Dresden University of Technology
Department of Computer Science
Institute for Theoretical Computer Science

Optimization of emptiness test of Büchi automata on infinite trees

Rafael Peñaloza

Master Thesis

Supervisor: Jan Hladik
Supervising University Professor: Prof. Franz Baader

Contents

A ustedes, todos ustedes, que fueron mi impulso de subida, y mi freno de bajada.

Abstract

Automata theory has proven to be a useful tool for solving decision problems, for example, the satisfiability problem in some logics. This approach consists basically in reducing the desired problem to the emptiness problem of automata.

Unfortunately, this reduction leads in some cases to suboptimal methods. When this happens, it is usually possible to find properties in the specific automata that allow the emptiness test to be solved in an optimized manner. This path has been followed many times, but always done solely for the specific automata at the time.

In this work general conditions are given, under which the emptiness test can be solved using only logarithmic space on the number of states. These conditions are later applied to prove that $\mathcal{ALC}$-satisfiability with respect to empty and acyclic TBoxes is in PSpace.

Declaration

Herewith I confirm that I independently prepared this thesis. No further references or auxiliary means than the ones indicated in this work were used for the preparation.

Signature: _____

Rafael Penaloza

Chapter 1

Introduction

In a world in which information and knowledge grow at gigantic steps, the ability to handle them, even in small, specialized parts, agonizes. During the time required to find and gather a piece of knowledge, new elements are found and developed. This makes it almost impossible to stay up-to-date within any knowledge domain.

Human beings have long since resigned to delegate this task to computers. These machines, with a virtually unlimited memory and ever growing computing power, seem like the natural prospects to cope with the weight of this assignment. The ability to store every segment of relevant knowledge, and process them expeditiously are their main desired properties.

Unfortunately, a transfer from human-based to machine-based knowledge handling is not as easy as it might be aspired to be. On one hand, the innate inflexibility of computers constrains knowledge to be formally stated. On the other, one source is usually insufficient to model even relatively limited and well-examined knowledge areas.

In the search of solving the first of these problems, many formalisms have been developed and studied. The basic idea in most of them consists simply in describing the relevant parts of the knowledge domain, and the relations between them. Well-studied examples of such formalisms are the logic-based ones; in particular, Description Logics.

Multiplicity of sources for modeling is not a problem *per se*, apart from the inconvenience of needing to consult several informants. An unsupervised increase of authors raises, nonetheless, the possibility of finding contradictions, errors, and misinterpretations of the model.

Were the only goal of the model mere *erudition*, such miscues would be little more than irrelevant nuisances. As the knowledge is intended, on the contrary, to be employed in solving reasoning tasks, absurdities or mistakes cannot be tolerated.

One way of distinguishing errors in the model is by means of satisfiability tests. Whenever a complex concept is defined, it is safe to assume that this concept should be satisfiable, as there is no need of giving an intricate construction of inanity. If one such concept results being unsatisfiable within the knowledge base, it is likely that wrong assertions were executed during the modeling process.

Inquiring whether concepts define only absurdities or not has, nonetheless, more motivation than just error pruning. As usually only the relevant parts of knowledge are stated explicitly in the model, most of the knowledge is stated implicitly. This capacity of keeping implicit expertise is desired on these systems as a matter of efficiency. Reasoning tasks, such as the satisfiability test, work then as guides to openly express these parts of the knowledge domain.

Depending on the used formalism, different mechanisms can be used in order to verify satisfiability of concepts. One common approach relies on the well-studied Automata Theory. In particular, the problem is usually reduced to that of verifying whether the language defined by an automaton is empty or not. This last problem has been deeply considered, and several methods for solving it have been developed.

But the seek is not to stop there. As these tests are really meant to be computed, the mere theoretical capacity of performing them is unsatisfactory. It is also necessary to be able to do it within *reasonable* resource bounds.

In other words, it is not enough to find *any* decision procedure for the satisfiability test of concepts, but rather one that is optimal in the use of resources such as time and space.

It turns out that, in many cases, the *natural* automata theoretic approach yields to suboptimal procedures; that is, methods that consume more resources than really required by the problem. However, when this happens, it is sometimes possible to adapt the method to obtain the optimal decision procedure.

The readjustment process is usually as follows: reduce the satisfiability problem of concepts to the emptiness problem of automata, and then verify

if it is possible to solve this problem with less resources as usual, taking into account the properties of the resulting automata. These results in a less costly decision procedure than the initial automaton.

Whenever this process has been followed, the properties of the automata were examined, and a new decision procedure for the emptiness of these automata was developed, useful only in these particular automata. This exclusiveness of use arises from the use of features exclusive of the studied class of automata.

The question then arises: is it possible to find conditions on the automata, for which the emptiness problem may be optimized? In other words, is it within reach to generalize the optimization process, and make it useful in other applications?

This work focuses in an answer to that inquiry. A class of automata is defined, for which the emptiness test can be done using logarithmic space in the number of states, in contrast to the polynomial time required in general. This means that, if the size of the automaton is exponential in the size of the concept for which the satisfiability test is to be performed, the overall process will require polynomial space, which is a significant improvement to the original exponential time method.

The work is distributed as follows. In Chapter 2, the basic notions of automata theory, and the description logic $\mathcal{ALC}$ are defined. These concepts will be necessary for the rest of the work. After that, Chapter 3 defines the class of automata for which the emptiness test can be optimized, and shows a way to make such a test, proving its correctness.

Chapter 4 holds then an example of how these results may be applied to show that $\mathcal{ALC}$ satisfiability, and $\mathcal{ALC}$ satisfiability with respect to acyclic TBoxes is in PSpace.

Afterwards, the last chapter holds some conclusions for this work, as well as some ideas regarding future work.

Chapter 2

Theoretical Background

This chapter introduces the basic concepts and results that will be used in the following chapters. The first section focuses on defining the basic concepts of automata theory and the emptiness problem.

The second section defines an instance of a Description Logic called $\mathcal{ALC}$, and how its decision problems can be related to those of automata. More precisely, it explains how automata theory can be used to decide the satisfiability problem of concept terms with respect to general TBoxes using exponential time in the size of the concept term. This should work as a motivation for the framework presented in Chapter 3, in which conditions are defined such that the emptiness problem can be solved in a more efficient manner for the automata that meet them.

With the help of that theory, two subproblems of satisfiability with respect to general TBoxes can be shown to be solvable using only polynomial space, as will be seen in Chapter 4.

2.1 Automata Theory

Intuitively, an automaton can be seen as a machine that traverses an input and decides whether this input is part of a set, called language, or not. Depending on the type of accepting condition, and the type of input, many kinds of automata exist.

This section is an introduction to Büchi automata over infinite unlabeled trees; that is, automata with the Büchi accepting condition that receive

infinite unlabeled trees as input.

Definition 2.1 *A* Büchi automaton over infinite k-ary unlabeled trees *is a tuple of the form* $M = (Q, \Delta, I, F)$ *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*;*
- $F \subseteq Q$ *is the set of* final states*.*

Automata are usually defined to receive as input *labeled* trees. For that reason, they usually include an extra parameter, Σ, which states the alphabet used to label the nodes. For the present framework, such labels are not necessary, and so they are not considered at all, for sake of simplicity.

An automaton over unlabeled trees can only receive one input: the unique infinite k-ary unlabeled tree. Hence, it will be always assumed that this is indeed the input given. For abbreviation, whenever infinite k-ary unlabeled trees are used, it will simply be written as *k-ary trees*.

Every node in a k-ary tree will be represented by a word in the set $\{1, \ldots, k\}^*$ stating the unique segment of path that must be followed to reach the node from the root. So, for example, the root node is represented with the empty word, ε, and the first descendant of the second descendant of the root is described by $2 \cdot 1$.

The acceptance of a k-ary tree by an automaton is defined by means of *runs*. A run describes the process that the automaton follows when traversing the input tree, visiting every node only once, having some internal state, and then changing this state while continuing reading the tree, according to the transition relation.

Definition 2.2 *Let* $M = (Q, \Delta, I, F)$ *be an automaton over k-ary trees. A* run *of M is a labeled k-ary tree r such that*

- $r(\varepsilon) \in I$*;*
- *for all* $w \in \{1, \ldots, k\}^*, (r(w), r(w \cdot 1), \ldots, r(w \cdot k)) \in \Delta$*.*

Such a run is called accepting *if every path contains infinitely often a final state.*

Intuitively, a run is simply a labeling of the input tree in which the root is labeled with one of the initial states, and the descendants of each node are labeled following the transition relation Δ. Such a run is accepting if this labeling is such that over every branch of the tree, the labels found are final states infinitely often. This is known as the *Büchi acceptance condition*.

Notice that there might be many different runs. The *emptiness problem* of these automata consists simply in deciding whether there exist an accepting run or not. This problem is well known to be solvable using polynomial time, on the number of states of the automaton [10, 15].

The subclass of Büchi automata in which every state is final, and hence every run is accepting, will be of special interest in the following chapters. This kind of automata are called *looping*, and are represented without the set of final states, which is implicitly the set of all states.

Definition 2.3 *The* looping automaton $M = (Q, \Delta, I)$ *is the Büchi automaton* $M' = (Q, \Delta, I, Q)$.

For this subclass of automata, runs and the emptiness problem are defined as for the Büchi automata. Notice that in this case, the emptiness problem reduces to simply deciding whether there exists a run or not, as every run is accepting.

In the next section it will be shown that automata can be used to solve other decision problems. In particular, it will be presented how looping automata assist in the solution of some decision problems in logic.

It is important to note that there are other kinds of automata, apart from the ones defined here. In particular, those which receive as input infinite words. The emptiness problem of these automata may also be applied for solving decision problems in logic [14]. An infinite word can be seen as a 1-ary infinite tree. In that sense, these automata are treated in the framework presented in this work for finding optimized methods for solving the emptiness problem.

Nonetheless, for this particular case, the constraints that will be given in Chapter 3 turn out to be too restrictive. While the framework requires conditions in the automata to improve the emptiness test, this test can always be done with a non-deterministic method using logarithmic space on the size of automata over infinite words [16], making such conditions superfluous.

All this means that the method that motivated the framework can always

be performed when using automata over infinite words. For this reason, this kind of automata will not be considered on detail in this work.

