An Automata View to Goal-directed Methods

Lisa Hutschenreiter[1]* and Rafael Peñaloza[2]

[1] Technische Universität Dresden, Germany (`lisa.hutschenreiter@tu-dresden.de`)
[2] Free University of Bozen-Bolzano, Italy (`rafael.penaloza@unibz.it`)

Abstract. Consequence-based and automata-based algorithms encompass two families of approaches that have been thoroughly studied as reasoning methods for many logical formalisms. While automata are useful for finding tight complexity bounds, consequence-based algorithms are typically simpler to describe, implement, and optimize. In this paper, we show that consequence-based reasoning can be reduced to the emptiness test of an appropriately built automaton. Thanks to this reduction, one can focus on developing efficient consequence-based algorithms, obtaining complexity bounds and other benefits of automata methods for free.

1 Introduction

In logic-based knowledge representation, the knowledge from a domain is encoded using a finite set of axioms, expressed and interpreted in a suitable logic, that restrict the ways in which the domain symbols can be used. To keep the representation task feasible, only the most relevant portions of the knowledge are explicitly represented, while other consequences of these axioms are only implicitly available. The task of making this implicit knowledge explicit is usually known as *reasoning.*

For practical applications, it is fundamental to have effective reasoning methods. Ideally, these should be optimal w.r.t. the computational complexity of the reasoning problem, easy to implement, and behave well in practice. Keeping these desiderata in mind, many reasoning algorithms have been developed over the years. These can be broadly classified into two approaches: the automata-based, and the tableaux-based approaches. In a nutshell, the automata-based approach constructs an automaton that accepts all relevant models of a knowledge base. An emptiness test can then be used to decide whether a given (implicit) consequence can be derived from it. On the other hand, tableaux methods apply rules to extend the explicit knowledge until the consequence is derived. They can thus be thought of as "goal-directed" approaches. Consequence-based algorithms fall into this latter category, where rule applications have a limited effect.

Automata-based methods provide formal tools for understanding the theoretical properties and limitations of reasoning in the underlying logical formalism. They have been successfully used to prove tight (worst-case) complexity

* Supported by the DFG Graduiertenkolleg 1763 (QuantLA).

bounds [2]. They also provide an elegant solution for dealing with supplemental reasoning problems that can be seen as weighted model counting problems [4,20]. However, with a few notable exceptions [11], automata-based methods are not suitable for efficient implementations due to the match between their best-case and their worst-case behaviour.

Due to their goal-directed behaviour, tableaux-based methods are the basis of some of the most efficient implementations of reasoning algorithms available. However, extending these methods to handle supplemental reasoning problems is far from obvious, and might even lead to non-terminating procedures [5]. In fact, the known method for transforming a consequence-based algorithm into a supplemental reasoning procedure requires the application of an NP-hard propositional entailment test on *every* step of the execution, potentially damaging the overall complexity of the methods [6]. This is especially negative if one takes into account that the supplemental extensions of automata-based methods typically preserve the original complexity.

In this paper, we combine the benefits of consequence-based and automata-based methods to overcome their respective drawbacks. More precisely, we show that every consequence-based algorithm can be effectively transformed into an automaton deciding the same reasoning problem. Thus, one can focus on designing and implementing an efficient consequence-based algorithm and take advantage of this transformation to obtain tight complexity bounds and extensions to supplemental reasoning provided by the automata-based view.

Throughout this paper, we consider a general notion of consequence-based algorithm, which makes our results applicable to a wide range of settings. To improve readability, we use as a running example a known method for deciding subsumption in the description logic $\mathcal{EL}$.

2 Consequence-based Algorithms

We consider a general notion of consequence-based algorithms as automated reasoning methods. The family of these methods, also called *ground tableaux* in the literature [5], encompasses many well-known algorithms, such as DPLL [8], congruence closure [19], and the prominent methods for reasoning in $\mathcal{EL}$ [1] and other description logics [13]. As a running example for all our notions, we will adapt the consequence-based algorithm for deciding subsumption first presented in $\mathcal{EL}$ from [14]. Thus we briefly recall this logic and its reasoning problem.

$\mathcal{EL}$ is a lightweight description logic whose main building blocks are *concepts* and *roles*. Given two disjoint infinite sets N_C and N_R of *concept names* and *role names*, respectively, $\mathcal{EL}$ *concepts* are constructed using the grammar rule $C ::= A \mid \top \mid C \sqcap C \mid \exists r.C$, where $A \in N_C$ and $r \in N_R$. An $\mathcal{EL}$ *TBox* is a finite set of *general concept inclusions (GCIs)* of the form $C \sqsubseteq D$, where C, D are $\mathcal{EL}$ concepts. The semantics of this logic is based on interpretations, which are pairs of the form $\mathcal{J} = (\Lambda^{\mathcal{J}}, \cdot^{\mathcal{J}})$, with $\Lambda^{\mathcal{J}}$ a non-empty set called the *domain* and $\cdot^{\mathcal{J}}$ the *interpretation function* that maps every concept name A to a subset $A^{\mathcal{J}} \subseteq \Lambda^{\mathcal{J}}$ and every role name r to a binary relation $r^{\mathcal{J}} \subseteq \Lambda^{\mathcal{J}} \times \Lambda^{\mathcal{J}}$. This function is extended to $\mathcal{EL}$ concepts by defining $\top^{\mathcal{J}} := \Lambda^{\mathcal{J}}$, $(C \sqcap D)^{\mathcal{J}} := C^{\mathcal{J}} \cap D^{\mathcal{J}}$, and $(\exists r.C)^{\mathcal{J}} := \{\lambda \in \Lambda^{\mathcal{J}} \mid \exists \mu \in C^{\mathcal{J}}.(\lambda, \mu) \in r^{\mathcal{J}}\}$. The interpretation $\mathcal{J}$ *satisfies* the GCI $C \sqsubseteq D$ iff $C^{\mathcal{J}} \subseteq D^{\mathcal{J}}$. It is a *model* of the TBox $\mathcal{T}$ if it satisfies all GCIs in $\mathcal{T}$. The concept C is *subsumed* by the concept D w.r.t. the TBox $\mathcal{T}$ if every model of $\mathcal{T}$ also satisfies the GCI $C \sqsubseteq D$.

