let $\mathsf{tl}_S(\Gamma)$ denote the set $(\wp(\mathsf{nle}_S(\Gamma)) \times \mathsf{el}_S(\Gamma)) \cup \{(\sharp, \sharp)\}$. The *padding* Π_t of t is the $p(|\Gamma|)$-ary $\mathsf{tl}_S(\Gamma)$-tree defined by setting

1. $\Pi_t(\epsilon) = (n(v_0), e_0)$ where e_0 is an arbitrary (but fixed) element of $\mathsf{el}_S(\Gamma)$;
2. $\Pi_t(\alpha) = (n(x), \Theta)$ if $\alpha \neq \epsilon$, $m(\alpha) = x \neq \sharp$, and $\ell(y, x) = \Theta$ where y is the predecessor of x in t;
3. $\Pi_t(\alpha) = (\sharp, \sharp)$ if $m(\alpha) = \sharp$.

Given the tree component t of a pattern for Γ of out-degree at most $p(|\Gamma|)$, its *k-padding* Π_t^k is the full $p(|\Gamma|)$-ary $\mathsf{tl}_S(\Gamma)$-tree *of depth k* obtained by adding the missing nodes with label $(\sharp, \sharp)$ and by representing edge labels via node labels, analogous to the definition of Π_t above.

The final obstacle on our way towards translating an input into a looping automaton that accepts the saturated and clash-free S-trees that are compatible with this input is the presence of the global memory component in our framework. Transitions of looping automata (even if they are of depth k) are *local*, whereas the notion of saturatedness involves the *global* memory component. For this reason, we define for each input $\Gamma \in \mathfrak{I}$ and each $\mu \subseteq \mathsf{gme}_S(\Gamma)$ a looping automaton $\mathcal{A}_\Gamma^\mu$ that accepts a non-empty language iff there exists a saturated and clash-free S-tree that is compatible with Γ and has global memory component μ.

[8]Note that the p used here stems from p-completeness.

Definition 3.7. (Automaton for input Γ and global memory component μ)
Let $\Gamma \in \mathfrak{I}$ be an input, $h = p(|\Gamma|)$, and $\mu \subseteq \mathsf{gme}_S(\Gamma)$. The looping automaton $\mathcal{A}^\mu_\Gamma = (Q, M, I, \Delta)$ with transitions of depth k is defined as follows:

- $Q := M := \mathsf{tl}_S(\Gamma)$;
- I consists of all elements t of $\mathsf{T}^k_h(Q)$ whose root label is of the form $\mathsf{rl}(t) = (\Psi, e_0)$ where Ψ is such that there exists a tuple $(\Lambda, \nu) \in \mathsf{ini}_S(\Gamma)$ with $\Lambda \subseteq \Psi$ and $\nu \subseteq \mu$.
- $(m, t) \in \Delta$ iff the following two conditions are satisfied:
  1. $m = \mathsf{rl}(t)$;
  2. either all nodes of t are labeled with $(\sharp, \sharp)$ or there is a pattern $P^* = (s, \mu)$ that satisfies the following conditions:
     (a) $t = \Pi^k_s$;
     (b) for each pattern P with $P \sim P^*$, P is saturated (i.e. $\mathcal{R}(P) = \emptyset$);
     (c) for each pattern $P \in \mathcal{C}$, we have $P \not\sim P^*$.

The following lemma shows that the automaton $\mathcal{A}^\mu_\Gamma$ accepts exactly the paddings of saturated and clash-free S-trees that are compatible with Γ and have global memory component μ. Consequently, it accepts a non-empty set of trees iff there exists a saturated and clash-free S-tree that is compatible with Γ and has global memory component μ.

Lemma 3.3. Let $\Gamma \in \mathfrak{I}$ be an input and $\mu \subseteq \mathsf{gme}_S(\Gamma)$. Then

$$L(\mathcal{A}^\mu_\Gamma) = \{\Pi_t \mid (t, \mu) \text{ is a saturated and clash-free } S\text{-tree compatible with } \Gamma\}.$$

Proof:
First, assume that (t, μ) is a saturated and clash-free S-tree compatible with Γ. We claim that Π_t itself is a run of $\mathcal{A}^\mu_\Gamma$ on Π_t. In fact, $\Pi_t, \epsilon \in I$ is an immediate consequence of the definition of padding and Condition 4 in the definition of S-trees compatible with Γ. Now, consider some node α of Π_t. The first condition in the definition of Δ is satisfied since we have Π_t as run on itself. Thus, consider the second condition. If $\Pi_t(\alpha) = (\sharp, \sharp)$, then the definition of padding implies that all the nodes below α also have label $(\sharp, \sharp)$, and thus the second condition in the definition of Δ is satisfied. Otherwise, it is easy to see that the pattern P^* defined by (a) in the second condition of the definition of Δ is a k-neighborhood in t. Since t is saturated and clash-free, P^* thus satisfies (b) and (c) as well. This completes the proof that Π_t is a run of $\mathcal{A}^\mu_\Gamma$ on Π_t, and thus shows that $\Pi_t \in L(\mathcal{A}^\mu_\Gamma)$.

Second, assume that $\widehat{t}$ is a tree accepted by $\mathcal{A}^\mu_\Gamma$. Because of the first condition in the definition of Δ, $\widehat{t}$ itself is a run of $\mathcal{A}^\mu_\Gamma$ on $\widehat{t}$. The definitions of Q, I, and Δ imply that there is an S-tree $T = (t, \mu)$ compatible with Γ such that $\Pi_t = \widehat{t}$. The tree t can be obtained from $\widehat{t}$ by "reversing" the padding procedure. It remains to be shown that (t, μ) is saturated and clash-free. Thus, consider a node x of t, and let α be the corresponding node in $\Pi_t = \widehat{t}$. Since x is a node in t, the node α has a label different from $(\sharp, \sharp)$. It is easy to see that the pattern P^* defined by (a) in the second condition in the definition of the transition relation coincides with T, x. Thus (b) and (c) in this condition imply that no rule and no clash-trigger is applicable to x. $\square$

We are now ready to prove the main result of this section: the ExpTime upper-bound induced by ExpTime-admissible tableau systems.