2.2 Description Logics

Description Logics [2] appear from the need of a formal way of building knowledge representations that could be used to effectively extract implicit consequences of the explicit elements in the model. Prior to these logics, a promising path for this goal was the use of *network-based structures* [4], such as *semantic networks* [9] and *frames* [7]. The intuition behind this was that by means of such structures, representation could be done in an simpler way, and reasoning would be more efficient, taking advantage of the hierarchical configurations.

The first attempts of modeling with hierarchical structures had the problem of lacking precise semantics. This meant that the behaviour of a system could differ greatly from that of another of similar shape. Description Logics give formal semantics to these structures, being a fragment of first-order logic.

As the name suggests, Description Logics are used to describe a portion of knowledge, by defining its relevant concepts. This is done, broadly, by characterizing the relationship between the concepts, and the relationship between objects and concepts obtaining, this way, a relation between the objects in the application.

As there are many ways to construct concepts, and also different ways to describe relations between them, there exist many different Description Logics. In this section, only one of these logics will be defined. This logic, called $\mathcal{ALC}$, may be seen as an example for understanding other Description Logics.

The name $\mathcal{ALC}$ stands for *attribute language with complement* and is an extension of the language $\mathcal{AL}$ as defined in [13]. In this logic, the concepts define formally the notions of the application domain. It includes also a set of binary relations called *roles*. The complete syntax is defined next.

Definition 2.4 (Syntax) *Let N_R and N_C be disjoint sets of* role names *and* concept names*, respectively. The set of $\mathcal{ALC}$-*concept terms *is defined inductively as follows:*

1. *Each concept name $A \in N_C$ is an $\mathcal{ALC}$-concept term.*

2. $\top$ *and* $\bot$ *are* $\mathcal{ALC}$*-concept terms.*

3. *If* C, D *are* $\mathcal{ALC}$*-concept terms, and* $r \in N_R$*, then the following are also* $\mathcal{ALC}$*-concept terms:*

   - $C \sqcap D, C \sqcup D, \neg C$
   - $\forall r.C, \exists r.C$

Example 2.5 *Let* `Positive` *and* `Odd` *be concept names with the intended meaning of positive and odd numbers, respectively, and* `has-factor` *be a role name with the intended meaning of the second element in the binary relation being a proper divisor of the first one*[1]*. Then,*

- `Positive`$\sqcap\neg$`Odd` *describes the notion of "even number";*
- $\neg\exists$`has-factor.Positive` *describes the notion of "1".*

The formal semantics of $\mathcal{ALC}$ is given by interpretations, which can be seen as mappings from concept terms to a specific domain. This is stated more formally in the next definition.

Definition 2.6 (Semantics) *An* interpretation $\mathcal{I}$ *consists of a non-empty* interpretation domain $\Delta^{\mathcal{I}}$ *and an interpretation function* $\cdot^{\mathcal{I}}$ *that*

- *assigns to each* $A \in N_C$ *a subset* $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$*,*
- *assigns to each* $r \in N_R$ *a binary relation* $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$*.*

The interpretation function is then inductively extended to the rest of $\mathcal{ALC}$ *concept terms as follows:*

- $\top^{\mathcal{I}} = \Delta^{\mathcal{I}}, \bot^{\mathcal{I}} = \emptyset$
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
- $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$
- $(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$

[1] In this case, divisor is understood in the following sense: a number a is a *divisor* of b if there is a natural number c such that $b = a \cdot c$. It is a *proper divisor* if, additionally, $a \neq b$.

- $(\forall r.C)^{\mathcal{I}} = \{d \in \Delta^{\mathcal{I}} \mid \forall e \in \Delta^{\mathcal{I}} : (d,e) \in r^{\mathcal{I}} \Rightarrow e \in C^{\mathcal{I}}\}$
- $(\exists r.C)^{\mathcal{I}} = \{d \in \Delta^{\mathcal{I}} \mid \exists e \in \Delta^{\mathcal{I}} : (d,e) \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\}$

Example 2.7 *A possible interpretation $\mathcal{I}$ for the concept names and role name in Example 2.5 is given by:*

- $\Delta^{\mathcal{I}} = \{1, 2, 3, \ldots\}$
- $\texttt{Positive}^{\mathcal{I}} = \Delta^{\mathcal{I}}$
- $\texttt{Odd}^{\mathcal{I}} = \{1, 3, 5, \ldots\}$
- $\texttt{has-factor}^{\mathcal{I}} = \{(i,j) \in \Delta \times \Delta \mid j < i \text{ and } j|i\}$

It is sometimes convenient to define some concept names as abbreviations of more complex concept terms. Such abbreviations are useful, for example, to improve the readability of a model, or to reduce the size of a model description. This is done with the help of a so called *terminology box (TBox).*

Definition 2.8 *If $A \in N_C$ and C is a concept term, then $A \doteq C$ is a* concept definition. *A finite set $\mathcal{T}$ of concept definitions is called* acyclic TBox *if the following conditions hold:*

1. *There exist no $A \in N_C$ and distinct concept terms C, D such that $\{A \doteq C, A \doteq D\} \subseteq \mathcal{T}$,*
2. *There are no $n \geq 1$ and concept definitions $A_1 \doteq D_1, \ldots, A_n \doteq D_n$ in $\mathcal{T}$ such that*
   - *D_i contains A_{i+1} for $1 \leq i < n$,*
   - *D_n contains A_1.*

These two conditions ensure that the elements of an acyclic TBox are really just definitions for concept terms; that is, abbreviations that can be replaced by complex concept terms.

The semantics of a TBox are given in a natural way.

Definition 2.9 *An interpretation $\mathcal{I}$ is a* model *of an acyclic TBox, $\mathcal{T}$, if for every concept definition $A \doteq C \in \mathcal{T}$, it holds that $A^{\mathcal{I}} = C^{\mathcal{I}}$.*

Sometimes it is necessary to express an inclusion relation between concepts. For example, if one is willing to express that every number divisible by an even number is itself even, there is no way of doing so using only $\mathcal{ALC}$-concept terms and acyclic TBoxes.

For that, a more general kind of terminology is needed. This leads to generalized TBoxes, which contain *generalized concept inclusion axioms*.

Definition 2.10 *If C and D are $\mathcal{ALC}$-concept terms, then the expression $C \sqsubseteq D$ is called a* generalized concept inclusion axiom (GCI). *A finite set $\mathcal{T}$ of GCIs is called a* general TBox.

An interpretation $\mathcal{I}$ is a model of a general TBox $\mathcal{T}$ if for every GCI $C \sqsubseteq D \in \mathcal{T}$, it holds that $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$.

Example 2.11 *The following GCI expresses the statement "every number divisible by an even number is even":*

$$\exists \texttt{has} - \texttt{factor}.(\texttt{Positive} \sqcap \neg \texttt{Odd}) \sqsubseteq \texttt{Positive} \sqcap \neg \texttt{Odd}.$$

It must be clear that general TBoxes are more general than acyclic TBoxes, since every concept definition $A \doteq C$ can be represented using the two GCIs $A \sqsubseteq C$ and $C \sqsubseteq A$, but not every GCI can be expressed using exclusively concept terms and concept definitions.

Note that every GCI $C \sqsubseteq D$ can be also written as $\top \sqsubseteq \neg C \sqcup D$. As every concept E term satisfies $E \sqsubseteq \top$, the mentioned GCI can be further represented as $\top \doteq \neg C \sqcup D$. With this in mind, it will be assumed that every GCI in a general TBox is of the form $\top \doteq C$. It is possible then to define, for a general TBox $\mathcal{T}$ the set $E(\mathcal{T}) = \{D \mid \top \doteq D \in \mathcal{T}\}$.

$\mathcal{ALC}$-concept terms and general TBoxes are used to formally describe the application domain. As it has been said before, these descriptions are not intended to just stay there as pictures of the current knowledge. Reasoning tasks are to be applied to them in order to extract knowledge that is implicitly defined. One of such reasoning tasks is the *satisfiability problem*.

Definition 2.12 *Let C be an $\mathcal{ALC}$-concept term, and $\mathcal{T}$ a general TBox. C is* satisfiable with respect to $\mathcal{T}$ *if there is a model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}} \neq \emptyset$.*

The satisfiability problem of $\mathcal{ALC}$-concept terms with respect to general TBoxes consists on deciding whether a given $\mathcal{ALC}$-concept term is satisfiable

with respect to a general TBox. This is useful in making some knowledge explicit, as shown in the next example.

Example 2.13 *Let $\mathcal{T}$ be the TBox containing only the GCI in Example 2.11. The fact every odd number must have only odd divisors is implicitly stated by this GCI. To make such knowledge explicit, one may try the satisfiability of the concept term* $\texttt{Odd} \sqcap \neg(\forall \texttt{has} - \texttt{factor}.\texttt{Odd})$.

This term will be unsatisfiable. Hence, every element in every model must either not be odd, or have only odd factors. This implies that every odd number has only odd divisors.

Two subproblems of this one that will be of interest are *satisfiability with respect to an acyclic TBox*, when the TBox consists only of concept definitions, and *satisfiability*, when the TBox is empty.

A way of solving the satisfiability problem with respect to general TBoxes is by means of a reduction to the emptiness problem of automata. For this, given a general TBox and a concept term, an automaton will be constructed in such a way that the language accepted by the automaton is not empty if and only if the concept is satisfiable with respect to the general TBox.

To construct such an automaton, it will be necessary to speak of sets of sub-concept terms, since they will be used as states.

Definition 2.14 *Let C be an $\mathcal{ALC}$-concept term. The* set $sub(C)$ of all sub-concept terms *of C is inductively defined by:*

- $sub(A) = \{A\}, A \in N_C$,
- $sub(C \sqcap D) = \{C \sqcap D\} \cup sub(C) \cup sub(D)$,
- $sub(C \sqcup D) = \{C \sqcup D\} \cup sub(C) \cup sub(D)$,
- $sub(\neg C) = \{\neg C\} \cup sub(C)$,
- $sub(\forall r.C) = \{\forall r.C\} \cup sub(C), sub(\exists r.C) = \{\exists r.C\} \cup sub(C)$.

If $\mathcal{C}$ is a set of $\mathcal{ALC}$-concept terms, then $sub(\mathcal{C}) = \bigcup_{C \in \mathcal{C}} sub(C)$.

If C is an $\mathcal{ALC}$-concept term, and $\mathcal{T}$ is a general TBox, then $sub_{\mathcal{T}}(C) = sub(E(\mathcal{T}) \cup \{C\})$.