We now define the notion of consequence-based algorithm. To remain as general as possible, for the rest of this paper we consider that we have two arbitrary (but fixed) sets $\mathfrak{I}$ of *inputs* and $\mathfrak{T}$ of *axioms*. We denote by $\mathscr{P}_{\mathsf{fin}}(\mathfrak{T})$ the set of all finite subsets of $\mathfrak{T}$. The different inputs in $\mathfrak{I}$ will be usually denoted by calligraphic letters like $\mathcal{I}$, and the axioms with lower-case letters; e.g., s.[3] We start by defining the consequences that will be decided by our algorithms.

Definition 1 (property). *A* consequence property *(or* property *for short) is a binary relation $\mathcal{P} \subseteq \mathfrak{I} \times \mathscr{P}_{\mathsf{fin}}(\mathfrak{T})$ such that if $(\mathcal{I}, \mathcal{T}) \in \mathcal{P}$, then $(\mathcal{I}, \mathcal{T}') \in \mathcal{P}$ for all $\mathcal{T} \subseteq \mathcal{T}' \in \mathscr{P}_{\mathsf{fin}}(\mathfrak{T})$.*

In other words, we are interested in deciding whether a given input $\mathcal{I}$ is entailed by a finite set of axioms $\mathcal{T}$ through a monotone relation. We use monotonicity in the standard logical sense in which new information can only generate more consequences, but never negate previously entailed ones. Throughout this paper we call pairs of the form $(\mathcal{I}, \mathcal{T})$ *axiomatized inputs*.

Consider for example the subsumption problem in $\mathcal{EL}$. In this case, the set $\mathfrak{T}$ of axioms consists of all possible GCIs $C \sqsubseteq D$. $\mathfrak{I}$ is the class of all possible subsumption relations, which will be denoted by $C \sqsubseteq^? D$ to distinguish them from the axioms in the TBox. The property $\mathcal{P}$ refers to the entailment of subsumption relations; that is, $(C \sqsubseteq^? D, \mathcal{T}) \in \mathcal{P}$ iff C is subsumed by D w.r.t. $\mathcal{T}$.

Depending on the specific shape of the property $\mathcal{P}$, there exist many different approaches for deciding whether a given axiomatized input $(\mathcal{I}, \mathcal{T})$ belongs to $\mathcal{P}$ or not. We focus on a class of decision methods which we call consequence-based algorithms.

Definition 2 (consequence-based algorithm). *Let $\mathfrak{I}$ be a set of inputs, and $\mathfrak{T}$ a set of axioms. A* consequence-based algorithm *for $\mathfrak{I}$ and $\mathfrak{T}$ is a tuple of the form $S = (\Sigma, \cdot^S, \mathcal{R}, \mathcal{C})$ where*

- *Σ is a set called the* signature*;*
- *$\cdot^S$ is the* initialization function *that maps every $\mathcal{I} \in \mathfrak{I}$ and every $t \in \mathfrak{T}$ to a finite subset of Σ;*
- *$\mathcal{R}$ is a set of* rules *of the form $(B_0, \mathcal{S}) \to B$, where B_0 and B are finite subsets of Σ and $\mathcal{S} \in \mathscr{P}_{\mathsf{fin}}(\mathfrak{T})$; and*
- *$\mathcal{C}$ is a set of finite subsets of Σ, which are called* clashes.

The idea of a consequence-based algorithm is to decide a property by making explicit all the consequences of the given axiomatized input. The explicit knowledge derived during the execution of the algorithm S is preserved in S-states.

[3] In the literature (e.g. [5]) the sets of axioms allowed are sometimes restricted to satisfy an *admissibility* criterion. As this criterion is irrelevant for our methods, we leave it out for the sake of simplicity.

Formally, given a consequence-based algorithm $S = (\Sigma, \cdot^S, \mathcal{R}, \mathcal{C})$, an *$S$-state* is a pair $\mathfrak{S} = (A, \mathcal{T})$ where A is a finite subset of Σ and $\mathcal{T} \in \mathscr{P}_{\mathsf{fin}}(\mathfrak{T})$. We call the elements of A in such an S-state *assertions*.

Given the axiomatized input $(\mathcal{I}, \mathcal{T})$, the decision procedure begins with the *initial S-state* $(\mathcal{I}, \mathcal{T})^S$ defined by $(\mathcal{I}, \mathcal{T})^S := (\mathcal{I}^S \cup \bigcup_{t \in \mathcal{T}} t^S, \mathcal{T})$. That is, the initial S-state contains all the assertions obtained by applying the initialization function to the input $\mathcal{I}$ and the axioms in $\mathcal{T}$. Notice that the second component of this S-state is the same set of axioms $\mathcal{T}$. As we will see, the algorithm never modifies this set. However, preserving the information of the set of axioms used will be helpful in the following sections. The first component of the S-state $\mathfrak{S}$ is iteratively extended through rule applications, which depend exclusively on the assertions and axioms appearing in $\mathfrak{S}$.

