Reasoning with Prototypes in the Description Logic $\mathcal{ALC}$ using Weighted Tree Automata

Franz Baader and Andreas Ecke[⋆]

Theoretical Computer Science, TU Dresden, Germany
{baader,ecke}@tcs.inf.tu-dresden.de

Abstract. We introduce an extension to Description Logics that allows us to use prototypes to define concepts. To accomplish this, we introduce the notion of prototype distance functions (pdfs), which assign to each element of an interpretation a distance value. Based on this, we define a new concept constructor of the form $P_{\sim n}(d)$ for $\sim \in \{<, \leq, >, \geq\}$, which is interpreted as the set of all elements with a distance $\sim n$ according to the pdf d. We show how weighted alternating parity tree automata (wapta) over the non-negative integers can be used to define pdfs, and how this allows us to use both concepts and pointed interpretations as prototypes. Finally, we investigate the complexity of reasoning in $\mathcal{ALCP}$(wapta), which extends the Description Logic $\mathcal{ALC}$ with the constructors $P_{\sim n}(d)$ for pdfs defined using wapta.

1 Introduction

Description Logics (DLs) [3] can be used to formalize the important notions of an application domain as *concepts*, by formulating necessary and sufficient conditions for an individual to belong to the concept. Basically, such conditions can be (Boolean combinations of) atomic properties required for the individual (expressed by concept names) and properties that refer to relationships with other individuals and their properties (expressed as role restrictions). The expressivity of a particular DL depends on what kind of properties can be required and how they can be combined. Given an interpretation of the atomic entities (concept and role names), the semantics of a DL determines, for each concept expressed in this DL, its extension, i.e., the set of individuals satisfying all the conditions stated in the definition of the concept. Knowledge about the application domain is then represented by stating subconcept-superconcept relationships between concepts within a terminology (TBox). Given such a TBox, reasoning procedures can be used to derive implicit knowledge from the explicitly represented knowledge. For example, the satisfiability tests checks whether a given concept is non-contradictory w.r.t. the knowledge represented in the TBox.

In many applications, it is quite hard to give exact definitions of certain concepts. In fact, cognitive psychologists [9] argue that humans recognize categories by prototypes rather than concepts. For example, assume that we want to

⋆ Supported by DFG in the Research Training Group QuantLA (GRK 1763).

define the concept of a human from an anatomical point of view. One would be tempted to require two arms and two legs, five fingers on each hand, a heart on the left side, etc. However, none of these conditions are necessarily satisfied by an individual human being, though most of them should probably be satisfied to be categorized as human being. Thus, an anatomical description talking about arms and legs etc. describes a prototypical human being rather than necessary and sufficient conditions for being human. As an other example, taken from [8], consider the notion of a cup: we can say that cups are small, cylindrical, concave containers with handles, whose top side is open; they can hold liquids and are used for drinking; and they are made of plastic or porcelain. But again, this describes a prototypical cup rather than stating necessary and sufficient conditions for being a cup: square metal cups are easily imaginable, measuring cups are not used for drinking and may hold non-liquids such as flour, while sippy cups for toddlers are not open on the top. One could, of course, try to capture all such exceptional cups by using a big disjunction of (exactly defined) concepts, but this would obviously be rather clumsy and with high likelihood one would overlook some exceptions.

In order to be used within a formal knowledge representation language with automated reasoning capabilities, prototypes need to be equipped with a formal semantics. To obtain such a semantics, we use the ideas underlying Gärdenfors' conceptual spaces [6], where categories are explained in terms of convex regions, which are defined using the distance from a focal point. To obtain a concrete representation language, we need to define what are focal points and how to define the distance of an individual to such a focal point. Instead of employing prototypical individuals or concepts as focal points, we take a more abstract approach based on automata, which is inspired by the automata-approach for reasoning in DLs (see Section 3.2 in [1] for a gentle introduction). Basically, in this approach, a given concept C and a TBox $\mathcal{T}$ are translated into a tree automaton $\mathcal{A}_{C,\mathcal{T}}$ that accepts all the tree-shaped models of $\mathcal{T}$ whose root belongs to C. Testing satisfiability of C w.r.t. $\mathcal{T}$ then boils down to the emptiness test for $\mathcal{A}_{C,\mathcal{T}}$, i.e., checking whether there is a tree accepted by $\mathcal{A}_{C,\mathcal{T}}$. Instead of using a classical automaton that returns 1 (accepted) or 0 (not accepted) for an input tree, we propose to use a weighted automaton [5]. Intuitively, this automaton receives as input a tree-shaped interpretation and returns as output a non-negative integer, which we interpret as the distance of the individual at the root of the tree to the prototype (focal point) described by the automaton. This approach can be applied to non-tree-shaped models by the usual unraveling operation. In order to integrate such prototypes into a Description Logic, we propose to use thresholds to derive concepts from prototypes. More precisely, the threshold concept $P_{\sim n}(\mathcal{A})$ for $\sim \in \{<, \leq, >, \geq\}$ is interpreted as the set of all elements with a distance $\sim n$ according to the weighted automaton $\mathcal{A}$. The concepts obtained this way can then be used like atomic concepts within a DL.

