Automata-based Pinpointing for DLs

Rafael Peñaloza[⋆]

Intelligent Systems, University of Leipzig, Germany
penaloza@informatik.uni-leipzig.de

1 Introduction

Recent years have seen a boom in the creation and development of ontologies. Unfortunately, the maintenance of such ontologies is an error-prone process. On one side, it is in general unrealistic to expect a developer to be simultaneously a domain- and an ontology-expert. This leads to problems when a part of the domain is not correctly understood, or when, although correctly understood, is translated wrongly to the ontology language. On the other side, most of the larger ontologies are developed by a group of individuals. The difference in their points of view can produce unexpected consequences.

Whenever an error is identified, one would like to be able to detect the portion of the ontology responsible for such it; additionally, it would also be desirable to modify the ontology as little as possible to remove the error. If, for instance, an ontology is expresed by a TBox of an expressive Description Logic (DL), an unwanted consequence could be the unsatisfiability of a certain concept term C. Given that C is indeed unsatisfiable, we can search for a minimal sub-TBox that still leads to unsatisfiability of the concept (explaining the consequence), or for a maximal sub-TBox where C is satisfiable (removing the consequence). Finding these sets by hand in large ontologies is not a viable option.

Schlobach and Cornet [14] describe an algorithm for computing the minimal subsets of an unfoldable $\mathcal{ALC}$-terminology that keep the unsatisfiability of a concept. The algorithm extends the known tableau-based satisfiability algorithm for $\mathcal{ALC}$ [15], using labels to keep track of the axioms responsible of the generation of an assertion during the execution of the algorithm. A similar approach was actually presented previously in [2], for checking consistency of $\mathcal{ALC}$-ABoxes. The main difference between the algorithms in [14] and [2] is that the latter does not directly compute the minimal subsets that have the consequence, but rather a Boolean formula, called *pinpointing formula*, whose minimal satisfying valuations correspond to the minimal sub-ABoxes that are inconsistent. The ideas sketched by these algorithms have been applied to other tableau-based decision algorithms for more expressive DLs (see, e.g. [13, 12, 11]), and generalized in [3] where so-called *general tableaux* are extended into *pinpointing algorithms* that compute a formula as in [2]. This general approach was then successfuly applied for explaining subsumption relations in $\mathcal{EL}$ [4].

The main drawback of the general approach in [3] is that it assumes that the original tableau algorithm stops after a finite number of steps without the

[⋆] Funded by the German Research Foundation (DFG) under grant GRK 446.

need of cycle-checking (or *blocking*) techniques. When dealing with expressive DLs, or even with general concept inclusions (GCIs) in $\mathcal{ALC}$, this assumption is not satisfied. The pinpointing extension described in [11] for $\mathcal{ALC}$ with GCIs, follows the same ideas of [2, 3], but shows that the blocking conditions need to be handled with care. Furthermore, the pinpointing extensions of terminating tableau algorithms need not terminate; thus, it is unknown whether every blocking condition can be adapted to a pinpointing algorithm.

A different approach for checking properties in DL consists on reducing them to the emptiness problem of an automaton. The finite (bottom-up) emptiness test for automata yields then the desired decision. In this paper we will show how these automata-based decision procedures can also be extended to compute a pinpointing formula. We will motivate the construction, and its applicability to DLs, by using the DL $\mathcal{SI}$ with GCIs.

The paper is structured as follows. We begin by a brief introduction to automata theory, defining looping and weighted looping automata in Section 2. This section is followed by a description of the automata-based decision algorithm for satisfiability w.r.t. TBoxes in the DL $\mathcal{SI}$. Afterwards, in Section 4 we show how weighted looping automata can be used for computing a pinpointing formula, along with a short analysis of the time bounds required for this task.

2 Automata Theory

We will begin by defining the automata that are used for deciding and explaining properties such as (un)satisfiability in DLs. These automata operate on infinite k-ary trees. For a positive integer k, we denote the set $\{1, \ldots, k\}$ by K. The nodes in these trees are identified by words in K^* as follows: the root node is identified by ε and the i-th successor of the node u is identified by $u \cdot i$ for $1 \leq i \leq k$. In the case of labelled trees, we will refer to the labelling of a the node u in the tree t by $t(u)$, and the set of all labels appearing in t by $t(K^*) = \{t(u) \mid u \in K^*\}$.

The automata used here do not include an alphabet for labeling the nodes in the trees. In general, when deciding the emptyness problem for these automata, we are interested only in the existence of a tree that is accepted by the automaton, but not in the labelings it may contain. Since all the information relevant for the existence is contained in the states of the automaton, the node labels are redundant. In particular, this means that the language accepted by such automata is either empty or contains the (only) unlabelled infinite k-ary tree.

