PSpace Automata for Description Logics

Jan Hladik*
Rafael Peñaloza†

Abstract

Tree automata are often used for satisfiability testing in the area of description logics, which usually yields ExpTime complexity results. We examine conditions under which this result can be improved, and we define two classes of automata, called *segmentable* and *weakly-segmentable*, for which emptiness can be decided using space logarithmic in the size of the automaton (and thus polynomial in the size of the input). The usefulness of segmentable automata is demonstrated by reproving the known PSpace result for satisfiability of $\mathcal{ALC}$ concepts with respect to acyclic TBoxes.

1 Introduction

Tableau- and automata-based algorithms are two mechanisms which are widely used for testing satisfiability in description logics (DLs). Aside from considerations regarding implementation (most of the efficient implementations are tableau-based) and elegance (tableaus for expressive logics require a blocking condition to ensure termination), there is also a difference between the complexity results which can be obtained "naturally", i.e. without using techniques with the sole purpose of remaining in a specific complexity class. With tableaus, the natural complexity class is usually NExpTime, e.g. for $\mathcal{SHIQ}$ [HST00, BS01] or $\mathcal{SHIQ}(\mathcal{D})$ [Lut04], although this result is not optimal in the former case. With automata, one usually obtains an ExpTime result, e.g. for $\mathcal{ALC}$ with general TBoxes [Sch94].

Previously, we examined which properties of a NExpTime tableau algorithm make sure that the logic is decidable in ExpTime, and we defined a class of tableau algorithms for which an ExpTime automata algorithm can be automatically derived [BHLW03]. For ExpTime automata, a frequently used

*Automata Theory, TU Dresden (jan.hladik@tu-dresden.de)
†Intelligent Systems, Uni Leipzig (rpenalozan@yahoo.com)

method to obtain a lower complexity class is testing emptiness of the language accepted by the automaton *on the fly*, i.e. without keeping the entire automaton in the memory at the same time. Examples for DLs can be found in [HST00] for $\mathcal{SI}$ and in [BN03] for $\mathcal{ALCN}$. Furthermore, the *inverse method* [Vor01] for the modal logic **K** can be regarded as an optimised emptiness test of the corresponding automaton [BT01], and also the "bottom up" version of the *binary decision diagram* based satisfiability test presented in [PSV02] can be considered as an on-the-fly emptiness test.

In this paper, our aim is to generalise these results by finding properties of ExpTime automata algorithms which guarantee that the corresponding logic can be decided in PSpace, and to develop a framework which can be used to automatically obtain PSpace results for new logics.

2 Preliminaries

We will first define the automata which in the following will be used to decide the satisfiability problem for DLs. These automata operate on infinite k-ary trees, for which the root node is identified by ε and the i-th successor of a node n is identified by $n \cdot i$ for $1 \leq i \leq k$. Thus, the set of all nodes is $\{1, \ldots, k\}^*$, and in the case of labelled trees, we will refer to the labelling of the node n in the tree t by $t(n)$.

The following definition of automata does not include an alphabet for labelling the tree nodes, because in order to decide the emptiness problem, we are only interested in the *existence* of a model and not in the labelling of its nodes.

Definition 1 (Automaton, run, accepted language.) A *Büchi automaton* over k-ary trees is a tuple (Q, Δ, I, F), where Q is a finite set of 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 *looping automaton* is a Büchi automaton where all states are accepting, i.e. $F = Q$. For simplicity, it will be written (Q, Δ, I).

A *run* of an automaton $\mathcal{A} = (Q, \Delta, I, F)$ on a k-ary tree t is a labelled k-ary tree r such that $r(\varepsilon) \in I$ and, for all $w \in \{1, \ldots, k\}^*$, it holds that $(r(w), r(w \cdot 1), \ldots, r(w \cdot k)) \in \Delta$. A run r is *accepting* if every path of r contains a final state infinitely often.

The *language accepted by* $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$, is the set of all trees t such that there exists an accepting run of $\mathcal{A}$ on t.

For the DL $\mathcal{ALC}$ [SS91], it is well-known that looping automata can be used to decide satisfiability of a concept C: we define a looping automaton $\mathcal{A}_C = (Q, \Delta, I)$, where the set of states Q consists of (propositionally expanded and clash-free) sets of subformulas of C, and the transition relation Δ ensures

that the successor states of a state q contain those concepts which are required by the universal and existential concepts in q. Since $\#Q$, the cardinality of Q, is exponential in the size of C and $\mathcal{T}$ (because it is bounded by the number of sets of subformulas of C and $\mathcal{T}$) and the emptiness test is linear [BT01, VW86] in the cardinality of Q, this yields an ExpTime algorithm. This result is optimal for $\mathcal{ALC}$ with general TBoxes, but not for $\mathcal{ALC}$ with acyclic (or empty) TBoxes.