It might appear to be more intuitive to use concepts or individuals rather than automata to describe prototypes. However, in these alternative settings, one then needs to give formal definitions of the distance between two individuals or between an individual and a concept, whereas in our approach this comes for free by the definition of the semantics of weighted automata. We show that these alternative settings can actually be seen as instances of our weighted automata approach.

In this paper, we investigate the extension $\mathcal{ALCP}$(wapta) of the DL $\mathcal{ALC}$ by threshold concepts defined using weighted alternating parity tree automata. In order to obtain inference procedures for the extended DL, the weighted automata are turned into automata that accept the cut-point language consisting of the trees whose distance (computed by the weighted automaton) is below a given threshold. In fact, this cut-point construction yields languages accepted by unweighted alternating parity tree automata, for which the emptiness problem is decidable. This allows us to extend the automata-approach for reasoning in $\mathcal{ALC}$ to $\mathcal{ALCP}$(wapta).

Regarding related work, non-monotonic logics are sometimes also used to formalize prototypes. However, there one usually tries to maximize typicality, i.e. one assumes that an individual stated to belong to a prototype concept has all the properties of the prototype, unless one is forced by other knowledge to retract this assumption. In contrast, our new logic is monotonic and we only conclude that an individual belongs to a threshold concept $P_{\sim n}(d)$ if this follows from the available knowledge. The work that comes closest to this paper is [2], where concepts of the lightweight DL $\mathcal{EL}$ are used to describe prototypes. To be more precise, the paper introduces a graded membership function, which for a given $\mathcal{EL}$-concept C and an individual d of an interpretation returns a membership degree in the interval $[0, 1]$. This is then used as "distance" to define threshold concepts and an extension of $\mathcal{EL}$ by such concepts basically in the same way as sketched above. The difference to the present work is, on the one hand, that prototypes are given by concepts rather than weighted automata and that the interval $[0, 1]$ is used in place of the non-negative integers. On the other hand, we consider a more expressive DL ($\mathcal{ALC}$ rather than $\mathcal{EL}$), and we can reason w.r.t. general TBoxes in the extended language, whereas the results in [2] are restricted to reasoning without a TBox.

2 Preliminaries

The Description Logic $\mathcal{ALC}$ $\mathcal{ALC}$-concepts are built from two disjoint sets N_C of concept names and N_R of role names using concept constructors. Every concept name is a basic $\mathcal{ALC}$-concept. Furthermore, one can construct complex $\mathcal{ALC}$-concepts using conjunction, disjunction, negation, and existential and universal restrictions as shown in Table 1. As usual, we use $\top$ as abbreviation for the concept $A \sqcup \neg A$, where A is an arbitrary concept name. The semantics of $\mathcal{ALC}$-concepts is defined using *interpretations* $\mathcal{I}$ consisting of a non-empty *interpretation domain* $\Delta^\mathcal{I}$ and an *interpretation function* $\cdot^\mathcal{I}$, which assigns to all concept names $A \in N_C$ a subset $A^\mathcal{I} \subseteq \Delta^\mathcal{I}$, and to all role names $r \in N_R$ a binary relation $r^\mathcal{I} \subseteq \Delta^\mathcal{I} \times \Delta^\mathcal{I}$ on the domain. The interpretation function is extended to complex concepts as shown in the last column of Table 1. Note that $\top$ is interpreted as $\Delta^\mathcal{I}$.

Table 1. Concept constructors for $\mathcal{ALC}$.