2.1 Looping Automata

The satisfiability problem can be decided in several DLs with the help of *looping automata*; that is, automata on infinite trees that do not impose any acceptance condition. This is the case also for any other property that can be decided by the presence or absence of an infinite tree model.

Definition 1 (Automaton, run). *A* looping tree automaton *over k-ary trees is a tuple (Q, Δ, I), where Q is a finite set of* states*, $\Delta \subseteq Q^{k+1}$ is the* transition relation*, and $I \subseteq Q$ is the set of* initial states*.*

A run *of this automaton on the (unique) unlabelled k-ary tree t is a labelled k-ary tree $r : K^* \to Q$ such that $(r(u), r(u\cdot 1), \ldots, r(u\cdot k)) \in \Delta$ for all $u \in K^*$. The run is* successful *if $r(\varepsilon) \in I$. The* emptiness problem for looping tree automata *is the problem of deciding whether a given looping tree automaton has a successful run or not.*

The emptiness problem for looping automata can be decided using time polynomial in the size of the automaton. The idea consists on computing all the *bad* states (that is, states that do not occur in any run) in a *bottom-up* fashion [16, 6]: all the states that do not occur as first components in the transition relation are bad, and if all the transitions starting from a state q lead to a known bad state, then q is also bad. The automaton has a successful run if and only if there is an initial state that is *not* bad.

2.2 Weighted Looping Automata

In some cases, deciding whether there is a successful run or not is not enough, and we want to asign a value to these runs when accepting a tree. That occurs when trying to explain the decision obtained: every accepted tree is assigned a *weight* that contains information useful for the desired explanation. In general, those weights are elements of a semiring.

A *semiring* is a tuple $(S, \oplus, \otimes, 0, 1)$ where S is a set, $\oplus$ and $\otimes$ are associative binary operators with identity elements 0 and 1, respectively, such that $\otimes$ distributes over $\oplus$, 0 absorbs over $\otimes$, and $\oplus$ is commutative.

Definition 2 (Weighted automaton). *Let S be a semiring. A* weighted looping automaton *(WLA) on S over k-ary trees is a tuple $\mathcal{A} = (Q, in, wt)$ where Q is a finite set of states, $in : Q \to S$ is the* initial distribution*, and $wt : Q^{k+1} \to S$ is the mapping of* weights *of the transitions of the automaton.*

A run *is simply a labelled tree $r : K^* \to Q$. The* weight *of this run is given by $wt(r) = in(r(\varepsilon)) \otimes \bigotimes_{u \in K^*} wt(u, u\cdot 1, \ldots, u\cdot k)$. The* behaviour *of this automaton is $\|\mathcal{A}\| = \bigoplus_{r:K^* \to Q} wt(r)$.*

Notice that for the definition of behaviour of a WLA to make sense, it is necessary to be able to perform infinite additions and products, and the product needs to be commutative even in the infinite case. In other words, we cannot choose any semiring but only one that is *totally commutative complete*. For a formal description of these semirings see e.g. [8].

3 $\mathcal{SI}$ satisfiability with TBoxes

We will show the applicability of our approach to DLs by describing how can one obtain explanations of unsatisfiability of a $\mathcal{SI}$ concept term with respect to a TBox. The DL $\mathcal{SI}$ extends $\mathcal{ALC}$ with transitive and inverse roles. In other words, if N_C is the set of concept names and N_R the set of role names, then there is a set $N_T \subseteq N_R$ of *transitive* role names such that for every $r \in N_T$ we can use r^- as a role name when building concept expressions. The semantics of this logic is defined in the usual way. See, for example, [10] for a more formal description. Given a $\mathcal{SI}$ role s, the *inverse* of s (denoted by $\bar{s}$) is s^- if s is a role name, and r if $s = r^-$. We will also use the predicate $\mathsf{trans}(r)$ on $\mathcal{SI}$ roles to express that either r or $\bar{r}$ belongs to N_T.

A TBox is a set of General Concept Inclusion axioms (GCIs) of the form $C \sqsubseteq D$ where both C and D are $\mathcal{SI}$ concept terms. The semantics of TBoxes and the satisfiability problem are defined as usual (see [5]).