In order to solve the satisfiability problem with respect to a TBox $\mathcal{T}$, it is necessary to find out if there is a model of $\mathcal{T}$ that maps the concept term C to a non-empty set. For that reason, not only the sub-concept terms of C, but also those of the elements in $\mathcal{T}$, are of interest. Hence, the need for defining $sub_{\mathcal{T}}(C)$. As only C and $\mathcal{T}$ come into consideration for solving this problem, no other concept terms are treated.

Definition 2.15 *An $\mathcal{ALC}$-concept term C is* in negation normal form *if for every $\neg D \in sub(C), D \in N_C$.*

Notice that by the way the semantics of $\mathcal{ALC}$ were defined, all the de Morgan laws apply. This means that the negations may be *pushed* inside the formula. All this implies that for every concept term, there is an equivalent one that is in negation normal form, which can be constructed in linear time with respect to the original concept.

With this in mind, without loss of generality, only concept terms in negation normal form will be considered. Hence, whenever a concept name is used in the rest of this text, it will be assumed that it has such a form. This applies also for the concept terms that define TBoxes.

The idea behind the automaton is that every run defines a model for the concept term, and that every model should be able to be translated into a run. To do this, the states of the automaton will be sets of concept terms.

Intuitively, the state in which the automaton visits a node will hold the set of sub-concept terms satisfied by the run on that node. A run is then seen as a model that has as domain the whole set of nodes, and every concept name is maped to all the nodes that have the desired concept name as element of their states. These sets of concepts, called Hintikka sets, must satisfy the following conditions.

Definition 2.16 *Let C be an $\mathcal{ALC}$-concept term, and $S \subseteq sub_{\mathcal{T}}(C)$, then S is a* Hintikka set *if and only if*

1. *if $C_1 \sqcap C_2 \in S$, then $\{C_1, C_2\} \subseteq S$*
2. *if $C_1 \sqcup C_2 \in S$, then $\{C_1, C_2\} \cap S \neq \emptyset$*
3. *$\{A, \neg A\} \not\subseteq S, A \in N_C$*

For a node to satisfy a concept of the form $\exists r.D$, this node must have an r-successor that satisfies the concept D. To simplify the process of keeping track of which successor correspond to which existential formula, the branching factor of the input trees for the automaton will be equal to the number of existential formulas in $sub_{\mathcal{T}}(C)$. Given an enumeration of these existential formulas, the i-th successor of a node, will correspond to the i-th formula.

In this way, it also becomes unnecessary to keep track of the role by which a node is successor of the other, as it is implicitly included in the existential formula. This is very useful, since runs have no labels on the edges, but only on the nodes.

To ensure that a run is a model, if a node holds the i-th existential formula, say $\exists r.D$, then its i-th successor must contain the concept D. Furthermore, for every universal formula using the role $r, \forall r.E, E$ must also belong to this successor. When the states of a node and its descendants satisfy these conditions, they form an *appropriate tuple*.

Definition 2.17 *Let C be an $\mathcal{ALC}$-concept term, and ϕ a bijective ordering function for the existential formulas, $\phi : \{1, \ldots, k\} \rightarrow \{\exists r.D \in sub_{\mathcal{T}}(C)\}$.*

A tuple $(S, S_1, \ldots, S_k)$ of Hintikka sets is appropriate *if and only if whenever $\phi(i) = \exists r.D \in S$, then $D \in S_i$ and for all formulas $\forall r.E \in S, E \in S_i$*

A tree T is a Hintikka tree *if and only if for every node $w \in \{1, \ldots, k\}^*$, the tuple $(T(w), T(w \cdot 1), \ldots, T(w \cdot k))$ is appropriate.*

A Hintikka tree corresponds then to an interpretation in the way previously described. In order to be a model, every node must satisfy also all the GCIs contained in the general TBox. These Hintikka trees are called *$\mathcal{T}$-restricted*.

Definition 2.18 *A Hintikka tree T is called* $\mathcal{T}$-restricted*if for every node $w \in \{1, \ldots, k\}^*, E(\mathcal{T}) \subseteq T(w)$.*

The next lemma relates the satisfiability problem to the existence of $\mathcal{T}$-restricted Hintikka trees [6].

Lemma 2.19 *An $\mathcal{ALC}$-concept term C is satisfiable with respect to a general TBox $\mathcal{T}$ if and only if there is a $\mathcal{T}$-restricted Hintikka tree T such that $C \in T(\varepsilon)$.*

By means of this lemma, it is possible to make an automaton M_C such that $L(M_C) \neq \emptyset$ if and only if C is satisfiable. This is easily achieved with a looping automaton whose runs are exactly the desired Hintikka trees. This way, there is a run of M_C if and only if C is satisfiable.

Definition 2.20 *Let C be an $\mathcal{ALC}$-concept term and $\mathcal{T}$ a general TBox such that $sub_{\mathcal{T}}(C)$ contains k existential formulas which are linearly ordered. Then the looping automaton $M_C = (Q, \Delta, I)$ is given by:*

- $Q = \{S \mid S \text{ is a Hintikka set with } E(\mathcal{T}) \subseteq S\}$
- $I = \{S \in Q \mid C \in S\}$
- $\Delta = \{(S, S_1, \ldots, S_k) \mid (S, S_1, \ldots, S_k) \text{ is appropriate}\}$

Notice that all the transitions of this automaton satisfy the requirement of Hintikka trees of forming appropriate tuples. Furthermore, the state at every node contains the set $E(\mathcal{T})$. This means that the runs of M_C are all $\mathcal{T}$-restricted Hintikka trees.

As the initial states all contain the concept term C, the state at the root of the run must have C as its element. Thus, the runs of the automaton M_C are exactly all those $\mathcal{T}$-restricted Hintikka trees T such that $C \in T(\varepsilon)$. Each of those runs are accepting, since M_C is a looping automaton. Hence, this automaton is the desired decision procedure.

Unfortunately, using all the restricted Hintikka sets as states makes the size of M_C exponential on the size of C. Hence, the emptiness test of M_C, which takes polynomial time on the size of M_C, yields an exponential decision procedure for satisfiability of $\mathcal{ALC}$-concept terms with respect to general TBoxes. It turns out, nonetheless, that this is an optimal decision procedure for this problem, as it is well known to be ExpTime-hard [12].

For the two subproblems defined in this section, that is, satisfiability with respect to empty and with respect to acyclic TBoxes, this optimality result is not true. In Chapter 4 it will be shown that these problems are in PSpace.

The idea for showing that will be not to change the automata-based decision procedure, but to show that these automata satisfy some conditions that allow the emptiness test to be solved using only space logarithmic on the number of states. This, with the exponential blow up of the automaton, gives in the end a PSpace result.

The next chapter defines a class of automata in which the emptiness test can be solved with a reduced use of resources, and describes the non-deterministic method for making that test using logarithmic space, proving its correctness.

Chapter 3

Optimization of emptiness test on reducible automata

This chapter introduces a class of automata for which the emptiness problem can be solved with an optimized use of resources. For these automata, the emptiness problem can be solved with a non-deterministic method, which requires logarithmic space with respect to the number of states of the automaton, depending solely on the branching factor of the input and the number of equivalence classes defined by a partition of these states. This is an important improvement from the polynomial time bound given for general automata [15].

One idea for making the emptiness test more memory efficient consists on stopping the verification for existence of a run when reaching particular states. If there is a set of states for which it is always possible to construct a labeled tree with all the nodes being final states, then once an element of this set is reached, the decision procedure can stop checking that section of the tree for the existence of an accepting run. Starting from that node, it is known already that the run could be extended, and that it will satisfy Büchi's accepting condition. That is the idea behind the following definition.

Definition 3.1 *Let $M = (Q, \Delta, I, F)$ be a Büchi automaton over m-ary trees, and $Q_0 \subseteq F$. M is called* Q_0-looping *if for every $q \in Q_0$ there exist $q_1, \ldots, q_m \in Q_0$ such that $(q, q_1, \ldots, q_m) \in \Delta$.*

It is important to notice that from this definition it follows that for every $q \in Q_0$ there exists a tree whose root is labeled with q, such that

every node is labeled with an element of Q_0 and every transition from a parent to its descendants satisfies the transition relation. This tree can be seen as a section of a run. As all the elements in Q_0 are final states, this section satisfies additionally the acceptance condition.

This tree may not be unique. For that reason, although to be able to stop the search for a section of run the only need is to find one of such trees, the set $\mathcal{T}_q$ containing all of them is defined.

Definition 3.2 *For every $q \in Q_0$, define $\mathcal{T}_q$ as the set of all the trees T_q that satisfy the following conditions:*

1. *$T_q(\varepsilon) = q$, and*
2. *for every $w \in \{1, \ldots, m\}^*, T_q(w) \in Q_0$ and $(T_q(w), T_q(w \cdot 1), \ldots, T_q(w \cdot m)) \in \Delta$.*

As it was argued before, for every $q \in Q_0, \mathcal{T}_q$ is not empty. This will become important to set a tighter space bound on the decision procedure for the emptiness problem of a class of automata over m-ary trees. The idea consists in a procedure that checks only until a state in Q_0 is found, and forgetting the section of the tree under this node. From that point on, a tree in $\mathcal{T}_q$ can be used to complete the accepting run.

First, a simple case, on a very restricted class of automata, will be treated. This class will be used afterwards find more relaxed conditions, defining this way a more general class.

3.1 Reducible automata

Suppose that it was possible to decide the existence of a run by checking the tree it constructs only up to certain depth. In this case, one can accomplish the test by doing a depth-first search. This kind of search is impossible in the general case, since the branches in a run are infinite, and then the rest of the branches could never be checked. Notice that automata are local mechanisms, in the sense that, in a run, each branch is independent of the others. For this reason, there is no need to keep in memory the whole tree, but only the branch being verified at the time, and the backtracking information necessary for covering every branch.

Sufficient backtracking information for traversing a tree is the set of descendants of each visited node, along with a pointer stating which branch

has been visited. Under certain conditions on the branching factor, and the maximum depth necessary to decide the existence of the run, this yields a procedure requiring only logarithmic space on the number of states of the automaton.