Definition 3 (rule application). *Let $\mathfrak{S} = (A, \mathcal{T})$ be an S-state. The rule $\mathsf{R} : (B_0, \mathcal{S}) \to B$ is* applicable *to $\mathfrak{S}$ iff (i) $\mathcal{S} \subseteq \mathcal{T}$, (ii) $B_0 \subseteq A$; and (iii) $B \not\subseteq A$. The* application of R to $\mathfrak{S}$ *yields the new S-state $\mathfrak{S}' := (A \cup B, \mathcal{T})$. In this case, we write $\mathfrak{S} \to_{\mathsf{R}} \mathfrak{S}'$, or $\mathfrak{S} \to_S \mathfrak{S}'$ if the specific rule applied is irrelevant.*

As usual, the reflexive and transitive closure of $\to_S$ is denoted by $\xrightarrow{*}_S$.

Starting from the initial state, consequence-based algorithms apply rules in a *not-care non-deterministic* manner until a *saturated* state is reached; that is, a state for which no rule is applicable. In this case, the not-care non-determinism means that whenever more than one rule can be applied to a given state, any one of them can be chosen. Finally, the set of clashes $\mathcal{C}$ is used to decide the property once a saturated S-state is reached: the axiomatized input is *accepted* (i.e., it belongs to the property $\mathcal{P}$) if and only if it contains a clash.

A consequence-based algorithm decides a property $\mathcal{P}$ if it always terminates and the resulting saturated S-state contains a clash whenever the axiomatized input belongs to $\mathcal{P}$. This notion is formalized next.

Definition 4 (correct). *The consequence-based algorithm $S = (\Sigma, \cdot^S, \mathcal{R}, \mathcal{C})$ is* correct *for the property $\mathcal{P}$ iff for every axiomatized input $\Gamma = (\mathcal{I}, \mathcal{T})$, the following two conditions hold:*

1. *S terminates on Γ; that is, there exists no infinite chain of rule applications $\mathfrak{S}_0 \to_S \mathfrak{S}_1 \to_S \cdots$ starting with $\mathfrak{S}_0 = \Gamma$; and*
2. *for every chain of rule applications $\mathfrak{S}_0 \xrightarrow{*}_S \mathfrak{S}_n$ with $\mathfrak{S}_0 = \Gamma$ and $\mathfrak{S}_n$ a saturated S-state, $\Gamma \in \mathcal{P}$ iff $\mathfrak{S}_n$ contains a clash.*

Example 5. A consequence-based algorithm for deciding subsumption in $\mathcal{EL}$ is given by $S_{\mathcal{EL}} = (\Sigma, \cdot^{S_{\mathcal{EL}}}, \mathcal{R}, \mathcal{C})$, where

- $\Sigma := \{C \sqsubseteq D, C \sqsubseteq^? D \mid C, D$ are $\mathcal{EL}$ concepts$\}$;
- for every GCI in $\mathfrak{T}$, $(C \sqsubseteq D)^{S_{\mathcal{EL}}} := \{C \sqsubseteq C, C \sqsubseteq \top\}$;
- for every $C \sqsubseteq^? D \in \mathfrak{I}$, $(C \sqsubseteq^? D)^{S_{\mathcal{EL}}} := \{C \sqsubseteq^? D\}$;
- the set of rules $\mathcal{R}$ is depicted in Figure 1; and
- $\mathcal{C} := \{\{C \sqsubseteq D, C \sqsubseteq^? D\}\}$

	B_0	$\mathcal{S}$	$\to$	B
$\mathsf{R}_\sqcap^-$:	$\{C \sqsubseteq D_1 \sqcap D_2\}$	$\emptyset$	$\to$	$\{C \sqsubseteq D_1, C \sqsubseteq D_2\}$
$\mathsf{R}_\sqcap^+$:	$\{C \sqsubseteq D_1, C \sqsubseteq D_2\}$	$\emptyset$	$\to$	$\{C \sqsubseteq D_1 \sqcap D_2\}$
$\mathsf{R}_\exists$:	$\{E \sqsubseteq \exists r.C, C \sqsubseteq D\}$	$\emptyset$	$\to$	$\{E \sqsubseteq \exists r.D\}$
$\mathsf{R}_\sqsubseteq$:	$\{C \sqsubseteq D\}$	$\{D \sqsubseteq E\}$	$\to$	$\{C \sqsubseteq E\}$

Fig. 1. Rules for the consequence-based algorithm $S_{\mathcal{EL}}$

Intuitively, the algorithm makes all the relevant subsumption relations that follow from the TBox $\mathcal{T}$ explicit by applying the rules. The input subsumption relation, whose entailment is being decided, is kept in the set of assertions to be able to produce a clash when it is derived through the application of rules. The correctness of this algorithm has been shown in [14].

Notice that the second condition from Definition 4 requires that the algorithm yields the same answer regardless of the order in which the rules were applied. This corresponds to the not-care non-determinism mentioned above. As a consequence, any order chosen suffices for deciding whether a property holds or not; i.e., there is no need to backtrack if a saturated S-state without a clash is found. In fact, for consequence-based algorithms, the second condition is equivalent to requiring that there exists one sequence of rule applications that leads to a saturated state with a clash. More precisely, for every axiomatized input Γ and saturated S-states $\mathfrak{S}, \mathfrak{S}'$, if $\Gamma^S \xrightarrow{*}_S \mathfrak{S}$ and $\Gamma^S \xrightarrow{*}_S \mathfrak{S}'$, then $\mathfrak{S}$ and $\mathfrak{S}'$ must be equivalent. This can be easily seen from the fact that whenever a rule $(B_0, \mathcal{S}) \to B$ is applicable in an S-state, it remains applicable to any successive S-state until all the assertions in B have been introduced, which has the same effect as having applied the rule before-hand. Once again, this in particular means that the order in which the rules are applied is irrelevant for deriving a clash or not.