$\mathcal{SI}$ has the tree model property. A tree model for a satisfiable $\mathcal{SI}$ concept can be obtained through unravelling [7]. For example, the $\mathcal{SI}$ concept A is satisfiable with respect to the TBox $\{A \sqsubseteq \exists r.A\}$ in a model having just one element belonging to A and related to itself via the role r. Unravelling this model yields a sequence $d_0, d_1, d_2, \dots$ of elements, all belonging to A, where d_i is related to d_{i+1} via the role r, for all $i \geq 0$. To obtain these tree models, we will construct so-called Hintikka trees. Intuitively, Hintikka trees are tree models where every node is labelled with the concepts to which the element represented by the node belongs. These concepts must be subconcepts of the concept tested for satisfiability or the concepts appearing in the TBox. In our small example, all the nodes d_i would be labelled by the concepts A and $\exists r.A$, since each element belongs to both of them.

To simplify the notation, we assume in the following that all concepts are in *negation normal form* (NNF); that is, negation appears only directly in front of concept names. Any $\mathcal{SI}$ concept can be transformed into NNF in linear time using de Morgan's laws, duality of quantifiers, and elimination of double negations. We denote the NNF of a concept C by $\mathsf{nnf}(C)$ and $\mathsf{nnf}(\neg C)$ by $\backsim C$.

Definition 3 (Hintikka set). *The set of* subconcepts *of an $\mathcal{SI}$ concept C ($\mathsf{sub}(C)$) is the least set S that contains C and has the following properties: if S contains $\neg A$ for a concept name A, then $A \in S$; if S contains $D \sqcap E$ or $D \sqcup E$, 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:*

$$\mathsf{sub}(C, \mathcal{T}) = \mathsf{sub}(C) \cup \bigcup_{D \sqsubseteq E \in \mathcal{T}} \mathsf{sub}(\backsim D \sqcup E)$$

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 $\{D, E\} \cap H \neq \emptyset$; and there is no concept name A such that $\{A, \neg A\} \subseteq H$.*

For a TBox $\mathcal{T}$, a Hintikka set H is called $\mathcal{T}$-expanded *if for every GCI $D \sqsubseteq E \in \mathcal{T}$ it holds that $\backsim D \sqcup E \in H$.*

Hintikka trees for C and $\mathcal{T}$ are infinite trees of a fixed arity k, which is determined by the number of existential restrictions – concepts of the form $\exists r.D$ – in $\mathsf{sub}(C, \mathcal{T})$. In our definition, we will need to know which successor in the tree corresponds to which existential restriction. For this purpose, we fix a linear order on the existential restrictions in $\mathsf{sub}(C,\mathcal{T})$. Let $\varphi : \{\exists r.D \in \mathsf{sub}(C,\mathcal{T})\} \to K$ be the corresponding ordering function; that is, $\varphi(\exists r.D)$ determines the successor node corresponding to $\exists r.D$. In general, such a successor node need not exist in a tree model. To obtain a full k-ary tree, Hintikka trees contain appropriate dummy nodes.

Definition 4 (Hintikka tree). *The tuple of Hintikka sets $(H_0, H_1, \ldots, H_k)$ is called $C,\mathcal{T}$-*compatible *if the following holds for every existential concept $\exists r.D \in \mathsf{sub}(C,\mathcal{T})$:*

- *if $\exists r.D \in H_0$, then*
  1. *$H_{\varphi(\exists r.D)}$ contains D, every concept E for which there is a universal restriction $\forall r.E \in H_0$, and additionally $\forall r.E$ if $\mathsf{trans}(r)$;*
  2. *for every concept $\forall \bar{r}.F \in H_{\varphi(\exists r.D)}$, H_0 contains F, and additionally $\forall \bar{r}.F$ if $\mathsf{trans}(r)$.*
- *if $\exists r.D \notin H_0$, then $H_{\varphi(\exists r.D)} = \emptyset$.*

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

This definition of Hintikka trees ensures that its existence characterizes satisfiability of $\mathcal{SI}$ concepts. The transitivity is dealt with by transfering all universal restrictions to the "successor" with respect to the transitive role, while the fact that restrictions can be also applied to the parent node in the tree handles the inverses. In [9, 1], these Hintikka trees are extended with additional data structures that allow detecting cycles in the tree in a depth as small as possible. Since we are not interested in the detection of such cycles, we do not require the data structures either.

Theorem 1. *The $\mathcal{SI}$ concept C is satisfiable w.r.t. the TBox $\mathcal{T}$ iff there exists a Hintikka tree for C and $\mathcal{T}$.*

We can construct now a looping tree automaton whose successful runs are exactly the Hintikka trees for C and $\mathcal{T}$. We would then be able to decide satisfiability of C with respect to $\mathcal{T}$ by performing an emptiness test on the automaton.