In the following, we will define this algorithm in detail. We assume that *acyclic* TBoxes are sets of concept definitions $A_i \doteq C_i$, for concept names A_i and concept terms C_i, where there is at most one definition for a concept name and there is no sequence of concept definitions $A_1 \doteq C_1, \ldots, A_n \doteq C_n$ such that C_i contains A_{i+1} for $1 \leq i < n$ and C_n contains A_1. In contrast, *general* TBoxes can additionally contain general concept inclusion axioms (GCIs) of the kind $C_i \sqsubseteq D_i$ for arbitrary concept terms C_i and D_i. For the sake of simplicity, we will assume that all concepts are in *negation normal form (NNF)*, i.e. negation appears only directly before concept names. All $\mathcal{ALC}$ concepts can be transformed into NNF in linear time using de Morgan's laws and their analogues for universal and existential formulas. We will denote the NNF of a concept C by $\mathsf{nnf}(C)$ and $\mathsf{nnf}(\neg C)$ by $\backsim C$.

The data structures that will serve as models for $\mathcal{ALC}$ concepts are *Hintikka trees*, a special kind of trees whose nodes are labelled with propositionally expanded and clash-free sets of $\mathcal{ALC}$ concepts.

Definition 2 (Sub-concept, Hintikka set, Hintikka tree.) Let C be an $\mathcal{ALC}$ concept term. The set of *sub-concepts of* C, $\mathsf{sub}(C)$, is the minimal set S which contains C and has the following properties: if S contains $\neg A$ for a concept name A, then $A \in S$; if S contains $D \sqcup E$ or $D \sqcap E$, then $\{D, E\} \subseteq S$; if S contains $\exists r.D$ or $\forall r.D$, then $D \in S$.

For a TBox $\mathcal{T}$, $\mathsf{sub}(C, \mathcal{T})$ is defined as follows:

$$\mathsf{sub}(C) \quad \cup \bigcup_{A \doteq D \in \mathcal{T}} (A \cup \mathsf{sub}(D) \cup \mathsf{sub}(\backsim D)) \quad \cup \bigcup_{C \sqsubseteq D \in \mathcal{T}} \mathsf{sub}(\backsim C \sqcup D)$$

A set $H \subseteq \mathsf{sub}(C, \mathcal{T})$ is called a *Hintikka set for* C if the following three conditions are satisfied: if $D \sqcap E \in H$, then $\{D, E\} \subseteq H$; if $D \sqcup E \in H$, then $\{D, E\} \cap H \neq \emptyset$; there is no concept name A with $\{A, \neg A\} \subseteq H$.

For a TBox $\mathcal{T}$, a Hintikka set S is called $\mathcal{T}$*-expanded* if for every GCI $C \sqsubseteq D \in \mathcal{T}$, it holds that $\backsim C \sqcup D \in S$, and for every concept definition $A \doteq C \in \mathcal{T}$, it holds that if $A \in S$ then $C \in S$ and if $\neg A \in S$ then $\backsim C \in S$. We will refer to this technique of handling definitions as *lazy unfolding* because, in contrast to GCIs, we use the definition only if A or $\neg A$ is explicitly present in a node label.

For a concept term C and TBox $\mathcal{T}$, fix an ordering of the existential concepts in $\mathsf{sub}(C, \mathcal{T})$ and let $\varphi : \{\exists r.D \in \mathsf{sub}(C, \mathcal{T})\} \to \{1, \ldots, k\}$ be the corresponding

ordering function. Then the tuple $(S, S_1, \ldots, S_k)$ is called $C, \mathcal{T}$*-compatible* if, for every existential formula $\exists r.D \in \mathsf{sub}(C, \mathcal{T})$, it holds that if $\exists r.D \in S$, then $S_{\varphi(\exists r.D)}$ contains D and every concept E_i for which there is a universal formula $\forall r.E_i \in S$.