For supplemental reasoning, it is often important to know all the possible ways in which clashes can be generated by rule applications (see Section 5). In order to keep track of the different orders that have already been followed, extensions of consequence-based algorithms developed for dealing with these problems require the use of an NP oracle [5].

3 Tree Automata

We briefly recall the basic notions of tree automata. These automata receive finite trees of a fixed arity k with identified successors as inputs. For the rest of this paper, given a positive integer k, we will denote by K the set $\{1, \ldots, k\}$. As is usual, we identify the nodes in a k-ary tree by words in K^*: the root node is identified by the empty word ε, and the i-th successor of the node $u \in \mathsf{K}^*$ is identified by the word ui for all $i \in \mathsf{K}$.

Definition 6 (finite tree). *A* finite k-ary tree *is a finite subset $t \subseteq \mathsf{K}^*$ such that for every node $ui \in t$ it holds that (i) $u \in t$, and (ii) $uj \in t$ for all $j, 1 \leq j \leq i$.*

Intuitively, Condition (i) states that the parent of every node belongs to the tree; this in particular means that a non-empty tree will always contain the root node ε. The second condition ensures that all the successors of a node are adequately identified by the smallest possible natural numbers without gaps. We say that a tree is *full* if every node is either a *leaf* (i.e., has no successors), or has exactly k successors.

A *labelled tree* is a tree $t \subseteq \mathsf{K}^*$ extended with a labelling function lab. If the labels belong to a given set Q, we will often denote labelled trees as a function $\mathsf{lab} : t \to Q$. For this paper, we focus on tree automata that receive full finite unlabelled trees of a fixed arity k as inputs.

Definition 7 (finite tree automaton). *A* finite tree automaton *of arity k is a tuple $\mathcal{A} = (Q, \Delta, I, F)$ where*

- *Q is a finite set, whose elements are called* states*;*
- *$\Delta \subseteq Q^{k+1}$ is the* transition relation*;*
- *$I \subseteq Q$ is the set of* initial states*; and*
- *$F \subseteq Q$ is the set of* final states*.*

A run *of this automaton on the full unlabelled finite tree t is a labelled tree of the form $\mathsf{lab} : t \to Q$ such that for every non-leaf node $u \in t$, it holds that $(\mathsf{lab}(u), \mathsf{lab}(u1), \ldots, \mathsf{lab}(uk)) \in \Delta$. This run is* successful *iff for every leaf node $u \in t$, $\mathsf{lab}(u) \in F$.*

The *emptiness problem* for finite tree automata refers to the problem of deciding whether there exists a finite unlabelled tree t and a successful run on t with $\mathsf{lab}(\varepsilon) \in I$. In this case, we say that the automaton is *non-empty*. We often refer to successful runs with $\mathsf{lab}(\varepsilon) \in I$ as *accepting*. It is well known that the emptiness problem can be solved in polynomial time on the number of states. This general bound can be in fact improved to *linear* time in the number of states through a bottom-up approach that identifies the states that may appear in a successful run—we call these *good* states. All final states are clearly good. Further good states can be iteratively found by adding all states $q \in Q$ such that there is a transition of the form $(q, q_1, \ldots, q_k) \in \Delta$ where every $q_i, 1 \leq i \leq k$ is a good state. This iteration detects an initial state that is good if and only if the automaton is non-empty.

4 From Consequence-based to Automata

We now show how to translate any given consequence-based algorithm S into a finite tree automaton of an appropriate arity in such a manner that the emptiness test of the latter can be used to verify whether an axiomatized input Γ belongs to the property decided by the former or not.

For this section we consider an arbitrary but fixed consequence-based algorithm $S = (\Sigma, \cdot^S, \mathcal{R}, \mathcal{C})$. We fix the constant $k := \max\{|B_0| \mid (B_0, \mathcal{S}) \to B \in \mathcal{R}\}$. In the following, whenever we refer to a tree, we implicitly assume that it is a finite k-ary tree, where k is defined as before.

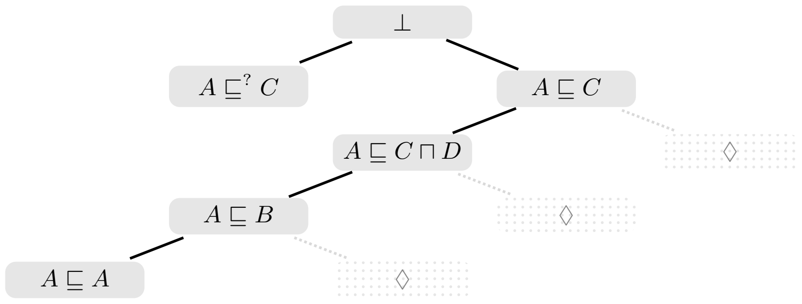

Fig. 2. A derivation tree for $(A \sqsubseteq^? C, \{A \sqsubseteq B, B \sqsubseteq C \sqcap D\})$ and its binary padding.