Definition 5 (Automaton $\mathcal{A}_{C,\mathcal{T}}$). *For an $\mathcal{SI}$ concept C and a TBox $\mathcal{T}$, let k be the number of existential restrictions in $\mathsf{sub}(C,\mathcal{T})$. The looping automaton $\mathcal{A}_{C,\mathcal{T}} = (Q, \Delta, I)$ is defined as follows:*

- *Q consists of all $\mathcal{T}$-expanded Hintikka sets for C;*
- *Δ consists of all $C,\mathcal{T}$-compatible tuples $(H_0, H_1, \ldots, H_k)$;*
- *$I = \{H \in Q \mid C \in H\}$.*

Theorem 2. *C is satisfiable w.r.t. $\mathcal{T}$ iff $\mathcal{A}_{C,\mathcal{T}}$ has a successful run.*

We have shown until now how we can use automata to decide satisfiability of $\mathcal{SI}$ concepts w.r.t. TBoxes. This approach has the advantage that it requires no additional cycle checking techniques. We will now turn our attention on extending this approach into a method that will allow us to explain the unsatisfiability of a concept; in other words, we want to find which axioms of the TBox are responsible for the concept to be unsatisfiable. In general, we will construct a so-called *pinpointing automaton*; a WLA whose behaviour contains all the information of the causes of the property to hold.

4 Automata-based Pinpointing

If we are given a set $\mathfrak{T}$ of *axioms*, then a *property* $\mathcal{P}$ is a set of finite subsets $\mathcal{T} \in \mathscr{P}_{fin}(\mathfrak{T})$. In the previous section we defined an automaton that depended on a set of axioms, which could be used to decide a property; in that case, (un)satisfiability of a concept. If the concept C turns out to be unsatisfiable w.r.t. the given TBox, we would like to be able to find out a minimal, with respect to set inclusion, sub-TBox w.r.t. which C stays unsatisfiable. In a more general scenario, we want to find a minimal subset of axioms – or *explanation* – from which the property tested still follows. For this task to make sense, it is necessary that the property is monotonic in the sense that if $\mathcal{T} \in \mathcal{P}$, then for every superset $\mathcal{T}' \supseteq \mathcal{T}$, it also holds that $\mathcal{T}' \in \mathcal{P}$. Notice that unsatisfiability w.r.t. TBoxes is in fact monotonic. Whenever we talk of a property, we will assume that it meets this monotonicity requirement.

In order to find an explanation we need to know how each of the axioms affects the runs of the automaton. We will do this with the help of a restricting function. Intuitively, the restricting function will tell us which states can be used in a run if a given axiom is present.

Definition 6 (Axiomatic automata). *Let $\mathcal{A} = (Q, \Delta, I)$ be a looping automaton over k-ary trees and $\mathcal{T}$ a set of axioms. The* restricting function *is a function* $\mathsf{res} : \mathcal{T} \to \mathscr{P}(Q)$. *The restricting function is extended to sets of axioms as follows: for $\mathcal{T}' \subseteq \mathcal{T}, \mathsf{res}(\mathcal{T}') = \bigcap_{t \in \mathcal{T}'} \mathsf{res}(t)$.*

For $\mathcal{T}' \subseteq \mathcal{T}$, $\mathcal{A}_{|\mathcal{T}'} = (Q \cap \mathsf{res}(\mathcal{T}'), \Delta \cap (\mathsf{res}(\mathcal{T}'))^{k+1}, I \cap \mathsf{res}(\mathcal{T}'))$ is called the $\mathcal{T}'$-restricted subautomaton *of $\mathcal{A}$. The set of* axiomatic automata *for $\mathcal{A}$ w.r.t.* res *is denoted by $(\mathcal{A}, \mathsf{res}) = \{\mathcal{A}_{|\mathcal{T}'} \mid \mathcal{T}' \subseteq \mathcal{T}\}$.*

Given a property $\mathcal{P}$, we say that $(\mathcal{A}, \mathsf{res})$ is correct *for $\mathcal{P}$ if for every $\mathcal{T}' \subseteq \mathcal{T}$ it holds that $\mathcal{T}' \in \mathcal{P}$ iff $\mathcal{A}_{|\mathcal{T}'}$ has no successful runs.*

In the automaton defined in Section 3 for deciding $\mathcal{SI}$ (un)satisfiability, the axioms restrict the set of states by forcing each Hintikka set to be $\mathcal{T}$-expanded. This is the only condition that depends on the GCIs used; hence, we can remove this condition in the definition of the looping automaton and use it as a restricting function to define a set of axiomatic automata that is correct for the property "C is unsatisfiable w.r.t. the TBox". More precisely, this set of axiomatic automata is given by $(\mathcal{A}_C, \mathsf{res})$, with $\mathcal{A}_C = (Q, \Delta, I)$, where Q consists of all Hintikka sets for C, Δ contains all $C, \mathcal{T}$-compatible $k + 1$-tuples,

$I = \{H \in Q \mid C \in I\}$; and for every $t \in \mathcal{T}$, we have $\mathsf{res}(t) = \{H \in Q \mid H$ is $\{t\}$-expanded$\}$.