Constructor	Syntax	Semantics
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
existential restriction	$\exists r.C$	$\{d \in \Delta^{\mathcal{I}} \mid \exists e.(d,e) \in r^{\mathcal{I}} \wedge e \in C^{\mathcal{I}}\}$
universal restriction	$\forall r.C$	$\{d \in \Delta^{\mathcal{I}} \mid \forall e.(d,e) \in r^{\mathcal{I}} \Rightarrow e \in C^{\mathcal{I}}\}$

Terminological knowledge can be expressed using *general concept inclusions* (GCIs) of the form $C \sqsubseteq D$, where C and D are $\mathcal{ALC}$-concepts. A GCI $C \sqsubseteq D$ is satisfied by an interpretation $\mathcal{I}$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. A *TBox* $\mathcal{T}$ is a set of GCIs, and we call an interpretation $\mathcal{I}$ a *model* of $\mathcal{T}$ if it satisfies all GCIs in $\mathcal{T}$.

For example, to express that every container that has a handle and is only used to hold liquids is either a cup or a jug, one could use the GCI

$$\mathsf{Container} \sqcap \exists \mathsf{hasPart}.\mathsf{Handle} \sqcap \forall \mathsf{holds}.\mathsf{Liquid} \sqsubseteq \mathsf{Cup} \sqcup \mathsf{Jug}.$$

DL systems usually come equipped with a range of reasoning services. Standard inferences provided by most DL systems include concept satisfiability and subsumption. We say that a concept C is satisfiable w.r.t. a TBox $\mathcal{T}$ if there exists a model $\mathcal{I}$ of $\mathcal{T}$ with $C^{\mathcal{I}} \neq \emptyset$; and a concept C is subsumed by a concept D w.r.t. $\mathcal{T}$ ($C \sqsubseteq_{\mathcal{T}} D$) if for all models $\mathcal{I}$ of $\mathcal{T}$ we have $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. Subsumption can be reduced to concept satisfiability. Indeed, we have $C \sqsubseteq_{\mathcal{T}} D$ iff $C \sqcap \neg D$ is unsatisfiable in $\mathcal{T}$. Therefore, an algorithm that decides concept satisfiability can also be used to decide subsumption. It is well-known [1] that $\mathcal{ALC}$ has the tree model property, i.e., every satisfiable $\mathcal{ALC}$-concept C has a tree-shaped model in which the root of the tree is an instance of C. Thus, to decide concept satisfiability, it is enough to consider tree-shaped interpretations. In fact, we will show that the tree model property still holds for the extended logic with prototypes, which is important for our approach to work.

Deciding concept satisfiability using alternating parity tree automata We show how tree automata can be used to decide satisfiability of $\mathcal{ALC}$-concepts w.r.t. $\mathcal{ALC}$-TBoxes. This result is a simple adaptation of the approach in [10] to $\mathcal{ALC}$. This approach requires the concept and the TBox to be in negation normal form. Recall that an $\mathcal{ALC}$-concept C is in negation normal form if negation occurs only directly in front of concept names. Any concept can be transformed in linear time into an equivalent concept in negation normal form [1].

We can transform a TBox $\mathcal{T}$ into a single concept $C_{\mathcal{T}} = \bigsqcap_{C \sqsubseteq D \in \mathcal{T}} \neg C \sqcup D$; then an interpretation satisfies $\mathcal{T}$ iff it satisfies the GCI $\top \sqsubseteq C_{\mathcal{T}}$. In order to decide $\mathcal{ALC}$-concept satisfiability using tree automata, we first need to introduce the relevant notions from automata theory.

A *tree domain* is a prefix-closed, non-empty set $D \subseteq \mathbb{N}^*$, i.e., for every $ui \in D$ with $u \in \mathbb{N}^*$ and $i \in \mathbb{N}$ we also have $u \in D$. The elements of D are called nodes, the node ε is the root of D, and for every $u \in D$, the nodes $ui \in D$ are called children of u. A node is called a leaf, if it has no children. A path π in D is a subset $\pi \subseteq D$ such that $\varepsilon \in \pi$ and for every $u \in \pi$, u is either a leaf or there is a unique $i \in \mathbb{N}$ with $ui \in \pi$. Given an alphabet Σ, a Σ-labeled tree is a pair (dom_T, T) consisting of a tree domain dom_T and a labeling function $T : \mathrm{dom}_T \to \Sigma$. Instead of the pair (dom_T, T) we often use only T to denote a labeled tree. With $\mathrm{Tree}(\Sigma)$ we denote the set of all Σ-labeled trees.