One simple idea to be able to perform such a search for run of an automaton is to find a partition of the states, and an ordering of the classes, such that every transition leads to a lower class, except for the smallest class that will be called Q_0. In this case, every run will have the property that all the nodes with sufficient depth, must be labeled with a state in Q_0. If this automaton happens to be also Q_0-looping, then the emptiness test can be reduced to checking the existence of a run up to a certain depth. As there may be many possible transitions to apply at every node, this procedure will be non-deterministic.

All this is now stated in a more formal way.

Definition 3.3 *An automaton $M = (Q, \Delta, I, F)$ is called* k-reducible *if there exists a partition $Q_0, Q_1, \ldots, Q_k$ of Q such that M is Q_0-looping, and for every $(q, q_1, \ldots, q_m) \in \Delta$ if $q \in Q_n$, then $q_i \in Q_0 \cup \bigcup_{j=1}^{n-1} Q_j$ for all $1 \leq i \leq m$.*

This definition implies that every node of every run will have a state in a lower class than each of its predecessors. For that reason, if a state is in a class different from Q_0, its depth cannot be greater than the number of class the state at the root is in. This is stated in the following theorem.

Theorem 3.4 *Let $M = (Q, \Delta, I, F)$ be a k-reducible automaton. Then every run r of M is such that if $r(w) \notin Q_0$, then $|w| \leq k$.*

Proof. Let r be a run. As M is k-reducible, if $r(w) \in Q_n$ for some word w and some $0 \leq n \leq k$, then for all $1 \leq i \leq m, r(w \cdot i) \in Q_0 \cup \bigcup_{j=1}^{n-1} Q_j$. Thus, if $r(\varepsilon) \in Q_{n_0}$, then for every word $w, r(w)$ is an element of $Q_{\hat{n}}$, where $\hat{n} = max\{0, n_0 - |w|\}$. As $n_0 \leq k$, every word w with $|w| > k$ is such that $r(w) \in Q_0$. ■

Note that this theorem implies that every run of a k-reducible automaton is accepting, since $Q_0 \subseteq F$ and only finitely many nodes of the run are labeled with a state in a different class.

With this in mind, one can produce a non-deterministic decision procedure that requires only logarithmic space on the number of states of the automaton, as stated in the next corollary.

Corollary 3.5 *The emptiness problem can be non-deterministically solved for a k-reducible automaton over m-ary trees, $M = (Q, \Delta, I, F)$, using space $O(log(|Q|) \cdot m \cdot k)$.*

Proof. The non-deterministic procedure defined in Algorithm 3.1 performs the emptiness test using the desired space.

The idea consists simply in guessing a run of the automaton by a depth-first search.

As k is the number of classes and every transition reduced the class, the states of all nodes at a depth greater or equal to k in any possible run will be in the class Q_0. As for every element in Q_0 there is a transition, and $Q_0 \subseteq F$ (by Definition 3.1), it is sufficient to check the existence of the run to depth k.

To avoid the exponential blow-up of keeping the complete tree, only one branch will be checked at a time, keeping additionally the needed backtracking information in memory. This information is stored in two words: TQ keeps record of the transitions that have been selected in the predecessor nodes, while TB stores the path of the next descendant that must be checked. Thus, $TQ \in (Q^m)^*$, and $TB \in \{1, \ldots, m\}^*$, where m is the branching number. These words are sufficient to check also the depth of the run, which is their length.

For example, look at Figure 3.1. It shows the state of the algorithm inside an iteration of the *while* loop starting at line 7. The labels over the nodes are the states given by the guessed transitions, and the circled nodes represent the path to the node that the algorithm is verifying at the time. On the right side, the values of TQ and TB are given by the concatenation from top to bottom of the symbols in the respective column.

Suppose first that $L(M) \neq \emptyset$, then there is an accepting run r of M; this means, by Theorem 3.4, that if $r(w) \notin Q_0$, then $|w| \leq k$. This run can be used to construct a run of the algorithm that returns "not empty" as follows.

For the first two steps, guess the initial state $r(\epsilon)$ and the transition $(r(\epsilon), r(1), \ldots, r(m))$, which must exist, since r is a run. Thus, at the beginning of the *while* loop, $TQ = (r(1), \ldots, r(m))$ and $TB = 1$.

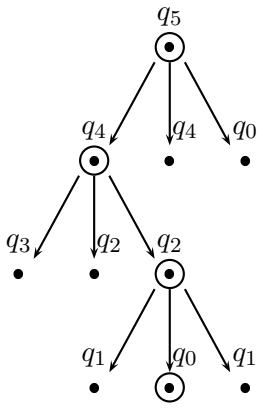
1: Guess an initial state $q \in I$
2: Guess a transition $(q, q_1, \ldots, q_m) \in \Delta$.
3: **if** there is no such transition **then**
4: **return** "empty"
5: **end if**
6: Set $TQ := (q_1, \ldots, q_m), TB = 1$.
7: **while** $TB \neq \varepsilon$ **do**
8: Let $w \cdot (q_1, \ldots, q_m) = TQ$, and $v \cdot n = TB$
9: **if** $n > m$ **then** {all the descendants have been processed}
10: $TQ := w, TB := v$ {backtrack}
11: **else if** $q_n \in Q_0$ **then** {lowest class reached}
12: $TQ := w \cdot (q_1, \ldots, q_m), TB := v \cdot (n+1)$ {continue}
13: **else**
14: Guess a transition $(q_n, q'_1, \ldots, q'_m) \in \Delta$.
15: **if** there is no such transition **then**
16: **return** "empty"
17: **end if**
18: Set $TQ := w \cdot (q_1, \ldots, q_m) \cdot (q'_1, \ldots, q'_m), TB := v \cdot (n+1) \cdot 1$
19: **end if**
20: **end while**
21: **return** "not empty"

Algorithm 3.1: Non-deterministically verify the existence of a run of the automaton, by traversing one branch at a time, and only until an element of Q_0 is reached.

Notice that if $TB = n_1 n_2 \ldots n_l$, then the algorithm is checking for a transition in the node given by $(n_1 - 1)(n_2 - 1) \ldots n_l$, except when $n_l > m$, but this case is treated by the first condition, on line 9. Call this word $next(TB)$. As an example, refer to the right-most column of Figure 3.1, showing the relation between TB and $next(TB)$.

If $n_l > m$ then the condition in line 9 is fulfilled and the algorithm enters the loop again with a shorter TB. If that is not the case and $r(next(TB))$ is in Q_0, as in line 11, the algorithm enters the loop again, also with a shorter TB and without failing.

If none of the previous is the case, the algorithm may guess the transition $(r(next(TB)), r(next(TB)) \cdot 1), \ldots, r(next(TB)) \cdot m))$. This transition exists, as r is a run, and hence the algorithm does not return "empty" at this point. Afterwards, the *while* loop is reentered with the information of checking the



TQ	TB	$next(TB)$
(q_4, q_4, q_0)	2	1
(q_3, q_2, q_2)	4	3
(q_1, q_0, q_1)	2	2

Figure 3.1: A segment of a run of Algorithm 3.1 for checking emptiness of a 5-reducible automaton over 3-ary trees, where $Q_i = \{q_i\}$ for $i \in \{1, \ldots, 5\}$. On the right, are the values of the variables TQ, TB, and $next(TB)$ when verifying the existence of a run on the branch marked with circles.

first descendant, defined by this transition.

Thus, every step can be performed and will not leave the *while* loop by returning "empty", which means that the algorithm will return in the end "not empty".

Conversely, if the algorithm returns "not empty", then at lines 1 and 2 of the algorithm, it must have guessed an initial state q and a transition $(q, q_0, \ldots, q_m) \in \Delta$. This transition is used to define the root of a run $r(\varepsilon) = q$, and its descendants $r(i) = q_i, i \in \{1, \ldots, m\}$.

Let now $TQ = w \cdot (q_1, \ldots, q_m), TB = v \cdot n$ be the ones given in an iteration of the *while* loop in line 8. As it was said before, the algorithm is checking for a transition in the node given by $next(TB)$. If $n > m, next(TB)$ defines no node of the run, and the algorithm backtracks, to make the transitions in the next branch. Also, if $q_n \in Q_0$, then as the automaton is Q_0-looping, there is a $T \in \mathcal{T}_{q_m}$, so defining $r(next(TB) \cdot u) = T(u)$ for all $u \in \{1, \ldots, m\}^*$ defines all the branches of a run that start with $next(TB)$.

Finally, if none of the previous is the case, then the algorithm guesses a transition $(q_n, q'_1, \ldots, q'_m) \in \Delta$. Setting $r(next(TB) \cdot i) = q'_i$ gives a correct transition. Note that, as the algorithm runs in order, the transitions of every

node on the tree are defined, either directly by a non-deterministic guess, or by using the properties of belonging to Q_0. Hence, r is a run. As every run of a k-reducible automaton is accepting, $L(M) \neq \emptyset$.

Thus, the algorithm is a correct decision procedure for the emptiness problem of k-reducible automata.

The algorithm stores the backtracking information. This means that, at each step, it stores the set of all the transitions made in the path from the root of the run to the current node to be checked, and a representation of such a path by means of the successor nodes that have been chosen.

Each transition is stored as a m-tuple of states. Furthermore, as the algorithm checks until an element in Q_0 is found, and every transition reduces the class in the node, the maximum length of such a path is k. Hence, at every step, at most k tuples of m states, and k additional numbers representing the path, are stored in memory.

Note that each state of the automaton can be represented using only space $O(log(|Q|))$. This is easily achieved giving an enumeration of the states, and using then a binary representation of the corresponding numbers. This yields the desired representation in logarithmic space. Hence the space required in total by this algorithm is $O(log(|Q|) \cdot m \cdot k)$. ■

The conditions required for this corollary, that is, M being a k-reducible automaton, are very strong. It is in general very difficult to find a partition of the states in which every transition diminishes the class.

However, it is possible to define a more general class of automata for which there is an emptiness-equivalent k-reducible automaton that can be constructed in constant space. This means that the emptiness problem can be solved for the automata in this general class within the same space bound stated in Corollary 3.5.

These automata, called weakly-k-reducible automata, are defined in the next section.

3.2 Weakly reducible automata

The idea behind weakly reducible automata is not to force every transition to reduce the class, as done with reducible automata, but to ensure that, if there is a transition, then there is also one that reduces the class. Of course, this has to be done carefully, since changing the states of the descendant

nodes may cause that no further transitions are possible in a run, as shown in the next example.