Notice that in particular the subautomaton $\mathcal{A}_{C|\mathcal{T}}$ is exactly the same as the automaton $\mathcal{A}_{C,\mathcal{T}}$ from Definition 5. The benefit of making this change is that we are now able to understand the behaviour of the automaton in the absense of some of the axioms.

A naïve approach for finding an explanation consists on deciding the emptiness problem for the subautomata obtained by removing some of the axioms, until one subset of axioms is found such that the property holds for it, but for none of its proper subsets. Another approach consists on computing a pinpointing formula [3]. We assume that every axiom $t \in \mathcal{T}$ is labelled with a unique propositional variable, $\mathsf{lab}(t)$. Let $\mathsf{lab}(\mathcal{T})$ be the set of all propositional variables labeling an axiom in $\mathcal{T}$. A *monotone Boolean fomula* over $\mathsf{lab}(\mathcal{T})$ is a Boolean formula using (some of) the variables in $\mathsf{lab}(\mathcal{T})$ and only the connectives conjunction and disjunction. We identify a propositional *valuation* with the set of propositional variables that it makes true. Given a valuation $\mathcal{V} \subseteq \mathsf{lab}(\mathcal{T})$, we denote $\mathcal{T}_\mathcal{V} = \{t \in \mathcal{T} \mid \mathsf{lab}(t) \in \mathcal{V}\}$. For a property $\mathcal{P}$ and a set of axioms $\mathcal{T}$, a monotone Boolean formula ϕ over $\mathsf{lab}(\mathcal{T})$ is called a *pinpointing formula* for $\mathcal{P}$ and $\mathcal{T}$ if for every valuation $\mathcal{V} \subseteq \mathsf{lab}(\mathcal{T})$ it holds that $\mathcal{T}_\mathcal{V} \in \mathcal{P}$ iff $\mathcal{V}$ satisfies ϕ.

From a pinpointing formula one can easily find out the minimal sets of axioms for which the property still follows; they correspond to the minimal valuations that satisfy it. Conversely, the maximal sets of axioms for which the same property does not hold correspond to the maximal valuations falsifying the formula.

We will use a set of axiomatic automata as a base for constructing a weighted looping automaton whose behaviour is a pinpointing formula. The semiring used needs to produce a monotonic Boolean formula by means of additions and products, but syntactic variations of the same formula must be treated equally. Hence, we use the $\mathcal{T}$-Boolean semiring $\mathbb{B}^\mathcal{T} = (\hat{\mathbb{B}}(\mathcal{T}) \cup \{\top, \bot\}, \wedge, \vee, \top, \bot)$, where $\hat{\mathbb{B}}(\mathcal{T})$ is the quotient set of all monotonic Boolean formulas over $\mathsf{lab}(\mathcal{T})$ by the propositional equivalence relation; in other words, two propositionally equivalent formulas will correspond to the exact same element in $\hat{\mathbb{B}}(\mathcal{T})$. The constants $\top$ and $\bot$ correspond to a tautology and a contradiction, respectively.

Definition 7 (Pinpointing automaton). *Let* $(\mathcal{A}, \mathsf{res})$*, with* $\mathcal{A} = (Q, \Delta, I)$*, be a set of axiomatic automata and* $\mathcal{T}$ *a set of axioms. The* violating function $\mathsf{vio} : Q \to \mathbb{B}^\mathcal{T}$ *is defined for every* $q \in Q$ *by*

$$\mathsf{vio}(q) = \bigvee_{\{t \in \mathcal{T} \mid q \notin \mathsf{res}(t)\}} \mathsf{lab}(t).$$