Theorem 3.1. Let $\mathfrak{I}$ be a set of inputs, $\mathcal{P} \subseteq \mathfrak{I}$ a property, and p a polynomial. If there exists an ExpTime-admissible tableau system S for $\mathfrak{I}$ that is sound and p-complete for $\mathcal{P}$, then $\mathcal{P}$ is decidable in ExpTime.

Proof:
Let $\Gamma \in \mathfrak{I}$ be an input. To decide whether $\Gamma \in \mathcal{P}$, we construct for each $\mu \subseteq \mathsf{gme}_S(\Gamma)$ the automaton $\mathcal{A}^{\mu}_{\Gamma}$. By Lemmas 3.1 and 3.3, $\Gamma \in \mathcal{P}$ iff at least one of these automata accepts a non-empty language.

It remains to be shown that this algorithm can be executed in exponential time. Let $n = |\Gamma|$ and $h = p(|\Gamma|)$. To see that each automaton $\mathcal{A}^{\mu}_{\Gamma}$ can be constructed in time exponential in n, note that, by Conditions 2 and 3 of ExpTime-admissibility, we can compute $\wp(\mathsf{nle}_S(\Gamma))$ and $\mathsf{el}_S(\Gamma)$ in time exponential in n, and thus the same holds for $\mathsf{tl}_S(\Gamma) = Q = M$. By Condition 2, to show that I can be computed in exponential time it suffices to show that $\mathsf{T}^k_h(Q)$ is of size exponential in n. This is the case since $|\mathsf{T}^k_h(Q)| = |Q|^{h^{k+1}-1}$, $|Q|$ is exponential in n, h is polynomial in n, and k is a constant. The transition relation Δ can be computed in exponential time due to the Conditions 5 and 6 of ExpTime-admissibility and the facts that $|\Delta| \leq |M| \cdot |\mathsf{T}^k_h(Q)|$, p is a polynomial and k is a constant. Since the automaton $\mathcal{A}^{\mu}_{\Gamma}$ can be computed in exponential time, its size is at most exponential in $|\Gamma|$. Thus, Lemma 3.2 and the fact that the emptiness test for looping tree automata can be realized in polynomial time [36] imply that emptiness of each automaton $\mathcal{A}^{\mu}_{\Gamma}$ can be tested in time exponential in the input. By Condition 4 of ExpTime-admissibility we can enumerate all global memory components $\mu \subseteq \mathsf{gme}_S(\Gamma)$ in exponential time, and there are exponentially many of them. Thus, the algorithm performs exponentially many ExpTime tests, which is still in ExpTime. □

Since we have shown that the tableau system $S_{\mathcal{ALC}^U}$ is ExpTime-admissible as well as sound and p-complete (for some polynomial p) for satisfiability of $\mathcal{ALC}^U$-concepts, we can immediately put Theorem 3.1 to work:

Corollary 3.1. $\mathcal{ALC}^U$-concept satisfiability is in ExpTime.

4. Tableau-based Decision Procedures from Tableau Systems

The tableau systems introduced in Section 2.2 cannot immediately be used as tableau-based decision procedures since rule application need not terminate. The purpose of this section is to show that, under certain natural conditions, the addition of a straightforward cycle detection mechanism turns them into (terminating) decision procedures. The resulting procedures are structurally similar to standard tableau-based algorithms for description logics, such as the ones underlying systems like FaCT and RACER. In contrast to the ExpTime algorithm constructed in the previous section, the procedures obtained here are usually not worst-case optimal—a weakness that is shared by almost all tableau algorithms for ExpTime-complete logics, and that is usually viewed as the price one has to pay for more easily implementable and optimizable decision procedures.

Fix a set of inputs $\mathfrak{I}$ and a tableau system $S = (\mathsf{NLE}, \mathsf{GME}, \mathsf{EL}, k, \cdot^S, \mathcal{R}, \mathcal{C})$ for $\mathfrak{I}$. As in the previous section, we require that S has a number of computational properties. Since we do consider decidability rather than complexity issues in this section, it is sufficient for our purposes to impose effectiveness (and not efficiency) constraints. We start with modifying Definition 3.2:

Definition 4.1. (Recursive Tableau System)
S is called *recursive* iff the following conditions are satisfied:

1. S is admissible (see Definition 2.7);
2. $\mathsf{ini}_S(\Gamma)$ can be computed effectively;
3. for each pattern P it can be checked effectively whether, for all patterns P', $P' \sim P$ implies $\mathcal{R}(P') = \emptyset$; if this is not the case, then we can effectively determine a rule
$$P' \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$$
and a bijection π such that $P' \sim_\pi P$.
4. for each pattern P it can be checked effectively whether there is a clash-trigger $P' \in \mathcal{C}$ such that $P' \sim P$.

The main difference between this definition and Definition 3.2 is Condition 3, which now requires that, besides checking the applicability of rules, we can effectively apply at least one rule whenever some rule is applicable at all. Another difference is that we do not actually need to compute the sets $\mathsf{el}_S(\Gamma)$, $\mathsf{nle}_S(\Gamma)$, and $\mathsf{gme}_S(\Gamma)$ in order to apply rules.

Analogously to the case of ExpTime-admissibility, it can be verified that the tableau system $S_{\mathcal{ALC}^U}$ is recursive. In particular, for the second part of Condition 3 we can again use the fact that the rules of $S_{\mathcal{ALC}^U}$ are invariant under isomorphism of patterns: this means that it suffices to compute, for a given non-saturated pattern P, a set of patterns $\{P_1, \ldots, P_m\}$ such that $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$. It is easy to see that this can be effectively done for the rules of $S_{\mathcal{ALC}^U}$.

We now define a more relaxed variant of Definition 3.1.

Definition 4.2. (f-complete)
Let $f : \mathbb{N} \to \mathbb{N}$ be a recursive function. The tableau system S is called *f-complete for $\mathcal{P}$* iff, for any $\Gamma \in \mathcal{P}$, there exists a saturated and clash-free S-tree for Γ with out-degree bounded by $f(|\Gamma|)$.