Example 3.6 *Let* $M = (\{q_0, q_1\}, \Delta = \{(q_1, q_1, q_1), (q_1, q_0, q_0)\}, \{q_1\})$ *be a looping automaton, with the partition given by* $Q_1 = \{q_1\}$*, and* $Q_0 = \{q_0\}$*. Then, for every transition in* Δ*, there is one that reduces the class. Nonetheless, the only possible run of this automaton is the binary tree where all the nodes are labeled as* q_1*, and changing any transition to the reducing one, that is, changing the labels of the descendants of a node to* q_0 *would make it impossible to find another run.*

To avoid the problem shown in Example 3.6, some further restriction have to be fulfilled by the automata, to ensure the possibility of reducing the class and still obtaining a run. The next definition states adequate conditions for being able to do such reductions.

Definition 3.7 *An automaton* $M = (Q, \Delta, I, F)$ *is called* weakly-k reducible *if there exists a partition* $Q_0, Q_1, \ldots, Q_k$ *of* Q *such that* M *is* Q_0*-looping, and for every* $q \in Q$ *there exist a function* $f_q : Q \to Q$ *such that the following hold:*

1. *if* $(q, q_1, \ldots, q_m) \in \Delta$*, then* $(q, f_q(q_1), \ldots, f_q(q_m)) \in \Delta$*, and if* $q \in Q_n$*, then for every* $i \in \{1, \ldots, m\}$*,* $f_q(q_i) \in Q_0 \cup \bigcup_{j=1}^{n-1} Q_j$
2. *if* $(q', q_1, \ldots, q_m) \in \Delta$*, then* $(f_q(q'), f_q(q_1), \ldots, f_q(q_m)) \in \Delta$

As it was said before, for every weakly-k-reducible automaton, there is an emptiness equivalent k-reducible automaton. This can be obtained by simply removing all the transitions that are not reducing. The new automaton generated this way is, obviously, not larger than the original one.

Definition 3.8 *Let* $M = (Q, \Delta, I, F)$ *be a weakly-*k*-reducible automaton. The* reduced automaton M_R *of* M *is the automaton* $M_R = (Q, \Delta', I, F)$*, where*

$$\Delta' = \{(q, q_1, \ldots, q_m) \in \Delta \mid \text{if } q \in Q_n, \text{ then } q_i \in Q_0 \cup \bigcup_{j=1}^{n-1} Q_j, 1 \leq i \leq m\}.$$

Note that, by definition, the reduced automaton M_R is k-reducible, since for all the transitions in this automaton, the class of the descendant nodes is strictly smaller than the class of their parent.

Every weakly-k-reducible automaton is emptiness equivalent to its reduced automaton, as stated in the next theorem.

Theorem 3.9 *Let $M = (Q, \Delta, I, F)$ be a weakly-k-reducible automaton, then $L(M) \neq \emptyset$ if and only if $L(M_R) \neq \emptyset$.*

Proof. As M_R is a subautomaton of M, the "if" direction is trivial, so just the "only if" direction will be proved.

Let t be an accepting run of M (such a run exists, as it was assumed that $L(M) \neq \emptyset$). This accepting run is transformed into one of M_R by means of Algorithm 3.2.

```
1:  for all w ∈ {1,...,m}* with |w| ≤ k do
2:    if t(w) ∈ Q_0 then
3:      for all v ∈ {1,...,m}^+ do
4:        Set t̂(w · v) := T_{t(w)}(v), where T_{t(w)} ∈ 𝒯_{t(w)}
5:      end for
6:    else
7:      for all v ∈ {1,...,m}^+ do
8:        Set t̂(w · v) := f_{t(w)}(t(w · v))
9:      end for
10:   end if
11:   Set t := t̂
12: end for
13: Set r := t
14: return r
```

Algorithm 3.2: Construct a run of M_R given a run of M.

The "for" loop in that algorithm is done in a way that shorter words are processed before longer ones.

Notice first that the "for" loop starting in line 1 is invariant with respect to runs of M; that is, if t is a run of M at the beginning of one of such iterations, then $\hat{t}$ will also be a run of M at the end of the same iteration. This is shown next.

Suppose that t is a run, and let w be the selected node in the iteration. If $t(w) \in Q_0$, then the subtree of $\hat{t}$ obtained by setting the root to w satisfies

the transition relation Δ, by the way $T_{t(w)}$ was defined. The rest of the tree remains unchanged, and hence satisfies also the same transition relation. As $\hat{t}(\varepsilon) = t(\varepsilon)$, and $\hat{t}(w) = t(w), \hat{t}$ is a run of M.

On the other hand, if $t(w) \notin Q_0$, then the first part of Condition 1 ensures that $(\hat{t}(w), \hat{t}(w \cdot 1), \ldots, \hat{t}(w \cdot m)) \in \Delta$. By Condition 2, for all $v \in \{1, \ldots, m\}^+, (\hat{t}(w \cdot v), \hat{t}(w \cdot v \cdot 1), \ldots, \hat{t}(w \cdot v \cdot m)) \in \Delta$. As the rest of the tree remains unchanged, $\hat{t}$ is a run of M.

Thus, at every iteration, t is a run of M. To show that r is a run of M_R, the only thing left is to see that at every transition, the class in which the descendant states are has a lower numbering than the class of the parent state.

The second part of Condition 1 ensures that if a state is in a class greater than 0, all its descendants will be in a smaller class. Thus, r is a run of M_R.

As it has been pointed out before, every run of a k-reducible automaton is accepting. Thus r is an accepting run of M_R. ■

As the k-reducible automaton can be constructed in space linear on the size of the original automaton, using the reduced automaton yields a decision procedure for emptiness of weakly-k-reducible automata, with the same space bound that was stated in Corollary 3.5.

Corollary 3.10 *The emptiness problem for $M = (Q, \Delta, I, F)$, a weakly-k reducible automaton over m-ary trees, can be non-deterministically solved using space $O(log(|Q|) \cdot m \cdot k)$.*

Proof. By means of Theorem 3.9, the algorithm presented in the proof of Corollary 3.5 can be used over the k-reducible automaton M_R. Furthermore, as M_R differs from M only in that the transitions that are not reduced do not appear in the former automaton, M_R can also be constructed *on the fly* during the execution of the algorithm.

This is done by simply verifying, whenever a transition is guessed, that it is reducing. This verification takes constant space, and hence does not affect the given space bound. ■

This corollary yields an optimized space bound for verifying if the language accepted by a weakly-n-reducible automaton is empty or not. This bound states, basically, that the emptiness test can be performed non-

deterministically, using space logarithmic in the number of states of the automaton, given some minor conditions.

The class of automata here defined might look as a very limited one. As a run in a n-reducible automaton must reach final states everywhere, except for a finite section of the tree, their computational power is at most that of looping automata. Nonetheless, this power is sufficient for solving interesting problems, for example, in logic.

In Chapter 4, the bounds given by Corollaries 3.5 and 3.10 will be used to give PSpace bounds on the satisfiability problem of $\mathcal{ALC}$ with respect to empty and acyclic TBoxes using the automata decision procedure defined in Section 2.2. This serves as example of the utility of reducible automata.

Chapter 4

$\mathcal{ALC}$ satisfiability

As an example of an application of Corollaries 3.5 and 3.10, it will be shown in this chapter that the satisfiability problem for $\mathcal{ALC}$ concept terms with respect to empty and acyclic TBoxes can be solved using polynomial space on the size of the formula. This will be done using the automaton M_C defined in Section 2.2.

One should notice first that the size of each of the states in the automaton M_C is polynomial on the length of the concept term C. Furthermore, as the branching factor of the input tree is equal to the number of existential formulas in $sub_{\mathcal{T}}(C)$, this is linear on the length of C given the TBox $\mathcal{T}$. Along with Corollaries 3.5 and 3.10 all this implies that, if M_C is n-reducible or weakly-n-reducible, for an n polynomial in the length of C, then this decision procedure needs in the end only polynomial space in the length of C.

To check reducibility of the automaton, one must find an appropriate partition satisfying the conditions of Definition 3.3 or 3.7. A fast search reveals that the *natural* partitions for M_C do not, in general, satisfy such requirements. This comes to no surprise, since satisfiability with respect to general TBoxes is known to be a ExpTime-hard problem [12]. Nonetheless, there are particular cases in which such a partition can be found. Empty TBoxes, and acyclic TBoxes are two of those, as will be shown in the following sections.

4.1 Satisfiability with respect to empty TBoxes

Recall that an interpretation $\mathcal{I}$ is a model of a TBox $\mathcal{T}$ if for every GCI $C \sqsubseteq D \in \mathcal{T}$, it is the case that $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Hence, when the TBox has the property of being empty, every interpretation is a model for it. This means that the satisfiability problem in this case consists only in finding, for an $\mathcal{ALC}$-concept term C, an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$.

In the sense of Lemma 2.19, this means that the existence of any Hintikka tree with $C \in T(\varepsilon)$ is enough to verify that C is satisfiable. As the TBox is empty, the runs of the automaton M_C yield all such trees, without any other restriction. Hence, the same automaton can be used to prove a polynomial space bound if it can be shown that it is weakly-n-reducible in this particular case.

Notice that, as the algorithm described in the proof of Corollary 3.5 is non-deterministic, using this method for making a logarithmic-space test over the exponentially large automaton would lead in the end to a NPSpace result. Savitch's Theorem [11] can then be used to show that this problem is in PSpace.

It will be shown that M_C is a weakly-n-reducible automaton, for some n, when the TBox is empty. The partition used to prove this will be based on the maximum number of nested existential or universal role restrictions, which will be called *role depth*.

Definition 4.1 *Let C be an $\mathcal{ALC}$-concept term. The* role depth *of C, $rd(C)$ is inductively defined as follows:*

- $rd(A) = rd(\neg A) = 0, A \in N_C$,
- $rd(C \sqcup D) = rd(C \sqcap D) = \max\{rd(C), rd(D)\}$,
- $rd(\forall r.C) = rd(C) + 1$,
- $rd(\exists r.C) = rd(C) + 1$.

If $\mathcal{C}$ is a set of $\mathcal{ALC}$-concept terms, then $rd(\mathcal{C}) = \max_{C \in \mathcal{C}} rd(C)$.

It is now possible to define the set of sub-concept terms, $sub_n(C)$, with a role depth smaller than n, except for the case when n equals 0. This sets will be used to define the functions that reduce the equivalence class, leading this way to weakly reducible automata.