A k-ary tree t is called a *Hintikka tree for* C *and* $\mathcal{T}$ if, for every node $n \in \{1, \ldots, k\}^*$, $t(n)$ is a $\mathcal{T}$-expanded Hintikka set and the tuple $(t(n), t(n{\cdot}1), \ldots, t(n{\cdot}k))$ is $C, \mathcal{T}$-compatible.

Note that in a Hintikka tree t, the k-th successor of a node n stands for the individual satisfying the k-th existential formula D if $D \in t(n)$. We can use Hintikka trees to test the satisfiability in $\mathcal{ALC}$:

Lemma 3 An $\mathcal{ALC}$ concept term C is satisfiable w.r.t. a TBox $\mathcal{T}$ iff there is a $C, \mathcal{T}$-compatible Hintikka tree t with $C \in t(\varepsilon)$.

Proof sketch. The proof is similar to the one presented for $\mathcal{ALC}^{\neg}$ in [LS00] where the "if" direction is shown by defining a model $(\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ from a Hintikka tree in the following way:

- $\Delta^{\mathcal{I}} := \{n \in \{1, \ldots, k\}^* \mid n = \varepsilon \text{ or } n = m \cdot \varphi(\exists r.D) \text{ for some } m \text{ with } \exists r.D \in t(m)\}$;
- for a role name r, $r^{\mathcal{I}} := \{(n, n \cdot i) \mid \exists r.C \in t(n) \text{ and } \varphi(\exists r.C) = i\}$;
- for a concept name A, $A^{\mathcal{I}} := \{n \mid A \in t(n)\}$;
- the function $\cdot^I$ is extended to concept terms in the natural way.

For our Hintikka trees, we have to modify the definition of $A^{\mathcal{I}}$ for concept names A in order to deal with TBoxes. For a GCI $C \sqsubseteq D$, it follows immediately from the definition of $\mathcal{T}$-expanded that a node whose label contains C also contains D. However, concept definitions need special consideration because due to the lazy unfolding of $A \doteq C$, a node label might contain C but not A, thus we have to modify the definition of $A^{\mathcal{I}}$. To this end, we define a hierarchy $\prec$ on concept names in such a way that if a concept name A appears in the definition of B, then $A \prec B$. As the concept definitions are acyclic, this hierarchy is well-founded. When we define the interpretation of concept names, we start with the lowest ones in the hierarchy, i.e. the primitive concepts, and define $A^{\mathcal{I}} := \{n \in \Delta^{\mathcal{I}} \mid A \in t(n)\}$. Then we move gradually up in the hierarchy and define, for a defined concept $B \doteq C$, $B^{\mathcal{I}} = \{n \in \Delta^{\mathcal{I}} \mid B \in t(n)\} \cup \{n \in \Delta^{\mathcal{I}} \mid n \in C^{\mathcal{I}}\}$. This is well-defined because the interpretation of all concept names appearing in C has already been defined.

Our Hintikka trees differ from those in [LS00] in that, if an existential formula $\exists r.D$ is not present in a node n, we do not require that $n \cdot \varphi(\exists r.D)$ is labelled with $\emptyset$. However, it can still be shown that if a concept term C is contained in

the label of a node n of the Hintikka tree, then $n \in C^{\mathcal{I}}$, because in the definition of the interpretation for the role names, we consider only the successors that will satisfy the existential restrictions of a node, and pay no attention to any other possible successors. The universal restrictions are then immediately satisfied by the definition of C,$\mathcal{T}$-compatible.

The "only-if" direction does not require significant modifications, because if there is a model for a concept C and a TBox $\mathcal{T}$, the node labels of the Hintikka tree constructed as in [LS00] can easily be extended to reflect the constraints imposed by the TBox. ■

With this result, we can use automata operating on Hintikka trees to test for the existence of models. As mentioned before, we can omit the labelling of the tree, since we are only interested in the existence of a model and all relevant information to answer this question is kept in the transition relation.

Definition 4 (Automaton $\mathcal{A}_{C,\mathcal{T}}$.) For a concept C and a TBox $\mathcal{T}$, let k be the number of existential formulas in $\mathsf{sub}(C,\mathcal{T})$. Then the looping automaton $\mathcal{A}_{C,\mathcal{T}} = (Q, \Delta, I)$ is defined as follows: $Q = \{S \subseteq \mathsf{sub}(C,\mathcal{T}) \mid S \text{ is a } \mathcal{T}\text{-expanded Hintikka set}\}$; $\Delta = \{(S, S_1, \ldots, S_k) \mid (S, S_1, \ldots, S_k) \text{ is } C,\mathcal{T}\text{-compatible}\}$; $I = \{S \in Q \mid C \in S\}$.