Since we have already shown that $S_{\mathcal{ALC}^U}$ is p-complete for some polynomial p, $S_{\mathcal{ALC}^U}$ is clearly f-complete for the (computable) function f induced by the polynomial p.

In order to implement a cycle detection mechanism, we introduce the notion of blocking: given an S-tree $T = (t, \mu)$, where $t = (V, E, n, \ell)$, we denote by E^* the transitive and reflexive closure of E and say that $x \in V$ is *blocked* iff there exist $u, v \in V$ such that

- uE^*x and vE^*x;
- uE^*v and the path from u to v is of length $\geq k$;
- $(T, u)_{k-1} \sim (T, v)_{k-1}$, where $(T, u)_{k-1}$ and $(T, v)_{k-1}$ denote the $k-1$ neighbourhoods of u and v in T, respectively.

Preconditions: Let $\mathfrak{I}$ be a set of inputs, $\mathcal{P} \subseteq \mathfrak{I}$ a property, f a recursive function, and S a recursive tableau system for $\mathfrak{I}$ that is sound and f-complete for $\mathcal{P}$.

Algorithm: Return true on input $\Gamma \in \mathfrak{I}$ if the procedure tableau(T) defined below returns true for at least one initial S-tree T for Γ. Otherwise return false.

procedure tableau(T)

If $P \sim T, x$ for some $P \in \mathcal{C}$ and node x in T or the out-degree of T exceeds $f(|\Gamma|)$,
then return false.

If no rule is applicable to a non-blocked node x in T,
then return true.

Take a a non-blocked node x in T and a rule $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$ with $P \sim T, x$.

Let T_i be the result of applying the above rule such that $P_i \sim T_i, x$, for $1 \leq i \leq m$.

If at least one of tableau(T_1), tableau$(T_2), \ldots,$ tableau(T_m) returns true,
then return true.

Return false.

Figure 2. Decision procedure for $\mathcal{P}$.

Note that, for $k = 1$, this blocking condition reduces to $u \neq v$ and $n(u) = n(v)$, which corresponds to the well-known "equality-blocking" technique that is used in various DL tableau algorithms [19, 5]. For $k = 2$, we obtain a more general variant of the "double-blocking" mechanism used for description logics such as $\mathcal{SHIQ}$ [20]. Our version is more general since, in the double-blocking variant, the isomorphic 2-neighborhoods in the third item above would be smaller and contain only a single node on depth 1.

The tableau-based decision procedure for $\mathcal{P}$ induced by the tableau system S is described in Figure 2. Note that the selection of rules and nodes in the procedure tableau is "don't care" non-deterministic: for the soundness and completeness of the algorithm, it does not matter which rule we apply when to which node.

Let us verify that the individual steps performed by the algorithm in Figure 2 are actually effective:

- the initial trees for an input Γ can be computed effectively, since $\mathsf{ini}_S(\Gamma)$ can be computed effectively by Condition 2 of Definition 4.1;
- the condition in the first "if" statement can be checked effectively by Condition 4 of Definition 4.1 and since f is a recursive function;
- the applicability of rules can be checked by the first part of Condition 3 of Definition 4.1;
- finally, that we can effectively take a rule and apply it to a node x follows from the second part of Condition 3 of Definition 4.1.

We now turn to termination, soundness, and completeness of the algorithm.

Lemma 4.1. (Termination)
Suppose the preconditions of Figure 2 are satisfied. Then the algorithm of Figure 2 terminates for any input $\Gamma \in \mathfrak{I}$.

Proof:
Let $\Gamma \in \mathfrak{I}$. The number of initial trees for Γ is finite and can be computed effectively. Hence, it is sufficient to show that the procedure $\mathsf{tableau}$ terminates on any initial tree for Γ. For each step in which the procedure does not immediately return true or false, nodes are added to the tree, $n(x)$ properly increases for some nodes x, or μ properly increases (due to Condition 1 of admissibility). Hence, since $n(x) \subseteq \wp(\mathsf{nle}_S(\Gamma))$ for any node x and $\mu \subseteq \wp(\mathsf{gme}_S(\Gamma))$ for any tree constructed during a run of $\mathsf{tableau}$, it is sufficient to show that both the out-degree and the depth of the trees constructed is bounded. The out-degree of the trees is bounded by $f(|\Gamma|)$ (more precisely, as soon as one rule application yields a tree with out-degree larger than $f(|\Gamma|)$, the algorithm returns false in the next step). Due to the blocking condition, the length of E-paths does not exceed the number of pairwise non-isomorphic labelled trees (V, E, n, l) of depth $\leq k-1$ and outdegree $\leq f(|\Gamma|)$ such that $\mathsf{ran}(n) \subseteq \wp(\mathsf{nle}_S(\Gamma))$ and $\mathsf{ran}(\ell) \subseteq \mathsf{el}_S(\Gamma))$. □

Lemma 4.2. (Soundness)
Suppose the preconditions of Figure 2 are satisfied. If the algorithm of Figure 2 returns true on input Γ, then $\Gamma \in \mathcal{P}$.

Proof:
Suppose the algorithm returns true on input Γ. Then the algorithm terminates with a clash-free S-tree $T = (t, \mu)$, $t = (V, E, n, \ell)$, whose out-degree does not exceed $f(|\Gamma|)$ and such that no rule is applicable to a non-blocked node in T. As S is sound for $\mathcal{P}$, it is sufficient to show that there exists a saturated and clash-free S-tree for Γ. To this end, we construct a clash-free and saturated S-tree

$$T' = ((V', E', n', \ell'), \mu)$$

that is compatible with Γ (from which, by Lemma 3.1, we obtain a clash-free and saturated S-tree for Γ). Say that a node $x \in V$ is *directly blocked* if it is blocked but its predecessor is not blocked. For any such x pick a y with yE^*x such that the path from y to x has length $\geq k$ and $(T, x)_{k-1} \sim (T, y)_{k-1}$, and say that x is *blocked by* y. Now, V' consists of all non-empty sequences $\langle v_0, x_1, \ldots, x_n \rangle$, where v_0 is the root of V, the $x_1, \ldots, x_n \in V$ are directly blocked or not blocked, and $(x_i, x_{i+1}) \in E$ if x_i is not blocked or x_i is blocked by some $y \in V$ such that $(y, x_{i+1}) \in E$. Define E' by setting, for $\vec{x} = \langle v_0, x_1, \ldots, x_n \rangle \in V'$ and $\vec{y} \in V'$, $(\vec{x}, \vec{y}) \in E'$ iff there exists x_{n+1} such that $\vec{y} = \langle v_0, x_1, \ldots, x_n, x_{n+1} \rangle$. Define n' by setting $n'(\langle v_0, x_1, \ldots, x_n \rangle) = n(x_n)$. Finally, define ℓ' by