The automata type we introduce now is based mainly on the alternating tree automata defined by Wilke [11], which are working on $\mathcal{P}(\Sigma)$-Trees, which are labeled with the power set of some finite alphabet Σ. Given such a Σ and a set of states Q, a transition condition $\mathrm{TC}(\Sigma, Q)$ is one of the following: true; false; σ or $\neg\sigma$ for $\sigma \in \Sigma$; $q_1 \wedge q_2$ or $q_1 \vee q_2$ for $q_1, q_2 \in Q$; or $\Box q$ or $\Diamond q$ for $q \in Q$.

Definition 1 (alternating parity tree automaton). *An alternating parity tree automaton (apta) $\mathcal{A}$ working on $\mathcal{P}(\Sigma)$-trees is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, \Omega)$, where 1. Σ is a finite alphabet; 2. Q is a finite set of states and $q_0 \in Q$ is the initial state; 3. $\delta : Q \to TC(\Sigma, Q)$ is the transition function; and 4. $\Omega : Q \to \mathbb{N}$ is the priority function that specifies the parity acceptance condition.*

Given a $\mathcal{P}(\Sigma)$-labeled tree T, a *run* is a $(\mathrm{dom}_T \times Q)$-labeled tree R such that $\varepsilon \in \mathrm{dom}_R$, $R(\varepsilon) = (\varepsilon, q_0)$, and for all $u \in \mathrm{dom}_R$ with $R(u) = (v, q)$ we have:

- $\delta(q) \neq \mathsf{false}$
- if $\delta(q) = \sigma$, then $\sigma \in T(v)$; and if $\delta(q) = \neg\sigma$, then $\sigma \notin T(v)$;
- if $\delta(q) = q_1 \wedge q_2$, then there exists $i_1, i_2 \in \mathbb{N}$ such that $R(ui_1) = (v, q_1)$ and $R(ui_2) = (v, q_2)$;
- if $\delta(q) = q_1 \vee q_2$, then there exists $i \in \mathbb{N}$ such that $R(ui) = (v, q_1)$ or $R(ui) = (v, q_2)$;
- if $\delta(q) = \Diamond q'$, then there exists $i, j \in \mathbb{N}$ with $R(ui) = (vj, q')$; and
- if $\delta(q) = \Box q'$, then for every $j \in \mathbb{N}$ with $vj \in \mathrm{dom}_T$ there exists $i \in \mathbb{N}$ with $R(ui) = (vj, q')$.

A run is *accepting*, if every infinite path in R satisfies the *parity acceptance condition* specified by Ω, i.e., the largest priority occurring infinitely often along the branch is even. The language accepted by an apta $\mathcal{A}$, $L(\mathcal{A})$, is the set of all $\mathcal{P}(\Sigma)$-trees T for which there exists an accepting run R of $\mathcal{A}$ on T.

The emptiness problem for apta, i.e., deciding whether $L(\mathcal{A}) = \emptyset$, is in ExpTime; the complement automaton which accepts the complement language $\mathrm{Tree}(\Sigma) \setminus L(\mathcal{A})$ can be constructed in linear time [11]. Note that, instead of only the transition conditions mentioned above, one could allow for complex transition conditions like $\Box(q_1 \wedge \neg B) \vee q_2$. Automata with complex transition conditions can be transformed into equivalent automata using only simple transition conditions by introducing new states for each subformula of the transition condition [11].

Example 2. Let $\mathcal{A}_{\mathrm{ex}} = (\Sigma, Q, q_0, \delta, \Omega)$ be an apta with alphabet $\Sigma = \{A, B\}$, with states $Q = \{q_0, \ldots, q_6\}$, initial state q_0, transition function δ as given in Figure 1, and priority function Ω with $\Omega(q) = 1$ for all $q \in Q$.