Using $\mathcal{A}_{C,\mathcal{T}}$, we can reduce the satisfiability problem for $\mathcal{ALC}$ to the (non-) emptiness problem of $\mathcal{L}(\mathcal{A}_{C,\mathcal{T}})$. Since these results are well known, they will not be formally proved.

Theorem 5 The language accepted by the automaton $\mathcal{A}_{C,\mathcal{T}}$ is empty iff C is unsatisfiable w.r.t. $\mathcal{T}$.

Corollary 6 Satisfiability of $\mathcal{ALC}$ concepts w.r.t. general TBoxes is decidable in ExpTime.

For general TBoxes, this complexity bound is tight [Spa93], but in the special case of an empty or acyclic TBoxes, it can be improved to PSpace. Usually, this is proved using a tableau algorithm, but in the next section we will show how the special properties of acyclic TBoxes can be used to perform the emptiness test of the automaton with logarithmic space.

3 Segmentable automata

In this section we will show how the space efficiency for the construction and the emptiness test of the automaton can be improved under specific conditions. The idea is to define a hierarchy of states and ensure that the level of the state decreases with every transition. In Section 4 we will then show how the role depth of concepts can be used to define this hierarchy.

```
guess an initial state q ∈ I
if there is a transition from q then
   guess a transition (q, q_1, ..., q_k) ∈ Δ
else
   return "empty"
end if
push (SQ, (q_1, ..., q_k)); push (SN, 0)
while SN is not empty do
   (q_1, ..., q_k) := pop (SQ)
   n := pop (SN) + 1
   if n ≤ k then
      push(SQ,(q_1, ..., q_k))
      push(SN,n)
      if q_n ∉ Q_0 then
         if there is a transition from q_n then
            guess a transition (q_n, q'_1, ..., q'_k)
         else
            return "empty"
         end if
         push(SQ,(q'_1, ..., q'_k))
         push(SN,0)
      end if
   end if
end while
return "not empty"
```

Figure 1: Emptiness test for segmentable automata

Definition 7 (Q_0-looping, m-segmentable.) Let $\mathcal{A} = (Q, \Delta, I, F)$ be a Büchi automaton over k-ary trees and $Q_0 \subseteq F$. We call $\mathcal{A}$ Q_0-*looping* if for every $q \in Q_0$ there exists a set of states $\{q_1, \ldots, q_k\} \subseteq Q_0$ such that $(q, q_1, \ldots, q_k) \in \Delta$.

An automaton $\mathcal{A} = (Q, \Delta, I, F)$ is called m-*segmentable* if there exists a partition $Q_0, Q_1, \ldots, Q_m$ of Q such that $\mathcal{A}$ is Q_0-looping and, for every $(q, q_1, \ldots, q_k) \in \Delta$, it holds that if $q \in Q_n$, then $q_i \in Q_{<n}$ for $1 \leq i \leq k$, where $Q_{<n}$ denotes $Q_0 \cup \bigcup_{j=1}^{n-1} Q_j$.

Note that it follows immediately from this definition that for every element q of Q_0 there exists an infinite tree with q as root which is accepted by $\mathcal{A}$. The hierarchy $Q_m, \ldots, Q_0$ ensures that Q_0 is reached eventually.

Our algorithm performing the emptiness test for m-segmentable Büchi automata is shown in Figure 1. Essentially, we perform a depth-first traversal of a run. Since $\mathcal{A}$ is m-segmentable, we do not have to go to a depth larger than

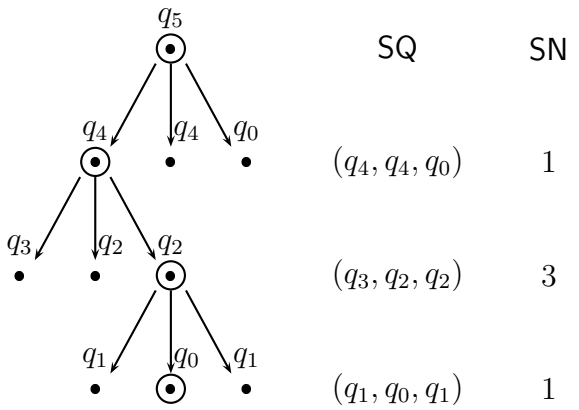

Figure 2: state of SQ, SN and the associated run, at an iteration of the algorithm

m. Moreover, since the different branches of the tree are independent, we only have to keep one of them in memory at a time. Note that the construction of the automaton is interleaved with the emptiness test, so we also never keep the whole automaton in memory, but only the states which are relevant for the current branch.