- $\ell'(\langle v_0, x_1, \ldots, x_n \rangle, \langle v_0, x_1, \ldots, x_n, x_{n+1} \rangle) = \ell(x_n, x_{n+1})$ if x_n is not blocked;
- $\ell'(\langle v_0, x_1, \ldots, x_n \rangle, \langle v_0, x_1, \ldots, x_n, x_{n+1} \rangle) = \ell(y, x_{n+1})$ if x_n is blocked and y blocks x_n.

We show that T' is a clash-free and saturated S-tree which is compatible with Γ. Compatibility is readily checked using the definition of T'. Since T is clash-free and no rule is applicable to a non-blocked node of T, we can prove clash-freeness and saturatedness of T' by showing that any S-pattern P that matches $T', \vec{x}$ for some node $\vec{x}$ in t' also matches a T, x for some non-blocked node x in t. But one can easily show by induction on m for $0 \leq m \leq k$ and any $\langle v_0, x_0, \ldots, x_n \rangle \in V'$ that

- $(T', \langle v_0, x_1, \ldots, x_n\rangle)_m \sim (T, x_n)_m$ if x_n is not blocked and
- $(T', \langle v_0, x_1, \ldots, x_n\rangle)_m \sim (T, y)_m$ if x_n is blocked by y.

The base case ($m = 0$) is trivial. Thus, consider the induction step $m-1 \to m$ for $m-1 < k$.[9]

First, consider the case where x_n is not blocked. By definition of T', the label of the root node $\langle v_0, x_1, \ldots, x_n\rangle$ of $(T', \langle v_0, x_1, \ldots, x_n\rangle)_m$ coincides with the label of the root node x_n of $(T, x_n)_m$. Thus, it is sufficient to show that the respective successor nodes have isomorphic neighborhoods of depth $m-1$. Let z be a successor of x_n in T, and let $\langle v_0, x_1, \ldots, x_n, z\rangle$ be the corresponding successor of $\langle v_0, x_1, \ldots, x_n\rangle$ in T' (which exists since x_n is not blocked). If z is not blocked, the induction yields $(T', \langle v_0, x_1, \ldots, x_n, z\rangle)_{m-1} \sim (T, z)_{m-1}$ and we are done. Otherwise, z is blocked by some node y. By induction, we know that $(T', \langle v_0, x_1, \ldots, x_n, z\rangle)_{m-1} \sim (T, y)_{m-1}$. In addition, the facts that y blocks z and that $m-1 \leq k-1$ implies that $(T, y)_{m-1} \sim (T, z)_{m-1}$. Thus, we also have $(T', \langle v_0, x_1, \ldots, x_n, z\rangle)_{m-1} \sim (T, z)_{m-1}$ in this case.

Second, consider the case where x_n is blocked by some node y. Let $\langle v_0, x_1, \ldots, x_i, y\rangle$ be the node in T' corresponding to y. By construction of T' we have $(T', \langle v_0, x_1, ..., x_i, y\rangle)_m \sim (T', \langle v_0, x_1, ..., x_n\rangle)_m$. Thus, it is sufficient to show that $(T', \langle v_0, x_1, \ldots, x_i, y\rangle)_m \sim (T, y)_m$. Since y is not blocked, this is an instance of the first case in the induction step, which we have already shown.

This finishes the induction proof. It follows that from $P \sim T', \langle v_0, x_1, \ldots, x_n\rangle$, P an S-pattern, we can deduce $P \sim T, x_n$ if x_n is not blocked and $P \sim T, y$ if x_n is blocked by y. □

Lemma 4.3. (Completeness)
Suppose the preconditions of Figure 2 are satisfied. If $\Gamma \in \mathcal{P}$, then the algorithm of Figure 2 returns `true` on input Γ.

Proof:
Suppose $\Gamma \in \mathcal{P}$. Since S is f-complete for $\mathcal{P}$, there exists a clash-free and saturated S-tree $T = (t, \mu)$, $t = (V, E, n, \ell)$, for Γ whose out-degree does not exceed $f(|\Gamma|)$. We use T to "guide" the algorithm to an S-tree of out-degree at most $f(|\Gamma|)$ in which no clash-trigger applies and no rule is applicable to a non-blocked node. This will be done in a way such that all constructed S-trees T' satisfy $T' \precsim T$.

For the start, we need to choose an appropriate initial S-tree T_1. Let v_0 be the root of t. Since S-trees for Γ are also compatible with Γ, the definition of compatibility implies that there exists $(\Lambda, \nu) \in \mathsf{ini}_S(\Gamma)$ such that $\Lambda \subseteq n(v_0)$ and $\nu \subseteq \mu$. Define T_1 to be the initial S-tree $((\{v_0\}, \emptyset, \{v_0 \mapsto \Lambda\}, \emptyset), \nu)$. Clearly, $T_1 \precsim T$. We start the procedure `tableau` with the tree T_1.