We can assume w.l.o.g. that all the rules in $\mathcal{R}$ are of the form $(B_0, \mathcal{S}) \to \{\sigma\}$ with $\sigma \in \Sigma$; that is, a rule application adds only one alphabet symbol as a consequence. To see this, notice that the rule $(B_0, \mathcal{S}) \to B$ can be equivalently replaced in $\mathcal{R}$ by the set of rules $\{(B_0, \mathcal{S}) \to \{\sigma\} \mid \sigma \in B\}$. Similarly, we assume that the set of clashes $\mathcal{C}$ contains only the singleton $\{\bot\}$. Otherwise, we can extend the set of rules $\mathcal{R}$ to include $(C, \emptyset) \to \{\bot\}$ for every $C \in \mathcal{C}$. This extension does not change the behaviour of the algorithm S, as it will still detect the presence of a clash by verifying whether the new symbol $\bot$ was derived. For example, we can change the rule $\mathsf{R}_\sqcap^-$ of the consequence-based algorithm $S_{\mathcal{EL}}$ (see Example 5) into two rules that derive $C \sqsubseteq D_1$ and $C \sqsubseteq D_2$, respectively, and add a new rule $\mathsf{R}_c : (\{C \sqsubseteq D, C \sqsubseteq^? D\}, \emptyset) \to \{\bot\}$ to satisfy the aforementioned assumptions without compromising its correctness for deciding subsumption relations.

Under these assumptions, we can view the possible derivations of the clash $\bot$ as labelled trees, where each node is labelled with an assertion from Σ, the root node is labelled by $\bot$, and the children of each node represent the set of assertions needed to apply a rule that generates them.

Definition 8 (derivation tree). *A* derivation tree *for the axiomatized input $\Gamma = (\mathcal{I}, \mathcal{T})$ w.r.t. the consequence-based algorithm S is a labelled finite k-ary tree $\mathsf{lab} : t \to \Sigma$ such that the following conditions hold:*

1. $\mathsf{lab}(\varepsilon) = \bot$*;*
2. *for every leaf node $u \in t$, $\mathsf{lab}(u) \in \Gamma^S$; and*
3. *for every non-leaf node $u \in t$ with i successors, there exists a rule of the form $(\{\mathsf{lab}(u1), \ldots, \mathsf{lab}(ui)\}, \mathcal{S}) \to \{\mathsf{lab}(u)\}$ in $\mathcal{R}$ such that $\mathcal{S} \subseteq \mathcal{T}$.*

Example 9. Consider again the algorithm $S_{\mathcal{EL}}$ for deciding subsumption in $\mathcal{EL}$ from Example 5. Since the rules in $S_{\mathcal{EL}}$ have at most two assertions as prerequisite for application, derivation trees for this algorithm will be binary. If we want to decide whether A is subsumed by C w.r.t. the TBox $\mathcal{T} = \{A \sqsubseteq B, B \sqsubseteq C \sqcap D\}$, we will call $S_{\mathcal{EL}}$ with the axiomatized input $\Gamma = (A \sqsubseteq^? C, \mathcal{T})$. A derivation tree for this input w.r.t. $S_{\mathcal{EL}}$ is depicted in Figure 2 (ignore the nodes labelled with $\Diamond$ for the moment). Notice that there are only two leaf nodes, labelled with $A \sqsubseteq^? C$ and $A \sqsubseteq A$. Both of them belong to $\Gamma^{S_{\mathcal{EL}}}$.

As shown by the following theorem, to decide whether the axiomatized input Γ belongs to the property $\mathcal{P}$, it suffices to check for the existence of a derivation tree.

Theorem 10. *Let S be a consequence-based algorithm that is correct for the property $\mathcal{P}$, and $\Gamma = (\mathcal{I}, \mathcal{T})$ an axiomatized input. Then $\Gamma \in \mathcal{P}$ iff there exists a derivation tree for Γ w.r.t. S.*

Proof. Let $\mathfrak{S} = (A, \mathcal{T})$ be a saturated S-state such that $\Gamma^S \xrightarrow{*}_S \mathfrak{S}$. It suffices to show that $\bot \in A$ iff there exists a derivation tree for Γ w.r.t. S.
(if) Assume first that $\mathsf{lab} : t \to \Sigma$ is a derivation tree. We show by induction on the tree structure that $\mathsf{lab}(u) \in A$ holds for all $u \in t$. First, for all leaf nodes u, we have by definition that $\mathsf{lab}(u) \in \Gamma^S \subseteq A$. For the induction step consider a node u such that all its successors satisfy the property. By Condition 3 of Definition 8, there exists a rule $(\{\mathsf{lab}(u1), \ldots, \mathsf{lab}(ui)\}, \mathcal{S}) \to \{\mathsf{lab}(u)\}$ in $\mathcal{R}$ with $\mathcal{S} \subseteq \mathcal{T}$. Since this rule is not applicable to $\mathfrak{S}$, it must be the case that $\mathsf{lab}(u) \in A$, which finishes the induction proof. In particular, this means that $\mathsf{lab}(\varepsilon) = \bot \in A$.
(only if) Assume now that $\bot \in A$. We construct a derivation tree $\mathsf{lab} : t \to \Sigma$ recursively as follows. First set $\mathsf{lab}(\varepsilon) := \bot$. For each node $u \in t$ do the following. If $\mathsf{lab}(u) \in \Gamma^S$, then u is a leaf node. Otherwise, since $\mathsf{lab}(u) \in A$, there exists a rule $(B_0, \mathfrak{S}) \to \{\mathsf{lab}(u)\} \in \mathcal{R}$ with $B_0 \subseteq A$ and $\mathcal{S} \subseteq \mathcal{T}$ such that for all predecessors v of u it holds that $\mathsf{lab}(v) \notin B_0$. Let $B_0 = \{b_1, \ldots, b_n\}$. Then we add n new nodes $u_1, \ldots, u_n$ to t with $\mathsf{lab}(ui) := b_i$ for all $i, 1 \leq i \leq n$. It is easy to see that this construction yields a derivation tree for Γ. $\square$

Based on this result, we will construct an automaton that produces derivation trees as its accepting runs. If such a run exists—i.e., if the automaton is not empty—then the axiomatized input belongs to the property.