Definition 4.2 *Let C be a $\mathcal{ALC}$-concept term, and $n \geq 0$. Define the set $sub_n(C) = \{D \in sub(C) \mid rd(D) \leq \max\{0, n-1\}\}$.*

As it was said before, these sets will be used to define the functions that reduce the equivalence class on a given transition. The function will consist, for a given state q, in the intersection of the argument, which is a Hintikka set, with $sub_n(C)$, where n is the role depth of q. The idea of these functions is that, when applying them to the descendants of a node, their role depth will be strictly smaller than that of the parent. If the equivalence classes are defined by the role depth, this would reduce the class.

For this functions to be adequate in the sense of Definitions 3.3 and 3.7, it is necessary to ensure first that when applying the function to a state, the result will also be a state. This is stated properly in the next lemma.

Lemma 4.3 *Let C be a $\mathcal{ALC}$-concept term, S a Hintikka set, and $n \geq 0$. Then $sub_n(C) \cap S$ is also a Hintikka set.*

Proof. First note that, as S is a Hintikka set, then for every concept name $A \in N_C, \{A, \neg A\} \not\subseteq S$. Hence, $A \in N_C, \{A, \neg A\} \not\subseteq sub_n(C) \cap S$.

By the way the role depth was defined, $rd(C_i) \leq rd(C_1 \sqcup C_2)$ and $rd(C_i) \leq rd(C_1 \sqcap C_2)$, for $i \in \{1, 2\}$.

Now, if $C_1 \sqcap C_2 \in sub_n(C) \cap S$, then $rd(C_i) \leq rd(C_1 \sqcap C_2) \leq \max\{0, n-1\}$, and $\{C_1, C_2\} \subseteq S$. Thus, $\{C_1, C_2\} \subseteq sub_n(C) \cap S$.

Analogously, if $C_1 \sqcup C_2 \in sub_n(C) \cap S$, then $rd(C_i) \leq rd(C_1 \sqcup C_2) \leq \max\{0, n-1\}$, and $\{C_1, C_2\} \cap S \neq \emptyset$. Thus, either C_1 or C_2 is an element of S. Suppose, without loss of generality, that $C_1 \in S$. But then, this entails that $\{C_1, C_2\} \cap sub_n(C) \cap S \neq \emptyset$. ■

The following lemmas show that, given a partition in which each class is defined by the role depth of its members, the functions $f_S(T) = T \cap sub_{rd(S)}(C)$, for every Hintikka set $S \in Q$, have the correct properties for weak reducibility of the automaton.

Lemma 4.4 *Let $S, S_1, \ldots, S_k$ be Hintikka sets such that $rd(S) = n$. If the tuple $(S, S_1, \ldots, S_k)$ is appropriate, then $(S, sub_n(C) \cap S_1, \ldots, sub_n(C) \cap S_k)$ is appropriate.*

Furthermore, for all $1 \leq i \leq k, rd(sub_n(C) \cap S_i) \leq \max\{0, n-1\}$.

Proof. Suppose that $\phi(i) = \exists r.D \in S$. Then, as $(S, S_1, \ldots, S_k)$ is appropriate, $D \in S_i$, and for all formulas $\forall r.E \in S, E \in S_i$. But $rd(D) < rd(\exists r.D) \leq rd(S) = n$, and $rd(E) < rd(\forall r.E) \leq rd(S) = n$.

Thus, $D \in sub_n(C) \cap S_i$, and for all $\forall r.E \in S, E \in sub_n(C) \cap S_i$.

The second part is trivial, as for all $D \in sub_n(C)$, it is the case that $rd(D) \leq \max\{0, n-1\}$. ∎

Lemma 4.5 *Let $S, S_1, \ldots, S_k$ be Hintikka sets, and $n \geq 0$. If $(S, S_1, \ldots, S_k)$ is appropriate, then $(sub_n(C) \cap S, sub_n(C) \cap S_1, \ldots, sub_n(C) \cap S_k)$ is also appropriate.*

Proof. If $\phi(i) = \exists r.D \in sub_n(C) \cap S$, then $D \in S_i$ and $rd(D) < rd(\exists r.D) \leq \max\{0, n-1\}$, hence $D \in sub_n(C) \cap S_i$.

If there is a formula $\forall r.E \in sub_n(C) \cap S$, then $E \in S_i$ and again $rd(E)) < rd(\forall r.E) \leq \max\{0, n-1\}$, and so $E \in sub_n(C) \cap S_i$. ∎

The previous lemmas show that the functions defined in the way stated before satisfy the conditions of the definition of weakly reducible automata. This leads to the following theorem.

Theorem 4.6 *Let C be an $\mathcal{ALC}$-concept term, and $n = rd(C)$. Then, M_C is weakly-n-reducible.*

Proof. Define the classes $Q_i = \{S \in Q \mid rd(S) = i\}$, for $0 \leq i \leq n$. And define for every $S \in Q$ the function $f_S : Q \to Q$ as $f_S(T) = sub_{rd(S)}(C) \cap T$.

Lemmas 4.4 and 4.5 ensure that this partition, with the given set of functions satisfy Conditions 1 and 2 of Definition 3.7. Hence, the only remaining part is to show that M_C is Q_0-looping.

Let S in Q_0, then $rd(S) = 0$. This means that there are no existential restrictions in S, hence the $(k+1)$-tuple $(S, \ldots, S)$ is appropriate, and thus, an element of Δ. As M_C is a looping automaton, $Q_0 \subseteq Q = F$. Hence, M_C is Q_0-looping. ∎

Corollary 4.7 *$\mathcal{ALC}$-satisfiability is in PSpace*

Proof. Given an $\mathcal{ALC}$-concept term C, construct the automaton M_C which is exponential in the size of C. As M_C is weakly-$rd(C)$-reducible, the

emptiness test for it can be solved using space $O(log(|Q|) \cdot k \cdot rd(C))$, where k is the number of existential formulas in C. As $log(|Q|)$ is polynomial in the size of C, and k and $rd(C)$ are linear on the length of C, in total only polynomial space is needed. ■

In this section it was shown that, whenever the TBox is empty, one can use the same decision procedure as in the case of general TBoxes, to solve the emptiness problem. The emptiness of the TBox, nonetheless, adds interesting properties to the automaton, that allow it to be tested for emptiness with an optimized use of resources. This, in the end, leads to a PSpace procedure for empty TBoxes, as compared with the ExpTime method on the general case.

Notice that the partition made on the automaton M_C by means of the role depth can also be defined in the presence of a general TBox $\mathcal{T}$. The difference would be that, as every Hintikka set must contain the set $E(\mathcal{T})$, the minimum role depth of each of those sets will be $n = rd(E(\mathcal{T}))$. Apart from these difference, whenever a Hintikka set has a role depth greater than $m > n$, and there is a transition, the same function defined by the intersection with the set of sub-concept terms with role depth strictly smaller than m will lead to a descent of the equivalent class. Then, if Q_0 is defined to be the set of all Hintikka sets having a role depth of n, for every transition there would be another that decreases the class, except when the parent is already in Q_0.

All this, along with the fact that M_C is a looping automaton, might lead to think that a partition has been found that makes M_C weakly reducible. But on the other hand, it has been said several times that the satisfiability problem with respect to general TBoxes is in ExpTime. The problem here relies in that it has not been checked whether the automaton is Q_0-looping or not.

The definition of Q_0-looping automata requires that, for every state in Q_0, there must always be a transition that keeps all the descendants in Q_0. In the general case, M_C does not satisfy this property, and even for the cases where it does, verifying it takes polynomial time on the number of states. Hence, doing such a verification would lead immediately to an ExpTime algorithm on the size of the original concept term.

The next section shows another example of the application of weakly-n-reducible automata for optimizing the satisfiability test of concept terms with respect to acyclic TBoxes. For this new application it will be neces-

sary to define a new automata-based decision procedure, that uses special properties of acyclic TBoxes.

4.2 Satisfiability with respect to acyclic TBoxes

Another problem, more general than the one studied in the previous section, will be treated now. This problem corresponds to satisfiability with respect to acyclic TBoxes, that is, TBoxes where all the elements are concept definitions. It is well known that this problem is also in PSpace [5].

The first idea to prove this would be to proceed as in the case of empty TBoxes; in other words, to find an appropriate partition of the automaton M_C. Unfortunately, the *natural* partitions, like the ones defined by the role depth, or the length of the formula, do not work for this purpose.

Recall that a concept definition is of the form $A \doteq C$, where $A \in N_C$ and C is a concept term. If the concept names occurring on the left-hand side of such a concept definition are called *defined concepts* and all the others are called *primitive concepts*, then the conditions on acyclicity on the TBox ensure that the concepts on the right side can always be expanded in a way that only primitive concept names occur on it. This means that defined concept names are just macros, or abbreviations, of the concept terms that define them.

If the automaton M_C is used, then the two concept terms $\neg A \sqcup C$ and $A \sqcup \neg C$ will appear in every state, and thus, in each label of the nodes of the run. As all these states are Hintikka sets, this means that every state will either contain $\{A, C\}$, or $\{\neg A, \neg C\}$.

This is a waste of space in two senses: on one hand, every defined concept name is explicitly stated, either in a positive or in a negative form, in every node of each run; on the other, both the defined term, and its expansion as a concept term appear together.

This removes all the sense of having definitions to abbreviate more complex concept terms. It would be the same to expand the formula by substituting every defined concept name by its definition, and then solve the problem as in the previous section, given that the remaining TBox is empty. Nonetheless, this expansion may lead to a concept term exponentially large on the size of the original concept term, getting this way a suboptimal procedure.

There is another way of solving the problem. It consists in basically

working with the defined concept names, and only expand them when it is necessary; that is, when the concept name appears by itself either in a positive or in a negative form. This way, the expansion of defined terms is done in a lazy manner, only when it is absolutely necessary to know the concept terms defined by them.

For doing this, it will be necessary to construct another automaton. The definitions required for this construction are now presented.