Now suppose that `tableau` is called with some S-tree T' such that $T' \precsim T$. If no rule is applicable to a non-blocked node in T', we are done: since $T' \precsim T$ and T is clash-free and of out-degree at most $f(|\Gamma|)$, the same holds for T'. Now suppose that a rule is applicable to a non-blocked node in T'. Assume that the `tableau` procedure has chosen the rule $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$ with $P \sim T', x$. Since $T' \precsim_\tau T$ for some τ, we have $P \precsim T, \tau(x)$. Since T is saturated, $T, \tau(x)$ is saturated. By Condition 2 of admissibility, we have $P_j \precsim T, \tau(x)$ for some $j, 1 \leq j \leq m$. So we "guide" the `tableau` procedure to continue exploring the S-tree T'_j obtained from T' by applying the rule $P \to_{\mathcal{R}} \{P_1, \ldots, P_m\}$ such that $P_j \sim T'_j, x$. Now, $P_j \precsim T, \tau(x)$ implies $T'_j \precsim T$.

[9] Note that the induction step goes through only for $m-1 < k$ because the blocking condition ensures isomorphism of neighborhoods only up to depth $k-1$.

Since the $\mathsf{tableau}$ procedure terminates on any input, the "guidance" process will also terminate and thus succeeds in finding an S-tree of out-degree at most $f(|\Gamma|)$ in which no clash-trigger applies and no rule is applicable to a non-blocked node. Hence, $\mathsf{tableau}(T_1)$ returns true. □

The three lemmas just proved imply that we have succeeded in converting the tableau system S into a decision procedure for $\mathcal{P}$.

Theorem 4.1. Suppose the preconditions of Figure 2 are satisfied. Then the algorithm of Figure 2 effectively decides $\mathcal{P}$.

5. A Tableau System for $\mathcal{ALCQI}$

As an example for a more expressive DL that can be treated within our framework, we consider the DL $\mathcal{ALCQI}$, which extends $\mathcal{ALC}$ with qualified number restrictions and inverse roles. Qualified number restrictions ($(\geqslant m\, r.C)$ and $(\leqslant m\, r.C)$) can be used to state constraints on the number of r-successors belonging to a given concept C, and the inverse roles allow us to use both a role r and its inverse r^- when building a complex concept.

Definition 5.1. **($\mathcal{ALCQI}$ Syntax and Semantics)**
Let $\mathsf{N_C}$ and $\mathsf{N_R}$ be pairwise disjoint and countably infinite sets of concept and role names. The set of *$\mathcal{ALCQI}$-roles* is defined as $\mathsf{ROL}_{\mathcal{ALCQI}} := \mathsf{N_R} \cup \{r^- \mid r \in \mathsf{N_R}\}$.

The set of $\mathcal{ALCQI}$-concepts $\mathsf{CON}_{\mathcal{ALCQI}}$ is the smallest set such that

- every concept name is a concept, and
- if C and D are $\mathcal{ALCQI}$-concepts and $r \in \mathsf{ROL}_{\mathcal{ALCQI}}$ is a role, then $\neg C, C \sqcap D, C \sqcup D, (\leqslant m\, r.C)$ and $(\geqslant m\, r.C)$ are also $\mathcal{ALCQI}$-concepts.

TBoxes are defined as in the case of $\mathcal{ALC}$, i.e., they are finite sets of GCIs $C \sqsubseteq D$, where $C, D \in \mathsf{CON}_{\mathcal{ALCQI}}$.

The semantics of $\mathcal{ALCQI}$ is defined as for $\mathcal{ALC}$, where the additional constructors are interpreted as follows:

$$\begin{array}{rcl} (r^-)^\mathcal{I} & := & \{(y,x) \mid (x,y) \in r^\mathcal{I}\}, \\ (\leqslant m\, r.C)^\mathcal{I} & := & \{d \in \Delta^\mathcal{I} \mid \#\{y \mid (d,y) \in r^\mathcal{I} \wedge y \in C^\mathcal{I}\} \leq m\}, \\ (\geqslant m\, r.C)^\mathcal{I} & := & \{d \in \Delta^\mathcal{I} \mid \#\{y \mid (d,y) \in r^\mathcal{I} \wedge y \in C^\mathcal{I}\} \geq m\}, \end{array}$$

where $\#S$ denotes the cardinality of the set S.

Although the constructors $\exists r.C$ and $\forall r.C$ are not explicitly present in $\mathcal{ALCQI}$, they can be simulated by $(\geqslant 1\, r.C)$ and $(\leqslant 0\, r.\neg C)$, respectively. Thus, $\mathcal{ALCQI}$ really extends $\mathcal{ALC}$.

The definition of negation normal form can easily be extended to $\mathcal{ALCQI}$ (see, e.g., [16]). As in the $\mathcal{ALC}^U$ case, we assume that all inputs to the tableau algorithm are in NNF. We use $\mathsf{nnf}(C)$ to denote the negation normal form of a concept C. With $\mathsf{sub}(C, \mathcal{T})$, we denote the set of subconcepts of the concept C and the TBox $\mathcal{T}$. In addition, we define the *closure* of the $\mathcal{ALCQI}$ concept C and the TBox $\mathcal{T}$ as

$$\mathsf{cl}(C, \mathcal{T}) := \mathsf{sub}(C, \mathcal{T}) \cup \{\mathsf{nnf}(\neg D) \mid D \in \mathsf{sub}(C, \mathcal{T})\}.$$

In order to simplify the treatment of inverse roles, we denote *the inverse* of the $\mathcal{ALCQI}$-role r by $\overline{r}$, i.e., $\overline{r} = r^-$ if r is a role name, and $\overline{r} = s$ if $r = s^-$ for a role name s.

Completion trees for $\mathcal{ALCQI}$ look like completion trees for $\mathcal{ALC}$, with the only difference that edges may also be labeled with inverse roles. In fact, to handle a number restriction of the form $(\geqslant 1\, r^-.C)$ (which corresponds to the existential restriction $\exists r^-.C$) in the label of the node v, the tableau algorithm may introduce an r^--successor of v. To simplify the presentation, in the following we say that a node w is an *r-neighbor* of a node v in a completion tree $((V, E, n, \ell), \mu)$, if $(v, w) \in E$ and $\ell(v, w) = r$ or if $(w, v) \in E$ and $\ell(w, v) = \overline{r}$.