The pinpointing automaton *for* $(\mathcal{A}, \mathsf{res})$ *w.r.t.* $\mathcal{T}$ *is the WLA* $\mathcal{A}^{\mathsf{pin}} = (Q, in, wt)$ *on* $\mathbb{B}^\mathcal{T}$*, where*

$$in(q) = \begin{cases} \mathsf{vio}(q) & \text{if } q \in I, \\ \top & \text{otherwise; and} \end{cases}$$

$$wt(q_0, q_1, \dots, q_k) = \begin{cases} \bigvee_{i=1}^k \mathsf{vio}(q_i) & \text{if } (q_0, q_1, \dots, q_k) \in \Delta, \\ \top & \text{otherwise.} \end{cases}$$

Notice that if r is a successful run of $\mathcal{A}$, then $wt(r) = \bigvee_{q \in r(K^*)} \mathsf{vio}(q)$; otherwise, $wt(r) = \top$. Intuitively, the violating function expresses which axioms are not satisfied by a given state, and thus the weight of a run accumulates all the axioms violated by any of the states appearing as labels in it. Removing all the axioms appearing in that formula would yield a subset of axioms for which it is possible to construct a run; and hence, the property does not hold anymore. Conjoining this information for all possible runs leads us to a pinpointing formula.

Theorem 3. *Let $\mathcal{P}$ be a property, $\mathcal{T}$ a set of axioms, and $(\mathcal{A}, \mathsf{res})$ a correct set of axiomatic automata for $\mathcal{P}$. Then $\|\mathcal{A}^{\mathsf{pin}}\|$ is a pinpointing formula for $\mathcal{P}$ and $\mathcal{T}$.*

This theorem shows that it is enough to compute the behaviour of the pinpointing automaton in order to obtain all the information necessary to extract the explanations for a property to hold. The question is now whether it is possible to effectively compute that behaviour. Clearly, the direct approach of computing and conjoining the weights for the infinitely many runs of infinite size is doomed to failure within finite resources. One idea for computing the behaviour consists in adapting the bottom-up method described in Section 2 for deciding the emptiness problem of unweighted looping automata. Recall that this procedure iterates labeling states as bad, depending on the transitions that start from them.

It is possible to reinterpret the same procedure as applied to a weighted automaton over the Boolean semiring $(\{0,1\}, \wedge, \vee, 1, 0)$, where the initial distribution maps every initial state to 0 and all the rest to 1, and a transition has weight 0 if it is an element of Δ and 1 otherwise. The behaviour of this WLA is 0 if and only if there is a successful run for the original unweighted automaton.

We can then iteratively construct the function $\mathsf{bad} : Q \to \{0,1\}$. Intuitively, if $\mathsf{bad}(q) = 1$, then q is a bad state. In the beginning, no state is considered to be bad, and hence the iteration begins by setting $\mathsf{bad}_0(q) = 0$ for all $q \in Q$. We then iterate as follows: given a state q, $\mathsf{bad}_{i+1}(q) = 1$ if every transition $(q, q_1, \ldots, q_k)$ starting from q leads to states known to be already bad; that is, if $\bigvee_{j=1}^{k} \mathsf{bad}_i(q_j) = 1$. Thus, the iteration takes the form:

$$\mathsf{bad}_{i+1}(q) = \bigwedge_{(q,q_1,\ldots,q_k) \in Q^{k+1}} wt(q, q_1, \ldots, q_k) \vee \bigvee_{j=1}^{k} \mathsf{bad}_i(q_j). \quad (1)$$

The function bad is then the limit of this iteration. It is easy to see that this limit is reached after linearly many steps, measured on the size of the automaton. In the end, we are only interested in knowing whether there is an initial state that is not bad; i.e. we compute $\bigwedge_{q \in Q} in(q) \vee \mathsf{bad}(q)$, which corresponds exactly to the behaviour of the WLA.

This procedure can be directly adapted to compute the behaviour of the automaton $\mathcal{A}^{\mathsf{pin}}$. Intuitively, $\mathsf{bad}(q)$ does not anymore show whether q is bad or not, but rather a formula that expresses the axioms that must be violated in order to construct a run that uses q as a label. We start again by considering that no axiom is violated, and hence $\mathsf{bad}(q) = \bot$ for all $q \in Q$. Notice that this corresponds to initializing the iteration with the neutral element of the product, as done in the previous case. From here, we can iterate again using the same process described by Equation (1).