In order to remember the backtracking information for the depth first traversal, we use two data structures: SQ is a stack storing, for every predecessor of the current node, the transition which led to that node, and thus it contains the required node labels for the nodes of the current branch and their siblings. SN is another stack recording the current path by storing, for every level of the tree, the number of the node on the current path. If we refer to the elements in SN by $\mathsf{SN}(1)$(the bottom element), $\ldots$, $\mathsf{SN}(d)$(the top element), the next node to be checked is $\mathsf{SN}(1) \cdot \mathsf{SN}(2) \cdot \ldots \cdot \mathsf{SN}(d) + 1$ (d is the depth of SN). Thus, $\mathsf{SQ} \in (Q^k)^*$, because every transition is a k-ary tuple, and $\mathsf{SN} \in \{1, \ldots, k\}^*$.

Figure 2 shows the values stored in each of the stacks SQ and SN at the beginning of an iteration, and their relation with the traversal of the run. The circled nodes represent the path followed to reach the node about to be checked. The values of the elements of the stack are shown next to the depth in the run to which they correspond. For this reason, the stacks appear backwards, with their bottom element at the top of the figure, and *vice versa*.

After starting the algorithm, we first guess an initial transition. If we can find one, we push the labels of the nodes $1, \ldots, k$ onto SQ and the number 0 onto SN. Then we enter the while loop. As long as the stacks are not empty, we take the top elements of both stacks. If $n > k$ in line 11, this indicates that we have checked all nodes on this level, and we backtrack without pushing anything on the stacks, which means that we will continue at the next upper level in the next loop. Otherwise, we store the information that we have to check our next sibling

by pushing the same tuple of states onto SQ and the incremented number n onto SN. If the current node belongs to Q_0 (line 14), we backtrack, which means that we will continue with the next sibling. Otherwise, we try to guess a transition from this node, and if we can find one, we push the required node labels for the children of the current node onto SQ and the value 0 onto SN (line 20), which means that we will descend to the first child of the current node in the next loop.

Theorem 8 The emptiness problem of the language accepted by an m-segmentable Büchi automaton $\mathcal{A} = (Q, \Delta, I, F)$ over k-ary trees can be decided by a non-deterministic algorithm using space $O(\log(\#Q) \cdot m \cdot k)$.

Proof. In order to show soundness, we will prove the claim "if the algorithm processes a node n or backtracks without descending into n, then there is a run r in which n is labelled with the same state as in the algorithm" by induction over the iterations of the while loop. Initially, if the algorithm does not answer "empty", there is a transition $(q_0, q_1, \ldots, q_k)$ from an initial state, which can serve as root of the run r, and for which the states $1, \ldots, k$ of r can be labelled with $q_1, \ldots, q_k$.

If the algorithm has reached a node $n = n_0 \cdot n_1 \cdot \ldots \cdot n_\ell$ without failing, it follows by induction hypothesis that each of the previously visited nodes corresponds to a node in r. Now there are two possibilities: firstly, if $r(n) \in Q_0$, then, since $\mathcal{A}$ is Q_0-looping, there exists a k-ary subtree rooted at n all of whose states are accepting. Otherwise, since the algorithm does not answer "empty", there is a transition $(r(n), q'_1, \ldots, q'_k)$, and we can use the same transition in the construction of a run.

In order to show completeness, we will prove the claim "if there exists a run, the algorithm can reach or skip every node in $\{1, \ldots, k\}^*$ without failing" by induction over the structure of the run r. Since there is a run, we can guess an initial transition, and the nodes of the first level have the same labels in the algorithm as in r. If we have reached a node n which corresponds to the node n in r with $r(n) = q$, there are again two possibilities: if $q \in Q_0$, the algorithm will backtrack and skip over all successor nodes of n. Otherwise, since r is a run, there exists a transition $(q, q'_1, \ldots, q'_k)$, which the algorithm can guess, and therefore it will not fail.