The fact that edges can also be labeled by inverse roles complicates the treatment of number restrictions by tableau rules. For a given node, we need to count the number of other nodes it is related to via the role r. Without inverse roles, this is quite easy: we just take the direct successors reached by an edge labeled with r. With inverse roles, we must also count the direct predecessor if the corresponding edge is labeled with $\overline{r}$, and therefore the patterns appearing in the rules and clash triggers for number restrictions are of depth (at most) 2. Since the root node, which does not have a predecessor, requires a special treatment, we reserve a special concept name ROOT for marking the roots of S-trees.

The TBox is stored in the global memory component, similar to the way value restrictions with the universal role are handled by $S_{\mathcal{ALC}^U}$. However, since the TBox is constant, the tableau rules need not modify the global memory component after its initialisation.

Definition 5.2. ($S_{\mathcal{ALCQI}}$)
The tableau system $S_{\mathcal{ALCQI}}$ is defined as follows: $\mathsf{NLE} := \mathsf{CON}_{\mathcal{ALCQI}}$ is the set of all $\mathcal{ALCQI}$-concepts, $\mathsf{GME} := \{C \sqsubseteq D \mid \{C, D\} \subseteq \mathsf{CON}_{\mathcal{ALCQI}}\}$ is the set of all possible $\mathcal{ALCQI}$ GCIs, $\mathsf{EL} := \mathsf{ROL}_{\mathcal{ALCQI}}$ is the set of all $\mathcal{ALCQI}$-roles, the pattern-depth k is set to 2, and the function $\cdot^{S_{\mathcal{ALCQI}}}$ assigns to any input pair $(C, \mathcal{T})$ the following tuple $(\mathsf{nle}_{S_{\mathcal{ALCQI}}}, \mathsf{gme}_{S_{\mathcal{ALCQI}}}, \mathsf{el}_{S_{\mathcal{ALCQI}}}, \mathsf{ini}_{S_{\mathcal{ALCQI}}})$:

$$\begin{aligned}
\mathsf{nle}_{S_{\mathcal{ALCQI}}}(C, \mathcal{T}) &:= \mathsf{cl}(C, \mathcal{T}) \cup \{\mathsf{ROOT}\}, \\
\mathsf{gme}_{S_{\mathcal{ALCQI}}}(C, \mathcal{T}) &:= \mathcal{T}, \\
\mathsf{el}_{S_{\mathcal{ALCQI}}}(C, \mathcal{T}) &:= \{r, r^- \mid r \in \mathsf{N_R} \text{ occurs in } C \text{ or } \mathcal{T}\}, \\
\mathsf{ini}_{S_{\mathcal{ALCQI}}}(C, \mathcal{T}) &:= \{(\{C, \mathsf{ROOT}\}, \mathcal{T})\}.
\end{aligned}$$

The rules and clash-triggers of $S_{\mathcal{ALCQI}}$ are introduced in the two following definitions.

Before formally introducing them, let us discuss the rules of $S_{\mathcal{ALCQI}}$ on an intuitive level. In addition to the rules handling conjunctions and disjunctions, $S_{\mathcal{ALCQI}}$ has one rule that treats TBox axioms and two rules for number restrictions:

$\mathsf{R}\mathcal{T}$ For every GCI $C \sqsubseteq D$, the concept $\mathsf{nnf}(\neg C \sqcup D)$ is added to every node.

$\mathsf{R}\geqslant$ To satisfy an at-least restriction $(\geqslant m\, r.C)$ for a node v, the rule creates or modifies the necessary neighbors one-by-one. In a single step, it (non-deterministically) adds C to v's predecessor if the corresponding edge is labelled with $\overline{r}$, or it adds C to the label of an existing r-successor of v, or it creates a new r-successor of v with label $\{C\}$.

As already noted above, the root of $S_{\mathcal{ALCQI}}$-trees is a special case since it doesn't have any predecessors. For better readability, the handling of the root node is described in a separate rule, $\mathsf{R}\geqslant_{\mathsf{ROOT}}$.

RC If a node v contains an at-most restriction $(\leqslant m\,r.C)$, then this so-called *choose-rule* adds either the concept C or the concept $\mathsf{nnf}(\neg C)$ to the label of all r-neighbours of v. Without this rule, there could be r-neighbors that are neither counted as being in C nor as being in $\neg C$, which is obviously impossible.

Definition 5.3. (The rules of $S_{\mathcal{ALCQI}}$)
Let $P = (t, \mu)$ be a pattern of depth ≤ 2 with $t = (V, E, n, \ell)$, and let v_0 be the root of t. Then $\mathcal{R}(P)$ is the smallest set of finite sets of patterns that contains all the sets of patterns required by the individual rules. The R$\sqcap$ and R$\sqcup$ rules are treated analogously to the $S_{\mathcal{ALC}^U}$ case. The conditions for the remaining rules are as follows:

R$\mathcal{T}$ if $C \sqsubseteq D \in \mu$ and $\mathsf{nnf}(\neg C \sqcup D) \notin n(v_0))$, then $\mathcal{R}(P)$ contains the set $\{((V, E, n', \ell), \mu)\}$ with $n'(v_0) = n(v_0) \cup \{\mathsf{nnf}(\neg C \sqcup D)\}$ and $n'(v) = n(v)$ for all $v \in V \setminus \{v_0\}$.