Recall that we are interested only in consequence-based algorithms that can correctly decide a given property $\mathcal{P}$. In particular, this means that the algorithm S must terminate on all axiomatized inputs Γ. It is thus reasonable to assume that for every such axiomatized input $\Gamma = (\mathcal{I}, \mathcal{T})$, there exists a finite subset $\Sigma_\Gamma \subseteq \Sigma$ of signature symbols that contains Γ^S and $\bot$, and is closed under rule applications w.r.t. the axioms in $\mathcal{T}$. In other words, Σ_Γ is a known over-approximation of all the symbols that will be used during the execution of the consequence-based algorithm. Such a set may e.g., be known from the proof of termination. For example, many reasoning algorithms—including $S_{\mathcal{EL}}$ from our running example—satisfy the *subformula property* that states that all assertions derived during the rule applications are subformulas of the input provided; i.e., of the concepts appearing in the TBox and in the subsumption relation to be verified.

We will use the transitions from the automaton to search for the preconditions of a rule application, which correspond to the successor nodes in a derivation tree. Notice, however, that while tree automata accept only *full* k-ary trees, nodes in a derivation tree can have less than k successors. To solve this issue, we will complete the trees through a distinguished new symbol $\Diamond \notin \Sigma$, that will be used as padding for labelling all the irrelevant nodes from the input tree.

Let $\mathsf{R} : (B_0, \mathcal{S}) \to \{\sigma\} \in \mathcal{R}$ with $B_0 = \{b_1, \dots, b_n\}, 1 \leq n \leq k$. We define the tuple $\delta_{\mathsf{R}} := (\sigma, b_1, \dots, b_n) \times \{\Diamond\}^{k-n}$; that is, the trailing $n - k$ symbols in the tuple δ_{R} are filled with the special symbol $\Diamond$. Using these notions, we can now construct the family of automata $\mathcal{A}_S$.

Definition 11 ($\mathcal{A}_S$). *Let $S = (\Sigma, \cdot^S, \mathcal{R}, \{\{\bot\}\})$ be a consequence-based algorithm for $\mathfrak{I}$ and $\mathfrak{T}$, and $\Gamma = (\mathcal{I}, \mathcal{T})$ be an axiomatized input. The finite tree automaton $\mathcal{A}_S(\Gamma) = (Q, \Delta, I, F)$ is defined by*

- $Q := \Sigma_\Gamma \cup \{\Diamond\}$
- $\Delta = \{\delta_{\mathsf{R}} \mid \mathsf{R} : (B_0, \mathcal{S}) \to \{\sigma\} \in \mathcal{R}, \mathcal{S} \subseteq \mathcal{T}\}$
- $I := \{\bot\}$
- $F := \Gamma^S \cup \{\Diamond\}$.

Our goal is to prove that if S is correct for the property $\mathcal{P}$, then for every axiomatized input Γ, it holds that $\Gamma \in \mathcal{P}$ iff $\mathcal{A}_S(\Gamma)$ has a successful run lab on some finite unlabelled tree with $\mathsf{lab}(\varepsilon) = \bot$. To achieve this, it would suffice to prove that the automaton is non-empty iff there exists at least one derivation tree. We will in fact provide a stronger result and show that the accepting runs of $\mathcal{A}_S(\Gamma)$ correspond exactly to the (padded) derivation trees of Γ w.r.t. S.

We start by showing that all successful runs that label the root node with $\bot$ correspond to derivation trees. Given a run $\mathsf{lab} : t \to Q$ of $\mathcal{A}_S(\Gamma)$ we define the sub-tree $t_\Sigma \subseteq t$ as the set of all nodes not labelled with $\Diamond$; in other words, $t_\Sigma := \{u \in t \mid \mathsf{lab}(u) \in \Sigma\}$. Since Δ does not have any transition with $\Diamond$ in the head, the tree t_Σ is well-defined. The labelled tree $\mathsf{lab}_\Sigma : t_\Sigma \to \Sigma$ is defined by restricting the labelling function lab to the nodes of t_Σ only. Formally, $\mathsf{lab}_\Sigma(u) = \mathsf{lab}(u)$ for all $u \in t_\Sigma$. Notice that by construction, no node in t_Σ can be labelled with the distinguished symbol $\Diamond$.