Regarding memory consumption, observe that the SQ stack contains, for every level, k states, each of which can be represented using space logarithmic in the number of states, e.g. by using binary coding. Since $\mathcal{A}$ is m-segmentable, there can be at most m tuples before the current state q_n belongs to Q_0, thus the size of SQ is bounded by $\log(\#Q) \cdot m \cdot k$. SN stores at most m numbers between 0 and k, so the algorithm uses space logarithmic in the size of $\mathcal{A}$. ■

The condition that an automaton $\mathcal{A}$ is m-segmentable is rather strong since it requires the transition relation to reduce the class with *every* possible transition, and thus e.g. the automaton $\mathcal{A}_{C,\mathcal{T}}$ in Definition 4 cannot easily be proved to be segmentable even if the TBox is empty. The reason for this is that $\mathcal{A}_{C,\mathcal{T}}$ does not require the Hintikka sets of the successor states to use only a lower quantification depth. However, in order to test emptiness, we only need the *existence* of such a transition. This is the idea behind the generalisation in the following definition.

Definition 9 (Weakly-m-segmentable, reduced.) A Büchi automaton $\mathcal{A} = (Q, \Delta, I, F)$ is called *weakly-m-segmentable* if there exists a partition $Q_0, Q_1, \ldots, Q_m$ of Q such that $\mathcal{A}$ is Q_0-looping and for every $q \in Q$ there exists a function $f_q : Q \to Q$ which satisfies the following conditions:

1. if $(q, q_1, \ldots, q_k) \in \Delta$, then $(q, f_q(q_1), \ldots, f_q(q_k)) \in \Delta$, and if $q \in Q_n$, then $f_q(q_i) \in Q_{<n}$ for all $1 \leq i \leq k$;
2. if $(q', q_1, \ldots, q_k) \in \Delta$, then $(f_q(q'), f_q(q_1), \ldots, f_q(q_k)) \in \Delta$.

If $\mathcal{A} = (Q, \Delta, I, F)$ is a weakly-m-segmentable automaton, then $\mathcal{A}^{\mathrm{r}}$, the *reduced automaton of* $\mathcal{A}$, is defined as follows: $\mathcal{A}^{\mathrm{r}} = (Q, \Delta', I, F)$ with $\Delta' = \{(q, q_1, \ldots, q_m) \in \Delta \mid \text{ if } q \in Q_n \text{ then } q_i \in Q_{<n} \text{ for } 1 \leq i \leq k\}$.

Note that the reduced automaton $\mathcal{A}^{\mathrm{r}}$ is m-segmentable by definition. Intuitively, condition 1 ensures that that the class decreases for the first transition, and condition 2 ensures that there are still transitions for *all* nodes after modifying the node labels according to f_q.

We can transfer the complexity result from segmentable to weakly-segmentable automata:

Theorem 10 Let $\mathcal{A} = (Q, \Delta, I, F)$ be a weakly-m-segmentable automaton. Then $\mathcal{L}(\mathcal{A})$ is empty iff $\mathcal{L}(\mathcal{A}^{\mathrm{r}})$ is empty.

Proof. Since every run of $\mathcal{A}^{\mathrm{r}}$ is also a run of $\mathcal{A}$, $\mathcal{L}(\mathcal{A})$ can only be empty if $\mathcal{L}(\mathcal{A}^{\mathrm{r}})$ is empty, thus the "only if" direction is obvious. For the "if" direction, we will show how to transform an accepting run r of $\mathcal{A}$ into an accepting run s of $\mathcal{A}^{\mathrm{r}}$. To do this, we traverse r breadth-first, creating an intermediate run $\hat{r}$, which initially is equal to r. At every node $n \in \{1, \ldots, k\}^*$, we replace the labels of the direct and indirect successors of n with their respective f_n values (see Definition 9). More formally, at node n, we replace $\hat{r}(p)$ with $f_n(\hat{r}(p))$ for all $p \in \{n \cdot q \mid q \in \{1, \ldots, k\}^+\}$, where for a set S, S^+ denotes $S^* \setminus \{\varepsilon\}$. By definition 9, $\hat{r}$ is still a run after the replacement, and all direct successors of n are in a lower class than n (or Q_0 if $\hat{r}(n) \in Q_0$). Note that the labels of n's successors are not modified anymore after n has been processed. We can

therefore define $s(n)$ as the value of $\hat{r}(n)$ after n has been processed. As argued before, s is a run in which every node is in a lower class than its father node (or both are in class 0). Consequently, all transitions used in s belong to the transition relation of $\mathcal{A}^{\mathrm{r}}$. ■

Corollary 11 The emptiness problem for weakly-m-segmentable automata is in NLogSpace.