R$\geqslant$ if the label $n(v_s)$ contains the concept $(\geqslant m\,r.C)$ for a son v_s of v_0, and there are less than m r-neighbours v of v_s with $C \in n(v)$, then $\mathcal{R}(P)$ contains the set $\{P_0, P_1, \ldots, P_t\}$, where $\{u_1, \ldots, u_t\}$ consists of all r-neighbours of v_s with $C \notin n(u_i)$ and

1. $P_0 = ((V_0, E_0, n_0, \ell_0), \mu)$, where $u_0 \notin V$, $V_0 = V \cup \{u_0\}$, $E_0 = E \cup \{(v_s, u_0)\}$, $n_0 = n \cup \{u_0 \mapsto \{C\}\}$, and $\ell_0 = \ell \cup \{(v_s, u_0) \mapsto r\}$,
2. for $1 \leq i \leq t$, $P_i = ((V, E, n_i, \ell), \mu)$, where $n_i(v) = n(v)$ for all $v \in V \setminus \{u_i\}$ and $n_i'(u_i) = n_i(u_i) \cup \{C\}$;

R$\geqslant_{\mathsf{ROOT}}$ if the root label $n(v_0)$ contains the concept $(\geqslant m\,r.C)$ and the ROOT marker, and if there are less than m r-neighbors v of v_0 with $C \in n(v)$, then $\mathcal{R}(P)$ contains the set $\{P_0, P_1, \ldots, P_t\}$, where $\{u_1, \ldots, u_t\}$ consists of all r-neighbours of v_0 with $C \notin n(u_i)$ and

1. $P_0 = ((V_0, E_0, n_0, \ell_0), \mu)$, where $u_0 \notin V$, $V_0 = V \cup \{u_0\}$, $E_0 = E \cup \{(v_0, u_0)\}$, $n_0 = n \cup \{u_0 \mapsto \{C\}\}$, and $\ell_0 = \ell \cup \{(v_0, u_0) \mapsto r\}$,
2. for $1 \leq i \leq t$, $P_i = ((V, E, n_i, \ell), \mu)$, where $n_i(v) = n(v)$ for all $v \in V \setminus \{u_i\}$ and $n_i'(u_i) = n_i(u_i) \cup \{C\}$;

RC if, for some node $v \in V$, the label $n(v)$ contains the concept $(\leqslant m\,r.C)$ and v' is an r-neighbour of v with $n(v') \cap \{C, \mathsf{nnf}(\neg C)\} = \emptyset$, then $\mathcal{R}(P)$ contains the set $\{((V, E, n', \ell), \mu), ((V, E, n'', \ell), \mu)\}$, where $n'(v') = n(v') \cup \{C\}$, $n''(v') = n(v') \cup \{\mathsf{nnf}(\neg C)\}$, and $n'(v^*) = n''(v^*) = n(v^*)$ for all $v^* \in V \setminus \{v'\}$.

Let us briefly discuss the doubling of the R$\geqslant$ rule, which is due to the presence of inverse roles. To deal with a concept $(\leqslant m\,r.C) \in n(v)$, the R$\geqslant$ rule has to take into account potential $\overline{r}$-predecessors of v satisfying C. This is achieved by defining the R$\geqslant$ rule such that, in patterns, v is not the root node, but rather a son of the root node. For this reason, the R$\geqslant$ rule can never be applied to concepts $(\leqslant m\,r.C)$ in the label of the root node since we cannot match the root node with a node on level 1 in a pattern. This necessitates the additional R$\geqslant_{\mathsf{ROOT}}$ rule.

Definition 5.4. ($S_{\mathcal{ALCQI}}$ Clash triggers)
The set of clash-triggers $\mathcal{C}$ contains all patterns $((V, E, n, \ell), \mu)$ with depth ≤ 2 such that, for some $v \in V$, we have one of the following:

- $\{A, \neg A\} \in n(v)$ for some concept name A;
- $(\leqslant m\, r.C) \in n(v)$ and v has more than m r-neighbors containing C.

Admissibility, ExpTime-admissibility, and recursive admissibility of $S_{\mathcal{ALCQI}}$ can be shown as for $S_{\mathcal{ALC}^U}$. The proof of soundness and completeness is similar to known soundness and completeness proofs for tableau algorithms for DLs containing qualified number restrictions and inverse roles (see, e.g., [21]). In order to have p-completeness for an appropriate polynomial p, we must assume that numbers in number restrictions $(\geqslant m\, r.C)$ and $(\leqslant m\, r.C)$ are given in unary coding, i.e., that the number m contributes linearly rather than logarithmically to the size of the input. As an immediate consequence of Theorem 3.1, we obtain the following upper-bound for the satisfiability problem w.r.t. TBoxes in $\mathcal{ALCQI}$.

Corollary 5.1. $\mathcal{ALCQI}$-concept satisfiability w.r.t. TBoxes is in ExpTime if numbers are coded in unary.

6. Related Work

In this paper, we have proposed an abstract framework for the development of tableau algorithms. The most prominent feature of this framework is that it allows to prove tight ExpTime-complexity bounds, and to obtain (tableau) algorithms that can serve as the basis for efficient implementations—without investing double work. We only know of one other attempt to capture tableau algorithms in an abstract framework, namely [22]. The main difference is that the framework in [22] is much less fine-grained than ours, and indeed also captures quite different types of algorithms such as resolution-based ones. For this reason, it does not allow for fain-grained complexity analyses such as the one performed in Section 3.

Closely related to our work is the attempt to implement automata-based algorithms in an efficient way. This is the case since, as we have seen in Section 3, automata are well-suited for proving ExpTime upper bounds: if they can be efficiently implemented, there is again no reason to do double work for obtaining theoretical complexity results and practical implementations. Let us discuss two recent approaches:

(1) In [6], Baader and Tobies show that the so-called inverse method for deciding satisfiability of the modal logic K can be viewed as an implementation of the automata-theoretic approach. Since K is known to be a notational variant of the description logic $\mathcal{ALC}$ [32], this observation is of direct relevance for the area of description logic as well. It is particularly interesting since there exist quite efficient implementations of the inverse method [37].

(2) Pan et al. propose a BDD-based decision procedure for the modal logic K, and show that it can be implemented rather efficiently [27, 28]. They also note that their method is inspired by and closely related to the automata approach.

Although the developments in the implementation of automata-based algorithms are promising, we believe that tableau-based implementations of ExpTime-complete logics will continue to play an important role in DL and related areas.

Finally, we should like to comment on the differences between the present version of this article and its conference version [3]. To achieve more generality, we have extended our initial formlism as proposed in [3] in two directions: first, we have introduced the global memory component that can be used to formulate rules of a global flavour. Such rules are useful for dealing with the universal modality, with TBoxes, and with nominals (concept names that have to be interpreted in singleton sets). Second, we have generalized our framework to patterns of arbitrary depth—in [3], only patterns of depth at most 1 are allowed. This generalization helps to capture some tableau algorithms, such as the one for $\mathcal{ALCQI}$ in Section 5, in a much more natural way.