Definition 4.8 *Let S be a Hintikka set and $\mathcal{T}$ an acyclic TBox. S is called* $\mathcal{T}$-expandable *if for every $A \doteq C \in \mathcal{I}$, the following conditions hold:*

- *if $A \in S$, then $C \in S$,*
- *if $\neg A \in S$, then $\neg C \in S$,*

A Hintikka tree is $\mathcal{T}$-expandable *if every node is $\mathcal{T}$-expandable.*

Lemma 4.9 *An $\mathcal{ALC}$-concept term C is satisfiable with respect to an acyclic TBox $\mathcal{T}$ if and only if there is a $\mathcal{T}$-expandable Hintikka tree T such that $C \in T(\varepsilon)$.*

Proof. Suppose first that there is a $\mathcal{T}$-expandable Hintikka tree T with $C \in T(\varepsilon)$. Let $A_1 \doteq C_1, \ldots, A_n \doteq C_n$ be all the concept definitions in $\mathcal{T}$, ordered in such a way that if C_i depends on an A_j, then $j < i$. This is always possible, since $\mathcal{T}$ is acyclic. It is possible to make a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of the TBox as follows. Define first the *primitive interpretation* $\mathcal{I}_0 = (\{1, \ldots, k\}^*, \cdot^{\mathcal{I}_0})$ by mapping:

- $r^{\mathcal{I}_0} = \{(w, w \cdot i) \mid \phi(i) = \exists r.D \in T(w)\}$
- $A^{\mathcal{I}_0} = \{w \mid A \in T(w)\}$, where A is a *primitive* concept name.

For every $i \in \{1, \ldots, n\}$, extend the interpretation $\mathcal{I}_{i-1}$ by including an interpretation for the concept name A_i as follows:

- $r^{\mathcal{I}_i} = r^{\mathcal{I}_{i-1}}$,
- $A^{\mathcal{I}_i} = A^{\mathcal{I}_{i-1}}$ if A is a primitive concept name, or $A = A_j$ for some $j < i$,
- $A_i^{\mathcal{I}_i} = \{w \mid w \in (C_i)^{\mathcal{I}_{i-1}}\}$.

Notice that this sequence is well defined, because every C_i depends only on concept names for which the interpretation had been previously defined. Hence, when defining the interpretation for A_i, the interpretation of C_i is already known. The interpretation $\mathcal{I} = \mathcal{I}_n$ is a model for $\mathcal{T}$.

It is now shown by induction that, for every concept term D using only primitive concept names, or concept names in the set $\{A_1, \ldots, A_i\}$, and every node w, if $D \in T(w)$, then $w \in D^{\mathcal{I}_i}$.

If $i = 0$, then the result is trivial by the definition of $\mathcal{I}_0$. Suppose now that it is true for some i, the result will be shown for $i + 1$ by induction on the shape of D. Suppose that $D \in T(w)$.

If D is a concept name, then its interpretation was already defined in $\mathcal{I}_i$, or $D = A_{i+1}$. In the first case, the induction hypothesis yields the result. In the second case, as T is $\mathcal{T}$-expandable, it is also the case that $C_{i+1} \in T(w)$, and C_{i+1} uses only primitive, or defined concept names with index at most i, so the induction hypothesis again yields $w \in (C_{i+1})^{\mathcal{I}_i}$. And hence, $w \in A_{i+1}^{\mathcal{I}_{i+1}}$. The induction step depends on the outermost constructor of D:

$\neg D_1$ D_1 is then a concept name. If its interpretation was defined already for $\mathcal{I}_i$, then the induction yields the result. Otherwise, $D = \neg A_{i+1}$, and hence $\neg C_{i+1} \in T(w)$. Again, as C_{i+1} depends only on previously defined concept names, $w \in D^{\mathcal{I}_{i+1}}$.

$D_1 \sqcap D_2$ As T is a Hintikka tree, $\{D_1, D_2\} \subseteq T(w)$. By induction, $w \in (D_1)^{\mathcal{I}_{i+1}}$, and $w \in (D_2)^{\mathcal{I}_{i+1}}$. Thus, $w \in (D_1)^{\mathcal{I}_{i+1}} \cap (D_2)^{\mathcal{I}_{i+1}} = D^{\mathcal{I}_{i+1}}$

$D_1 \sqcup D_2$ Analogous to $D_1 \sqcap D_2$.

$\exists r.D_1$ As T is a Hintikka tree, if $\phi(j) = \exists r.D_1$, then $D_1 \in T(w \cdot j)$ and so $w \cdot j \in (D_1)^{\mathcal{I}_{i+1}}$. Furthermore, $(w, w \cdot j) \in r^{\mathcal{I}_{i+1}}$. Hence, $w \in D^{\mathcal{I}_{i+1}}$.

$\forall r.D_1$ Let $(w, w \cdot j) \in r^{\mathcal{I}_{i+1}}$. Then $\phi(j) \in T(w)$; but as T is a Hintikka tree, this implies that $D_1 \in T(w \cdot j)$. Thus, $w \cdot j \in (D_1)^{\mathcal{I}_{i+1}}$. As this is true for every successor, $w \in D^{\mathcal{I}_{i+1}}$.

Thus, as $C \in T(\varepsilon), \varepsilon \in C^{\mathcal{I}}$. Which means that C is satisfiable.

Conversely, let $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ be a model of $\mathcal{T}$ such that there is an $x \in \Delta^{\mathcal{I}}$ with $x \in C^{\mathcal{I}}$. It is possible then to construct a $\mathcal{T}$-expandable Hintikka tree

T by inductively defining a mapping $\eta : \{1, \ldots, k\}^* \to \Delta^{\mathcal{I}} \cup \{✠\}$, where $✠ \notin \Delta^{\mathcal{I}}$, as follows:

For the beginning of the induction, set

- $\eta(\varepsilon) = x$
- $T(\varepsilon) = \{D \in sub_{\mathcal{T}}(C) \mid x \in D^{\mathcal{I}}\}$

For the induction step, let $w \in \{1, \ldots, k\}^*$ such that $\eta(w)$ is already defined, and $i \in \{1, \ldots, k\}$. The following two cases are distinguished:

1. if $\eta(w) \neq ✠$ and $\phi(i) = \exists r.D \in T(w)$, then select a $y \in \Delta^{\mathcal{I}}$ such that $(\eta(w), y) \in r^{\mathcal{I}}$ and $y \in D^{\mathcal{I}}$ (such a y must exist, as $\mathcal{I}$ is a model). Set now $\eta(w \cdot i) = y$ and $T(w \cdot i) = \{E \in sub_{\mathcal{T}}(C) \mid y \in E^{\mathcal{I}}\}$.
2. in any other case, set $\eta(w \cdot i) = ✠$ and $T(w \cdot i) = \emptyset$.

By construction, T and η satisfy that, whenever $\eta(w) \neq ✠$, then $T(w) = \{D \in sub_{\mathcal{T}}(C) \mid \eta(w) \in D^{\mathcal{I}}\}$, and $T(w) = \emptyset$ if $\eta(w) = ✠$. This implies that $T(w)$ is a Hintikka set for every w.

By the inductive step, whenever $\phi(i) = \exists r.D \in T(w)$, then also $D \in T(w \cdot i)$. Even more, if there is a concept term of the form $\forall r.E$ in $T(w)$, as $(w, w \cdot i) \in r^{\mathcal{I}}$, then it must be the case that $E \in T(w \cdot i)$. Hence, T is a Hintikka tree.

Again, as $\mathcal{I}$ is a model, for every $A \doteq D \in \mathcal{T}$, whenever $\eta(w) \in A^{\mathcal{I}}$, it must be the case that $\eta(w) \in D^{\mathcal{I}}$. Thus, if $A \in T(w)$, then also $D \in T(w)$. Thus, T is $\mathcal{T}$-expandable.

The induction start yields $C \in T(\varepsilon)$, which was the only remaining property to prove. ■

This lemma implies that if it is possible to define an automaton whose runs define the Hintikka trees having the original formula as an element of the root, then the satisfiability problem can be solved by means of the emptiness problem for this automaton. The next definition gives the construction of the desired automaton.

Definition 4.10 *Let C be an $\mathcal{ALC}$-concept term and $\mathcal{T}$ an acyclic TBox such that $sub_{\mathcal{T}}(C)$ contains k existential formulas. Then the looping automaton $M_C^e = (Q, \Delta, I)$ is given by:*

- $Q = \{S \mid S \textit{ is a } \mathcal{T}\textit{-expandable Hintikka set}\}$
- $I = \{S \in Q \mid C \in S\}$
- $\Delta = \{(S, S_1, \ldots, S_k) \mid (S, S_1, \ldots, S_k) \textit{ is appropriate}\}$

Obviously, the runs of this automaton are exactly the desired Hintikka trees. Unfortunately, as in the case of M_C, the size of M_C^e is exponential in the size of C. Nonetheless, a partition similar to the one used for empty TBoxes shows that M_C^e is weakly-n-reducible, for some n. This will lead to the PSpace result.

Before giving the partition, a new concept similar to the role depth, the expanded role depth, is given.

Definition 4.11 *Let $\mathcal{T}$ be an acyclic TBox. The* expanded role depth *of a concept term $C, rd^e(C)$, is inductively defined as follows:*

- $rd^e(A) = rd^e(\neg A) = 0$ *if A is a primitive concept name*
- $rd^e(A) = rd^e(\neg A) = rd^e(C)$ *if $A \doteq C \in \mathcal{T}$*
- $rd^e(C \sqcup D) = rd^e(C \sqcap D) = \max\{rd^e(C), rd^e(D)\}$
- $rd^e(\forall r.C) = rd^e(C) + 1$
- $rd^e(\exists r.C) = rd^e(C) + 1$

If S is a set of concept terms, then $rd^e(S) = \max\{rd(D) \mid D \in S\}$.

Notice that the expanded role depth is well defined for acyclic TBoxes, and corresponds exactly to the role depth that a concept term would have if all the defined concept names were substituted by the corresponding definition in the TBox. This is needed to ensure the capability of reducing the role depth when a defined concept name is substituted, given that the Hintikka sets must be expandable with respect to the TBox.

The rest of this section consists in proving that M_C^e is weakly-n-reducible, for some n. This is done in a fashion similar to the proof for M_C in Section 4.1.