4 An application to $\mathcal{ALC}$ with acyclic TBoxes

In order to apply our framework to $\mathcal{ALC}$ with acyclic TBoxes, we will use the role depth to define the different classes. The following definition of role depth considers concept definitions:

Definition 12 (Expanded role depth.) For an $\mathcal{ALC}$ concept C and an acyclic TBox $\mathcal{T}$, the *expanded role depth* $\mathsf{rd}_{\mathcal{T}}(C)$ is inductively defined as follows: for a primitive role name A, $\mathsf{rd}_{\mathcal{T}}(A) = 0$; for a concept definition $A \doteq C$, $\mathsf{rd}_{\mathcal{T}}(A) = \mathsf{rd}_{\mathcal{T}}(C)$; $\mathsf{rd}_{\mathcal{T}}(\neg A) = \mathsf{rd}_{\mathcal{T}}(A)$; $\mathsf{rd}_{\mathcal{T}}(D \sqcap E) = \mathsf{rd}_{\mathcal{T}}(D \sqcup E) = \max\{\mathsf{rd}_{\mathcal{T}}(D), \mathsf{rd}_{\mathcal{T}}(E)\}$; $\mathsf{rd}_{\mathcal{T}}(\forall r.D) = \mathsf{rd}_{\mathcal{T}}(\exists r.D) = \mathsf{rd}_{\mathcal{T}}(D) + 1$. For a set of concepts S, $\mathsf{rd}_{\mathcal{T}}(S)$ is defined as $\max\{\mathsf{rd}_{\mathcal{T}}(D) \mid D \in S\}$.

The set $\mathsf{sub}_{<n}(C, \mathcal{T})$ is defined as $\{D \in \mathsf{sub}(C, \mathcal{T}) \mid \mathsf{rd}_{\mathcal{T}}(D) \leq \max\{0, n-1\}\}$.

Again, note that $\mathsf{rd}_{\mathcal{T}}$ is well-defined because $\mathcal{T}$ is acyclic.

The intuition behind using the role depth is that, for a node q in a Hintikka tree, any concept having a higher role depth than q is superfluous in successors of q and thus the tuple without these formulas is also in the transition relation.

Lemma 13 Let C be an $\mathcal{ALC}$ concept, $\mathcal{T}$ an acyclic TBox, $(S, S_1, \ldots, S_k)$ a C,$\mathcal{T}$-compatible tuple, and $n = \mathsf{rd}_{\mathcal{T}}(S)$. Then $(S, \mathsf{sub}_{<n}(C, \mathcal{T}) \cap S_1, \ldots, \mathsf{sub}_{<n}(C, \mathcal{T}) \cap S_k)$ is C,$\mathcal{T}$-compatible, and for every $m \geq 0$, the tuple $(\mathsf{sub}_{<m}(C, \mathcal{T}) \cap S, \mathsf{sub}_{<m}(C, \mathcal{T}) \cap S_1, \ldots, \mathsf{sub}_{<m}(C, \mathcal{T}) \cap S_k)$ is C,$\mathcal{T}$-compatible.

Proof. We need to show that the conditions in Definition 2 are satisfied for both tuples. In the case of the first tuple suppose that $\exists r.D \in S$. Then, $S_{\varphi(\exists r.D)}$ contains D and every concept E_i for which there is a universal formula $\forall r.E_i \in S$. But since $\mathsf{rd}_{\mathcal{T}}(D) < \mathsf{rd}_{\mathcal{T}}(\exists.rD) \leq n$ and $\mathsf{rd}_{\mathcal{T}}(E_i) < \mathsf{rd}\mathcal{T}(\forall r.E_i) \leq n$, it holds that $\mathsf{sub}_{<n}(C, \mathcal{T}) \cap S_{\varphi(\exists r.D)}$ contains D and each of the E_i concepts.