References

[1] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[2] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H.-J. Profitlich. An empirical analysis of optimization techniques for terminological representation systems or: Making KRIS get a move on. *Applied Artificial Intelligence. Special Issue on Knowledge Base Management*, 4:109–132, 1994.

[3] F. Baader, J. Hladik, C. Lutz, and F. Wolter. From tableaux to automata for description logics. In M. Vardi and A. Voronkov, editors, *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2003)*, volume 2850 of *Lecture Notes in Computer Science*, pages 1–32, Almaty, Kazakhstan, 2003. Springer-Verlag.

[4] F. Baader and W. Nutt. Basic description logics. In *[1]*, pages 43–95. 2003.

[5] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.

[6] F. Baader and S. Tobies. The inverse method implements the automata approach for modal satisfiability. In *Proc. of the Int. Joint Conf. on Automated Reasoning IJCAR'01*, volume 2083 of *LNAI*, pages 92–106. Springer-Verlag, 2001.

[7] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*, volume 53 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2001.

[8] R. J. Brachman and H. J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann, Los Altos, 1985.

[9] D. Calvanese and G. DeGiacomo. Expressive description logics. In *[1]*, pages 178–218. 2003.

[10] G. De Giacomo and M. Lenzerini. TBox and ABox reasoning in expressive description logics. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Proc. of the 5th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'96)*, pages 316–327. Morgan Kaufmann, Los Altos, 1996.

[11] F. Donini. Complexity of reasoning. In *[1]*, pages 96–136. 2003.

[12] F. Donini and F. Massacci. EXPTIME tableaux for $\mathcal{ALC}$. *Acta Informatica*, 124(1):87–138, 2000.

[13] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proc. of the 2nd Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'91)*, pages 151–162. Morgan Kaufmann, Los Altos, 1991.

[14] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. Tractable concept languages. In *Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI'91)*, pages 458–463, Sydney (Australia), 1991.

[15] V. Haarslev and R. Möller. RACER system description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2001)*, 2001.

[16] B. Hollunder and F. Baader. Qualifying number restrictions in concept languages. In *Proc. of the 2nd Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'91)*, pages 335–346, 1991.

[17] I. Horrocks. Using an expressive description logic: FaCT or fiction? In *Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.

[18] I. Horrocks. Implementation and optimization techniques. In *[1]*, pages 306–346. 2003.

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

[20] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for expressive description logics. In H. Ganzinger, D. McAllester, and A. Voronkov, editors, *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes in Artificial Intelligence, pages 161–180. Springer-Verlag, 1999.

[21] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *J. of the Interest Group in Pure and Applied Logic*, 8(3):239–264, 2000.

[22] R. Kontchakov, C. Lutz, F. Wolter, and M. Zakharyaschev. Temporal tableaux. *Studia Logica*, 2003. To appear.

[23] R. MacGregor. The evolving technology of classification-based knowledge representation systems. In J. F. Sowa, editor, *Principles of Semantic Networks*, pages 385–400. Morgan Kaufmann, Los Altos, 1991.

[24] E. Mays, R. Dionne, and R. Weida. K-REP system overview. *SIGART Bull.*, 2(3), 1991.

[25] M. Minsky. A framework for representing knowledge. In J. Haugeland, editor, *Mind Design*. The MIT Press, 1981. A longer version appeared in *The Psychology of Computer Vision* (1975). Republished in [8].

[26] R. Möller and V. Haarslev. Description logic systems. In *[1]*, pages 282–305. 2003.

[27] G. Pan, U. Sattler, and M. Y. Vardi. BDD-based decision procedures for K. In A. Voronkov, editor, *Proc. of the 18th Int. Conf. on Automated Deduction (CADE 2002)*, volume 2392 of *LNAI*, pages 16–30. Springer-Verlag, 2002.

[28] G. Pan and M. Y. Vardi. Optimizing a bdd-based modal solver. In F. Baader, editor, *Proc. of the 19th Int. Conf. on Automated Deduction (CADE 2003)*, volume 2741 of *LNAI*, pages 75–89. Springer-Verlag, 2003.

[29] P. F. Patel-Schneider, D. L. McGuiness, R. J. Brachman, L. Alperin Resnick, and A. Borgida. The CLASSIC knowledge representation system: Guiding principles and implementation rational. *SIGART Bull.*, 2(3):108–113, 1991.

[30] C. Peltason. The BACK system — an overview. *SIGART Bull.*, 2(3):114–119, 1991.

[31] M. R. Quillian. Word concepts: A theory and simulation of some basic capabilities. *Behavioral Science*, 12:410–430, 1967. Republished in [8].

[32] K. Schild. A correspondence theory for terminological logics: Preliminary report. In *Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI'91)*, pages 466–471, 1991.

[33] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[34] E. Spaan. *Complexity of Modal Logics*. PhD thesis, Department of Mathematics and Computer Science, University of Amsterdam, 1993.

[35] S. Tobies. A PSPACE algorithm for graded modal logic. In H. Ganzinger, editor, *Proc. of the 16th Int. Conf. on Automated Deduction (CADE'99)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 52–66. Springer-Verlag, 1999.

[36] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. of Computer and System Sciences*, 32:183–221, 1986. A preliminary version appeared in *Proc. of the 16th ACM SIGACT Symp. on Theory of Computing (STOC'84)*.

[37] A. Voronkov. How to optimize proof-search in modal logics: new methods of proving redundancy criteria for sequent calculi. *ACM Trans. on Comp. Logic*, 2(2):182–215, 2001.

[38] W. A. Woods and J. G. Schmolze. The KL-ONE family. In F. W. Lehmann, editor, *Semantic Networks in Artificial Intelligence*, pages 133–178. Pergamon Press, 1992. Published as a special issue of *Computers & Mathematics with Applications*, Volume 23, Number 2–9.