Lemma 12. *If $\mathsf{lab} : t \to Q$ is an accepting run of $\mathcal{A}_S(\Gamma)$, then $\mathsf{lab}_\Sigma : t_\Sigma \to \Sigma$ is a derivation tree for Γ w.r.t. S.*

Proof. Since $\mathsf{lab} : t \to Q$ is an accepting run, it follows that $\mathsf{lab}(\varepsilon) = \bot \in \Sigma$, and hence also $\mathsf{lab}_\Sigma(\varepsilon) = \bot$. Moreover, lab is successful. Thus, for every node $u \in t_\Sigma$, if u is a leaf, then u was also a leaf in t and thus $\mathsf{lab}_\Sigma(u) \in \Gamma^S$. If u is not a leaf, then there is a transition $(\mathsf{lab}(u), \mathsf{lab}(u1), \dots, \mathsf{lab}(uk)) \in \Delta$. By construction, this transition is of the form $(\sigma, b_1, \dots, b_n) \times \{\Diamond\}^{k-n}$ for some rule $(\{b_1, \dots, b_n\}, \mathcal{S}) \to \{\sigma\} \in \mathcal{R}$ with $\mathcal{S} \subseteq \mathcal{T}$, $n \leq k$. Thus lab_Σ satisfies also the third condition from Definition 8. □

Conversely, every derivation tree can be padded to form a full tree by adding the necessary nodes labelled with $\Diamond$. If $\mathsf{lab} : t \to \Sigma$ is a derivation tree, we construct the tree $t_\Diamond$ by adding all missing nodes needed to have a full tree from t; that is, $t_\Diamond := t \cup \{uj \mid u1 \in t, 1 \leq j \leq k\}$. The labelling function $\mathsf{lab}_\Diamond : t_\Diamond \to Q$ extends lab by mapping all new nodes to $\Diamond$:

$$\mathsf{lab}_\Diamond(u) := \begin{cases} \mathsf{lab}(u) & \text{if } u \in t \\ \Diamond & \text{otherwise} \end{cases}$$

For instance, the gray nodes and edges in Figure 2 show the padding for the derivation tree described in Example 9.

Lemma 13. *If* $\mathsf{lab} : t \to \Sigma$ *is a derivation tree for* Γ *w.r.t.* S*, then* $\mathsf{lab}_\Diamond : t_\Diamond \to Q$ *is a successful run of* $\mathcal{A}_S(\Gamma)$ *on* $t_\Diamond$ *and* $\mathsf{lab}_\Diamond(\varepsilon) = \bot \in I$*.*

Proof. First, since $\varepsilon \in t$, we immediately have that $\mathsf{lab}_\Diamond(\varepsilon) = \mathsf{lab}(\varepsilon) = \bot$, where the last equality follows from the first condition in Definition 8. Similarly, for every leaf node $u \in t_\Diamond$, if $u \in t$, then $\mathsf{lab}_\Diamond(u) = \mathsf{lab}(u) \in \Gamma^S$ and if $u \notin t$, then $\mathsf{lab}_\Diamond(u) = \Diamond$. In both cases, $\mathsf{lab}_\Diamond(u) \in F = \Gamma^S \cup \{\Diamond\}$. We now only need to show that for every non-leaf node u it holds that $(\mathsf{lab}_\Diamond(u), \mathsf{lab}_\Diamond(u1), \dots, \mathsf{lab}_\Diamond(uk)) \in \Delta$. Notice first that the nodes in $t_\Diamond$ that are labelled with $\Diamond$ are all leafs. Hence, all non-leaf nodes of this extended tree existed already in the derivation tree t. Given such a non-leaf node $u \in t$, by the conditions of derivation trees, we know that there exists a rule $\mathsf{R} : (\{\mathsf{lab}(u1), \dots, \mathsf{lab}(un)\}, \mathcal{S}) \to \{\mathsf{lab}(u)\} \in \mathcal{R}$ with $\mathcal{S} \subseteq \mathcal{T}$, and $1 \leq n \leq k$. Hence, $\delta_\mathsf{R} \in \Delta$. But

$$\begin{aligned} \delta_\mathsf{R} &:= (\mathsf{lab}(u), \mathsf{lab}(u1), \dots, \mathsf{lab}(un)) \times \{\Diamond\}^{k-n} \\ &= (\mathsf{lab}_\Diamond(u), \mathsf{lab}_\Diamond(u1), \dots, \mathsf{lab}_\Diamond(uk)), \end{aligned}$$

which concludes the proof. □

From these two lemmas it follows that the family of automata $\mathcal{A}_S(\Gamma)$ decides the same property as the consequence-based algorithm S.

Theorem 14. *Let* S *be a consequence-based algorithm that is correct for the property* $\mathcal{P}$*. For every axiomatized input* Γ *it holds that* $\Gamma \in \mathcal{P}$ *iff* $\mathcal{A}_S(\Gamma)$ *is non-empty.*

Recall from the last paragraph of Section 3 that the emptiness test iteratively constructs the set of all good states, starting from the final states, making the transition relation explicit, and at the end verifies whether an initial state is good. In the case of the automaton $\mathcal{A}_S(\Gamma)$, the final states are exactly those from Γ^S; the transition relation emulates the rules from S, and the only initial state is $\bot$ expressing that there is a clash found. Thus, the execution of the consequence-based algorithm S over the axiomatized input Γ is in fact an application of the emptiness test of the automaton $\mathcal{A}_S(\Gamma)$.

As mentioned already, the relationship between a consequence-based algorithm S and its associated family of automata $\mathcal{A}_S$ is much stronger. Rather than merely checking whether the execution of S over Γ yields the clash $\bot$, the automaton $\mathcal{A}_S(\Gamma)$ accepts all possible derivation trees. These trees can be seen as different proofs for the existence of a clash, and hence for the derivation of the properties. Notice that each of these proofs may require the presence of different axioms to trigger the rule applications. By tracing these axioms through the rule applications, it is possible to understand the axiomatic causes for this derivation, which is a fundamental task for many non-standard reasoning tasks that have been studied for many different formalisms.