For the second tuple, if $\exists r.D \in \mathsf{sub}_{<m}(C, \mathcal{T}) \cap S$, then $\mathsf{rd}_{\mathcal{T}}(\exists r.D) < m$ and $D \in S_{\varphi(\exists r.D)}$. If additionally there is a concept term E_i such that the universal formula $\forall r.E_i \in \mathsf{sub}_{<m}(C, \mathcal{T}) \cap S$, then again $\mathsf{rd}_{\mathcal{T}}(E_i) < m$ and $E_i \in S_{\varphi(\exists r.D)}$. Hence, $\mathsf{sub}_{<m}(C, \mathcal{T}) \cap S_{\varphi(\exists r.D)}$ contains D and each such concept E_i. ■

Theorem 14 Let C be an $\mathcal{ALC}$ concept, $\mathcal{T}$ an acyclic TBox and $m = \max\{\mathsf{rd}_{\mathcal{T}}(D) \mid D \in \mathsf{sub}(C, \mathcal{T})\}$. Then $\mathcal{A}_{C,\mathcal{T}}$ is weakly-m-segmentable.

Proof. We have to give the segmentation of Q and the functions f_q and show that they satisfy the conditions in Definition 9. Define the classes $Q_i := \{S \in Q \mid \mathsf{rd}_{\mathcal{T}}(S) = i\}, 0 \leq i \leq m$ and $f_q(q') := q' \cap \mathsf{sub}_{<n}(C, \mathcal{T})$, where $n = \mathsf{rd}_{\mathcal{T}}(q)$ for every $q, q' \in Q$. By this definition, it is obvious that for every q', $f_q(q')$ is in a lower class than q (or in Q_0 if $q \in Q_0$). Lemma 13 shows that conditions 1 and 2 of Definition 9 are satisfied. It remains to show that $\mathcal{A}_{C,\mathcal{T}}$ is Q_0-looping. If $q \in Q_0$, there are no existential formulas in q, and therefore $(q, \emptyset, \ldots, \emptyset) \in \Delta$. ■

Although our algorithm in Figure 1 is non-deterministic, we can obtain a deterministic complexity class by using Savitch's theorem [Sav70].

Corollary 15 Satisfiability of $\mathcal{ALC}$ concepts with respect to acyclic TBoxes is in PSPACE.

5 Conclusion

We have introduced segmentable and weakly-segmentable Büchi automata, two classes of automata for which the emptiness problem of the accepted language is decidable in NLOGSPACE, whereas in general the complexity class of this problem for Büchi automata is P. The complexity bound is proved by testing the possibility of a model on the fly using depth-first search. This generalises previous results of on-the-fly emptiness tests for several modal and description logics. As an example, we showed how our framework can be used to obtain the PSPACE upper complexity bound for $\mathcal{ALC}$ with acyclic TBoxes in an easy way. We hope that this framework will make it easier to prove a PSPACE upper bound also for new logics.

Acknowledgements

We would like to thank Franz Baader for fruitful discussions. We also thank the reviewers for suggesting improvements of this paper.

References

[BHLW03] F. Baader, J. Hladik, C. Lutz, and F. Wolter. From tableaux to automata for description logics. *Fundamenta Informaticae*, 57:1–33, 2003.

[BN03] F. Baader and W. Nutt. *The Description Logic Handbook*, chapter 2: Basic Description Logics. Cambridge University Press, 2003.

[BS01] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69, 2001.

[BT01] F. Baader and S. Tobies. The inverse method implements the automata approach for modal satisfiability. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of IJCAR-01*, volume 2083 of *LNAI*. Springer-Verlag, 2001.

[HST00] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–264, 2000.

[LS00] C. Lutz and U. Sattler. Mary likes all cats. In F. Baader and U. Sattler, editors, *Proceedings of DL 2000*, CEUR Proceedings, 2000.

[Lut04] C. Lutz. NExpTime-complete description logics with concrete domains. *ACM Transactions on Computational Logic*, 5(4):669–705, 2004.

[PSV02] G. Pan, U. Sattler, and M. Y. Vardi. BDD-based decision procedures for K. In *Proceedings of the Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Artificial Intelligence*, 2002.

[Sav70] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of computer and system sciences*, 4(2):177–192, 1970.

[Sch94] K. Schild. Terminological cycles and the propositional μ-calculus. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Proceedings of KR-94*. Morgan Kaufmann, 1994.

[Spa93] E. Spaan. *Complexity of Modal Logics*. PhD thesis, University of Amsterdam, 1993.

[SS91] M. Schmidt-Schauß and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[Vor01] A. Voronkov. How to optimize proof-search in modal logics: new methods of proving reduncancy criteria for sequent calculi. *ACM transactions on computational logic*, 2(2), 2001.

[VW86] M. Y. Vardi and P. Wolper. Automata-theoretic techniques for modal logics of programs. *J. of Computer and System Science*, 32:183–221, 1986.