We will proceed now to show both that the function bad can be computed in time polynomial on the number of states of $\mathcal{A}$, and that the formula $\bigwedge_{q\in Q} in(q) \vee \mathsf{bad}(q)$ corresponds to the behaviour of $\mathcal{A}^{\mathsf{pin}}$. To do this, we will use partial runs of depth m. Let $K^{\leq n} := \bigcup_{i=0}^{n} K^i$. A *partial run of depth m* for a looping automaton is a mapping $r : K^{\leq m-1} \to Q$ such that $(r(u), r(u\cdot 1), \ldots, r(u\cdot k)) \in \Delta$ for all $u \in K^{\leq m-2}$. For a weighted looping automaton, a partial run is simply a mapping $r : K^{\leq m-1} \to Q$. All the notations and terminology used for runs will also be applied for partial runs. We further denote $K_q^i = \{r : K^{\leq i} \to Q \mid r(\varepsilon) = q\}$.

Lemma 1. *For all $i \geq 0$ and $q \in Q$ it holds that*

$$\mathsf{bad}_i(q) = \bigwedge_{r\in K_q^i} \bigvee_{u\in K^{\leq i-1}} wt(r(u), r(u\cdot 1), \ldots, r(u\cdot k))$$

This lemma can be proved by induction, by simply applying the definition of bad_{i+1} in terms of bad_i. The intuition behind this lemma is that after the i-th iteration, the function $\mathsf{bad}_i(q)$ states the conjunction of weights of all the partial runs up to depth i. Hence, if computing the partial runs of a certain depth m is enough for knowing the existence of a run, then only m iterations are necessary to compute the limit bad.

Definition 8 (m-complete). *A looping automaton $\mathcal{A}$ is called* m-complete *if the following property holds: $\mathcal{A}$ has a successful run iff $\mathcal{A}$ has a successful partial run of depth m.*

It is easy to see that every looping automaton $\mathcal{A} = (Q, \Delta, I)$ is m-complete for every m greater than the cardinality of Q. However, there are also classes of automata for which this bound can be lowered [1]. The following corollary follows easily from Lemma 1.

Corollary 1. *If $\mathcal{A}$ is m-complete, then* $\mathsf{bad}_{m+1} = \mathsf{bad}_m = \mathsf{bad}$.

We know now that we need at most as many iterations as there are states in the automaton to find the fixed point of bad. It remains to show that this process is indeed helpful for computing the behaviour of the WLA.

Theorem 4. $\|\mathcal{A}^{\mathsf{pin}}\| = \bigwedge_{q\in Q} in(q) \vee \mathsf{bad}(q)$

Proof. Let n be such that $\mathsf{bad}_n(q) = \mathsf{bad}_{n+1}(q)$ for all q, and $\mathcal{V}$ a valuation. Assume that $\mathcal{V}$ does not satisfy $\|\mathcal{A}^{\mathsf{pin}}\|$. Then there must exist a run r such that $\mathcal{V}$ does not satisfy $wt(r) = in(r(\varepsilon)) \cup \bigvee_{u\in K^*} wt(r(u), r(u\cdot 1), \ldots, r(u\cdot k))$. In particular, $\mathcal{V}$ satisfies neither $in(r(\varepsilon))$ nor $\bigvee_{u\in K^{\leq n}} wt(r(u), r(u\cdot 1), \ldots, r(u\cdot k))$.

By Lemma 1, $\mathcal{V}$ cannot satisfy $\mathsf{bad}_n(r(\varepsilon)) = \mathsf{bad}(r(\varepsilon))$. Thus, $\mathcal{V}$ does not satisfy $in(r(\varepsilon)) \vee \mathsf{bad}(r(\varepsilon))$.

Conversely, suppose that there is a $q \in Q$ such that $\mathcal{V}$ does not satisfy $in(q) \vee \mathsf{bad}(q)$. We can construct a run r whose weight is not satisfied by $\mathcal{V}$ as follows. First set $r(\varepsilon) = q$. Since $\mathcal{V}$ does not satisfy $\mathsf{bad}(q)$, there is a tuple $(q, q_1, \ldots, q_k) \in Q^{k+1}$ such that $\mathcal{V}$ satisfies neither $wt(q, q_1, \ldots, q_k)$ nor $\bigvee_{j=1}^{k} \mathsf{bad}(q_j)$. We set $r(i) = q_i$ for $1 \leq i \leq k$. We can then iterate this procedure for each of the new nodes in the tree, adding always transitions that are not satisfied by $\mathcal{V}$. Thus, $\mathcal{V}$ cannot satisfy $\|\mathcal{A}^{\mathsf{pin}}\|$. □

We have finally shown that we can compute the behaviour of a pinpointing automaton, and hence a pinpointing formula, in a time polynomial on the size of the automaton. Since the automaton constructed for $\mathcal{SI}$ has exponentially many states, measured on the size of the TBox, we would require exponential time for computing the pinpointing formula. This bound is optimal for $\mathcal{SI}$ w.r.t. TBoxes since only deciding unsatisfiability requires already exponential time. If we restrict our attention to $\mathcal{SI}$ with only acyclic TBoxes, we have a decision problem in PSPACE. We can also show that in this case, the automaton $\mathcal{A}_C$ is m^4-complete, where m is the cardinality of $\mathsf{sub}(C, \mathcal{T})$. Thus, we need only polynomially many iterations of the function bad to compute the pinpointing formula. Unfortunately, each iteration still requires the computation of a value for each state, and hence needs exponential time and space on the size of the TBox. It is unclear whether this exponential bound is also optimal for the restricted case or not.