5 Supplemental Reasoning

The automata $\mathcal{A}_S(\Gamma)$ can be seen as dual constructions to the axiomatic automata from [4]. Similar to that approach, associating a weight to every transition of the automaton—i.e., to every rule application of the original algorithm S—one can define the weight of every derivation tree. By extension, every axiomatized input is associated to the weight obtained from aggregating the weights of all its derivation trees. Supplemental reasoning refers to the task of computing the weight of each axiomatized input according to different interpretations.

Some of the typical examples of supplemental reasoning are axiom pinpointing [6, 12, 22] and MUS enumeration [7, 15, 18], in which the goal is to compute all the sets of axioms that entail the consequence, and its weaker version of lean kernel computation [16, 17]; probabilistic logics with distribution semantics [21]; access control [3] and reasoning with meta-knowledge [9], to name a few.

An automaton that decides a property through an emptiness test can be modified into a weighted automaton [10] as described in [4] to solve these supplemental reasoning tasks through a so-called *behaviour computation* execution. Interestingly, if the weights of the automaton (which arise from the supplemental reasoning task under consideration) form a distributive lattice, then the behaviour of this automaton can be computed in polynomial time. That is, supplemental reasoning is as expensive as standard reasoning, when the underlying reasoning method is automata-based.

6 Conclusions

We have shown that reasoning with consequence-based methods can be reduced to the emptiness test of an automaton that accepts all the derivation trees of the former. In fact, the execution of a consequence-based method and the emptiness test of its associated automaton can be seen as two faces of the same process. This duality seamlessly combines the benefits of both approaches. On the one hand, we have a method that is easy to describe, implement, and optimize; and on the other, we have the complexity bounds and supplemental reasoning extensions that automata provide. As a simple application of our techniques, we obtain the first pinpointing extension of the consequence-based approach for $\mathcal{EL}$. Further similar results can be attained by instantiating our framework.

One important consideration for future work is to consider the application of non-deterministic rules in consequence-based methods. We notice, however, that tree automata can only provide deterministic complexity classes, due to their polynomial-time emptiness test. Thus, the benefits of translating non-deterministic procedures into automata are less obvious.

References

1. Baader, F., Brandt, S., Lutz, C.: Pushing the $\mathcal{EL}$ envelope. In: Proc. of IJCAI'05. Morgan Kaufmann, Edinburgh, UK (2005)

2. Baader, F., Hladik, J., Peñaloza, R.: Automata can show PSPACE results for description logics. Information and Computation 206(9–10), 1045–1056 (2008)
3. Baader, F., Knechtel, M., Peñaloza, R.: Context-dependent views to axioms and consequences of semantic web ontologies. J. of Web Semantics 12–13, 22–40 (2012)
4. Baader, F., Peñaloza, R.: Automata-based axiom pinpointing. J. of Autom. Reas. 45(2), 91–129 (August 2010)
5. Baader, F., Peñaloza, R.: Axiom pinpointing in general tableaux. Journal of Logic and Computation 20(1), 5–34 (2010)
6. Baader, F., Peñaloza, R., Suntisrivaraporn, B.: Pinpointing in the description logic $\mathcal{EL}^+$. In: Proc. of KI 2007. LNAI, vol. 4667, pp. 52–67. Springer (2007)
7. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. 25(2), 97–116 (2012)
8. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7(3), 201–215 (1960)
9. Dividino, R.Q., Schenk, S., Sizov, S., Staab, S.: Provenance, trust, explanations - and all that other meta knowledge. KI 23(2), 24–30 (2009)
10. Droste, M., Gastin, P.: Weighted automata and weighted logics. Theoretical Computer Science 380(1-2), 69–86 (2007)
11. Gastin, P., Oddoux, D.: Fast LTL to büchi automata translation. In: Proc. of CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer (2001)
12. Kalyanpur, A., Parsia, B., Horridge, M., Sirin, E.: Finding all justifications of OWL DL entailments. In: Proc. of ISWC 2007. LNCS, vol. 4825, pp. 267–280. Springer (2007)
13. Kazakov, Y.: Consequence-driven reasoning for Horn SHIQ ontologies. In: Boutilier, C. (ed.) Proc. of IJCAI'09. pp. 2040–2045 (2009)
14. Kazakov, Y., Krötzsch, M., Simančík, F.: The incredible ELK: From polynomial procedures to efficient reasoning with $\mathcal{EL}$ ontologies. J. of Autom. Reas. 53, 1–61 (2014)
15. Kleine Büning, H., Kullmann, O.: Minimal unsatisfiability and autarkies. In: Handbook of Satisfiability, pp. 339–401 (2009)
16. Kullmann, O.: Investigations on autark assignments. Discrete Applied Mathematics 107(1-3), 99–137 (2000)
17. Kullmann, O., Lynce, I., Marques-Silva, J.: Categorisation of clauses in conjunctive normal forms: Minimally unsatisfiable sub-clause-sets and the lean kernel. In: SAT. pp. 22–35 (2006)
18. Liffiton, M.H., Previti, A., Malik, A., Marques-Silva, J.: Fast, flexible MUS enumeration. Constraints 21(2), 223–250 (2016)
19. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Information and Computation 205(4), 557–580 (2007)
20. Peñaloza, R.: Using sums-of-products for non-standard reasoning. In: Proc. of LATA 2010. LNCS, vol. 6031, pp. 488–499. Springer (2010)
21. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R.: Probabilistic description logics under the distribution semantics. Semantic Web 6(5), 477–501 (2015)
22. Schlobach, S., Cornet, R.: Non-standard reasoning services for the debugging of description logic terminologies. In: Proc. of IJCAI'03. pp. 355–362. Morgan Kaufmann (2003)