Definition 4.12 *Let C be an $\mathcal{ALC}$-concept term, $\mathcal{T}$ an acyclic TBox, and $n \geq 0$. Then $sub_n^e(C) = \{D \in sub_{\mathcal{T}}(C) \mid rd^e(D) \leq \max\{0, n-1\}\}$.*

Lemma 4.13 *Let C be an $\mathcal{ALC}$-concept term, S a Hintikka set, and $n \geq 0$. Then $sub_n^e(C) \cap S$ is also a Hintikka set.*

Proof. As S is a Hintikka set, for every $A \in N_C, \{A, \neg A\} \not\subseteq S$. Hence, $\{A, \neg A\} \not\subseteq sub_n^e(C) \cap S$. If $C_1 \sqcup C_2 \in sub_n^e(C) \cap S$, then $rd^e(C_i) \leq rd^e(C_1 \sqcup C_2) \leq \max\{0, n-1\}$, and $\{C_1, C_2\} \subseteq S$. Thus, $\{C_1, C_2\} \subseteq sub_n^e(C) \cap S$. Analogously, if $C_1 \sqcap C_2 \in sub_n^e(C) \cap S$, then $rd^e(C_i) \leq rd^e(C_1 \sqcap C_2) \leq \max\{0, n-1\}$, and $\{C_1, C_2\} \cap S \neq \emptyset$. Thus, $\{C_1, C_2\} \cap sub_n^e(C) \cap S \neq \emptyset$. ■

The following lemmas show that, given the partition defined by the expanded role depth of the states, and the set of functions given by the intersection with the set of all sub-concept terms having a smaller expanded role depth, the automaton M_C^e satisfies the conditions required for being weakly-n-reducible.

Lemma 4.14 *Let $S, S_1, \ldots, S_k$ be Hintikka sets such that $rd^e(S) = n$. If the tuple $(S, S_1, \ldots, S_k)$ is appropriate, then $(S, sub_n^e(C) \cap S_1, \ldots, sub_n^e(C) \cap S_k)$ is appropriate.*

Furthermore, for all $1 \leq i \leq k, rd^e(sub_n^e(C) \cap S_i) \leq \max\{0, n-1\}$.

Proof. Recall that a tuple $(T, T_1, \ldots, T_k)$ is appropriate if whenever the i-the existential concept $\phi(i) = \exists r.D$ is in T, then D must be an element of T_i, and for all concepts of the form $\forall r.E \in T$, E must also belong to T_i.

Suppose that $\phi(i) = \exists r.D \in S$. Then, as $(S, S_1, \ldots, S_k)$ is appropriate, $D \in S_i$, and for all concept terms $\forall r.E \in S, E \in S_i$.

By the way rd^e was defined, $rd^e(D) < rd^e(\exists r.D) \leq rd(S) = n$, and also $rd^e(E) < rd^e(\forall r.E) \leq rd(S) = n$. Thus, $D \in sub_n^e(C) \cap S_i$, and for all $\forall r.E \in S, E \in sub_n^e \cap S_i$.

Hence, $(S, sub_n^e(C) \cap S_1, \ldots, sub_n^e(C) \cap S_k)$ is appropriate. ■

Lemma 4.15 *Let $S, S_1, \ldots, S_k$ be Hintikka sets, and $n \geq 0$. If the tuple $(S, S_1, \ldots, S_k)$ is appropriate, then $(sub_n^e(C) \cap S, sub_n^e(C) \cap S_1, \ldots, sub_n^e(C) \cap S_k)$ is appropriate.*

Proof. If $\phi(i) = \exists r.D \in sub_n^e(C) \cap S$, then, as $(S, S_1, \ldots, S_k)$ is appropriate, $D \in S_i$. Furthermore, $rd^e(D) < rd^e(\exists r.D) \leq \max\{0, n-1\}$, and so $D \in sub_n^e(C) \cap S_i$.

If additionally there is a formula $\forall r.E \in sub^e_n(C) \cap S$, then $E \in S_i$ and $rd^e(E) < rd^e(\forall r.E) \leq \max\{0, n-1\}$, and so $E \in sub^e_n(C) \cap S_i$.

Hence, $(sub^e_n(C) \cap S, sub^e_n(C) \cap S_1, \ldots, sub^e_n(C) \cap S_k)$ is appropriate. ■

Theorem 4.16 *Let C be an $\mathcal{ALC}$-concept term, $\mathcal{T}$ an acyclic TBox, and $n = rd^e(sub_{\mathcal{T}}(C))$. Then, M^e_C is weakly-n-reducible.*

Proof. Define the classes $Q_i = \{S \in Q \mid rd^e(S) = i\}$, for $0 \leq i \leq n$. And let $f_S(T) = sub_{rd^e(S)}(C) \cap T$ for every $S, T \in Q$.

Lemmas 4.14 and 4.15 ensure that this partition, with the given set of functions, satisfy Conditions 1 and 2 of Definition 3.7. Hence, the only remaining part is to show that M^e_C is Q_0-looping.

Let S in Q_0, then $rd^e(S) = 0$. This means that there are no existential restrictions in S, hence the $k+1$-tuple $(S, \ldots, S) \in \Delta$. As M^e_C is a looping automaton, $Q_0 \subseteq Q = F$. Hence, M_C is Q_0-looping. ■

Corollary 4.17 *Satisfiability of $\mathcal{ALC}$-concept terms with respect to acyclic TBoxes is in PSpace.*

Proof. Given an $\mathcal{ALC}$-concept term C, construct the automaton M_C which is exponential in the size of C.

Let $n = rd^e(sub_{\mathcal{T}}(C))$, this number is linear in the size of C. As M^e_C is weakly-n-reducible, the emptiness problem of it can be solved by a non-deterministic method using space $O(log|Q| \cdot k \cdot n)$, where k is the number of existential formulas in $sub_{\mathcal{T}}(C)$.

As $log(|Q|)$ is polynomial in the size of C, and k is linear on the same size, only polynomial space is needed, with respect to the length of C.

As the method is non-deterministic, this shows that satisfiability of $\mathcal{ALC}$-concept terms with respect to acyclic TBoxes is in NPSpace. By Savitch's Theorem, it is known that NPSpace=PSpace. Hence, the satisfiability problem is in PSpace. ■

Thus, an alternative proof that $\mathcal{ALC}$ satisfiability with respect to acyclic TBoxes is in PSpace has been given. This proof needed a new automaton created explicitly for this case. Nonetheless, only the properties for weakly-n-reducibility needed to be proved to find the optimized resource bound for the solution of the decision problem.

The framework presented in Chapter 3 generalizes previous results by using the same method in two different automata, yielding in both cases an improved emptiness test in them.

Chapter 5

Conclusions

In this work, a new class of automata, called reducible automata, was defined. It was then proved that, for this class of automata, an emptiness test can be performed in a way that uses less resources than the regular test for general automata.

More explicitly, it was proved that the emptiness test for reducible automata can be solved using only logarithmic space on the number of states of the automaton, depending solely on the branching factor, and the number of classes in the partition defining the reducibility of the automaton.

Considering that, for general automata, the emptiness test requires polynomial time, this optimized method yields a considerable improvement. Such an improvement is more clear when the automaton has a big number of states.

This class of automata is then used to prove that both satisfiability of $\mathcal{ALC}$-concept terms, and $\mathcal{ALC}$ with respect to acyclic TBoxes is in PSpace. These results are not new, nonetheless they show that the framework can be applied to questions of interest.

In the first case, this was done using the same automaton that yields an exponential decision procedure for $\mathcal{ALC}$ satisfiability with respect to general TBoxes. For this last problem, the exponential procedure is optimal.

For satisfiability with respect to acyclic TBoxes, it was necessary to build a new automata-based decision procedure, as no adequate partition was found for showing reducibility, when using the automata for general TBoxes. This was mainly caused by the way concept definitions are changed into GCIs, making it impossible for the role-depth to decrease. This by no

means implies that there exist no such partitions, but only that they were not found during the research for this work.

Although different automata were used for the two examples, it was possible to use the exact same framework on both. This shows that the method works not only for one specific application, but is useful in general.

This framework could then be used for optimally solving other decision problems. For example, when trying to show that some problem is in PSpace, it would be sufficient to find out if it is possible to reduce this problem to the emptiness test of an exponentially large reducible automaton. This generalization may then be useful beyond the applications shown here.

Possible further work on this topic could be to search for a more general class of automata that still allows an optimized emptiness test. On another branch, it would be interesting to find more applications of the same framework, either in logic, or within other areas of knowledge.

Bibliography

[1] Franz Baader. Logic-based knowledge representation. In *Artificial Intelligence Today*, pages 13–41. 1999.

[2] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[3] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[4] F. Lehmann. *Semantic Networks in Artificial Intelligence*. Elsevier Science Inc., New York, NY, USA, 1992.

[5] C. Lutz. Complexity of terminological reasoning revisited. In *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning LPAR'99*, Lecture Notes in Artificial Intelligence, pages 181–200. Springer-Verlag, September 6 – 10, 1999.

[6] C. Lutz and U. Sattler. Mary likes all cats. In F. Baader and U. Sattler, editors, *Proceedings of the 2000 International Workshop in Description Logics (DL2000)*, number 33 in CEUR-WS, pages 213–226, Aachen, Germany, August 2000. RWTH Aachen. Proceedings online available from http://SunSITE.Informatik.RWTH-Aachen.DE/Publications/CEUR-WS/Vol-33/.

[7] Marvin Minsky. A framework for representing knowledge. Technical report, Cambridge, MA, USA, 1974.

[8] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.

[9] M. Quillian. Semantic memory. In M. Minsky, editor, *Semantic Information Processing*, pages 227–270. MIT Press, 1968.

[10] Michael O. Rabin. Weakly definable relations and special automata. In Y. Bar-Hillel, editor, *Mathematical Logic and Foundations of Set Theory*, pages 1–23. North Holland, 1970.

[11] Walter J. Savitch. Relationship between nondeterministic and deterministic tape classes. *J. Comp. Syst. Sci.*, 4:177–192, 1970.

[12] Klaus Schild. Terminological cycles and the propositional μ-calculus. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 509–520, Bonn, Germany, 1994. Morgan Kaufmann, Los Altos.

[13] Manfred Schmidt-Schauß; and Gert Smolka. Attributive concept descriptions with complements. *Artif. Intell.*, 48(1):1–26, 1991.

[14] A. Prasad Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theor. Comput. Sci.*, 49(2-3):217–237, 1987.

[15] Moshe Y Vardi and Pierre Wolper. Automata-theoretic techniques for modal logics of programs. *J. Comput. Syst. Sci.*, 32(2):183–221, 1986.

[16] Moshe Y. Vardi and Pierre Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 1994.