5 Conclusions

We have introduced the notion of axiomatic automata, which can be easily applied to automata-based decision procedures for DLs; that is, axiomatic automata can decide unsatisfiability of concept terms with respect to sets of GCIs. We have then shown how to construct a weighted looping automaton from a set of axiomatic automata in such a way that the behaviour of the WLA corresponds to a pinpointing formula for the property decided by the axiomatic automata.

We have also shown a bottom-up procedure for computing this behaviour. Although it was presented only for pinpointing automata, this bottom-up algorithm can be adapted for computing the behaviour of any WLA in a straightforward manner. This procedure requires at most as many iterations as states in the automaton, and each iteration requires time polynomial to the number of states. The automata described here for unsatisfiability in DLs has exponentially many states with respect to the number of GCIs in the TBox. This means that the computation of a pinpointing formula requires an exponential time on the size of the TBox, even for logics where deciding unsatisfiability is in a lower complexity bound, such as $\mathcal{ALC}$ with acyclic TBoxes. One interesting question that arises is whether this bound is optimal or the algorithm can be improved for specific cases, for example, by constructing only parts of the automaton at a time.

References

[1] Franz Baader, Jan Hladik, and Rafael Peñaloza. Automata can show PSPACE results for description logics. *Information and Computation*, 2008. To appear.

[2] Franz Baader and Bernhard Hollunder. Embedding defaults into terminological knowledge representation formalisms. *J. of Automated Reasoning*, 14:149–180, 1995.

[3] Franz Baader and Rafael Peñaloza. Axiom pinpointing in general tableaux. In *Proc. of TABLEAUX 2007*, LNAI, Aix-en-Provence, France, 2007. Springer.

[4] Franz Baader, Rafael Peñaloza, and Boontawee Suntisrivaraporn. Pinpointing in the description logic $\mathcal{EL}^+$. In *Proc. of KI'07*, LNAI, Germany, 2007. Springer.

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

[6] Franz Baader and Stephan Tobies. The inverse method implements the automata approach for modal satisfiability. In *Proc. of IJCAR 2001*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 92–106. Springer-Verlag, 2001.

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

[8] Manfred Droste and George Rahonis. Weighted automata and weighted logics on infinite words. In Oscar H. Ibarra and Zhe Dang, editors, *Developments in Language Theory*, volume 4036 of *Lecture Notes in Computer Science*, pages 49–58. Springer, 2006.

[9] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. A PSpace-algorithm for deciding ALCNI$_{R^+}$-satisfiability. LTCS-Report LTCS-98-08, LuFG Theoretical Computer Science, RWTH Aachen, 1998.

[10] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proc. of the 6th Int. Conf. on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes in Artificial Intelligence, pages 161–180. Springer-Verlag, 1999.

[11] Kevin Lee, Thomas Meyer, and Jeff Z. Pan. Computing maximally satisfiable terminologies for the description logic *alc* with GCIs. In *Proc. of Description Logics 2006*, 2006.

[12] Thomas Meyer, Kevin Lee, Richard Booth, and Jeff Z. Pan. Finding maximally satisfiable terminologies for the description logic $\mathcal{ALC}$. In *Proc. of the 21st Nat. Conf. on Artificial Intelligence (AAAI 2006)*. AAAI Press/The MIT Press, 2006.

[13] Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Debugging OWL ontologies. In Allan Ellis and Tatsuya Hagino, editors, *Proc. of the 14th International Conference on World Wide Web (WWW'05)*, pages 633–640. ACM, 2005.

[14] Stefan Schlobach and Ronald Cornet. Non-standard reasoning services for the debugging of description logic terminologies. In Georg Gottlob and Toby Walsh, editors, *Proc. of IJCAI 2003*, pages 355–362, Acapulco, Mexico, 2003. Morgan Kaufmann, Los Altos.

[15] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with unions and complements. Technical Report SR-88-21, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern (Germany), 1988.

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