Transition function δ

$\delta(q_0) = q_1 \vee q_2$

$\delta(q_1) = B$

$\delta(q_2) = q_3 \wedge q_4 \wedge q_5$

$\delta(q_3) = A$

$\delta(q_4) = \Box q_0$

$\delta(q_5) = \Diamond q_6$

$\delta(q_6) = \mathsf{true}$

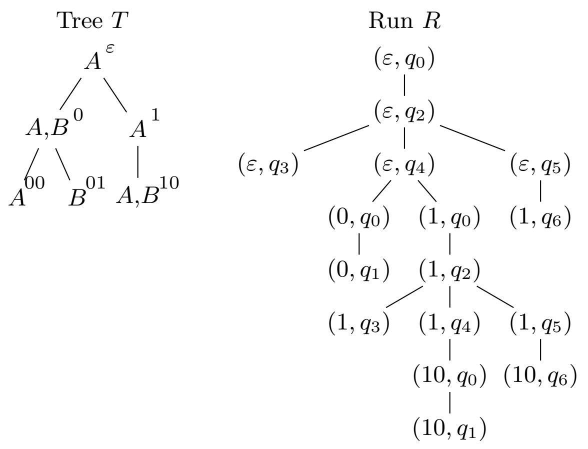

Fig. 1. Transition function δ, $\mathcal{P}(\Sigma)$-tree T, and accepting run R of $\mathcal{A}_{\text{ex}}$ on T.

This automaton accepts only trees where the root label contains B (state q_1), or it is labeled with A and all of its successors (at least one) are again of this form. Since the parity function prohibits infinite paths in the run (though not in the input tree), $\mathcal{A}_{\text{ex}}$ accepts exactly those trees where all paths start with nodes labeled with A until eventually a node with a label containing B is encountered. Figure 1 shows such a tree T and an accepting run R of $\mathcal{A}_{\text{ex}}$ on T.

We now show how to construct an automaton that decides concept satisfiability in $\mathcal{ALC}$. Given a TBox $\mathcal{T}$ and a concept C, the idea underlying this approach is that the constructed automaton will accept exactly the tree models of $\mathcal{T}$ for which the root is an instance of C. Note that the trees introduced above do not have labeled edges, while interpretations do. To overcome this, we push role names into the labels of the children. Thus, the alphabet Σ consists of all concept and role names of C and $C_{\mathcal{T}}$ (called the signature, sig). The automaton contains a state for each subconcept of C and $\mathcal{T}$, denoted sub(C) and sub$(C_{\mathcal{T}})$, which are used to simulate the semantics of $\mathcal{ALC}$. Cycles in $\mathcal{T}$ can enforce infinite tree models; infinite paths are always accepting if they satisfy the axioms in $\mathcal{T}$.

Definition 3. *Let $\mathcal{T}$ be an $\mathcal{ALC}$-TBox of the form $\{\top \sqsubseteq C_{\mathcal{T}}\}$ and C an $\mathcal{ALC}$-concept with both C and $C_{\mathcal{T}}$ in negation normal form. We define the automaton $\mathcal{A}_{C,\mathcal{T}} = (\Sigma, Q, q_0, \delta, \Omega)$ as follows:*

- *$\Sigma = sig(C) \cup sig(C_{\mathcal{T}})$ and $\Omega(q) = 0$ for all $q \in Q$,*
- *$Q = \{q_D \mid D \in sub(C) \cup sub(C_{\mathcal{T}})\} \cup \{q_r, q_{\neg r} \mid r \in sig(C) \cup sig(C_{\mathcal{T}})\} \cup \{q_0, q_{\mathcal{T}}\}$,*
- *the transition function δ is defined as follows (where $\sigma \in N_C \cup N_R$):*

$$\begin{aligned} \delta(q_0) &= q_C \wedge q_{\mathcal{T}} & \delta(q_{\mathcal{T}}) &= q_{C_{\mathcal{T}}} \wedge \Box q_{\mathcal{T}} \\ \delta(q_\sigma) &= \sigma & \delta(q_{\neg\sigma}) &= \neg\sigma \\ \delta(q_{C_1 \sqcap C_2}) &= q_{C_1} \wedge q_{C_2} & \delta(q_{C_1 \sqcup C_2}) &= q_{C_1} \vee q_{C_2} \\ \delta(q_{\exists r.C}) &= \Diamond(q_r \wedge q_C) & \delta(q_{\forall r.C}) &= \Box(q_{\neg r} \vee q_C) \end{aligned}$$

The proof of the following proposition is similar to the one in [7, pp. 59–62]. It relies on the fact that any tree with accepting run can be interpreted as a model of the TBox with the root being an instance of C, and any model of the TBox can be unraveled into a tree for which an accepting run can be inductively constructed.

Proposition 4. *Given an $\mathcal{ALC}$-TBox $\mathcal{T}$ and an $\mathcal{ALC}$-concept C, the concept C is satisfiable w.r.t. $\mathcal{T}$ iff $L(\mathcal{A}_{C,\mathcal{T}}) \neq \emptyset$.*

Since the automaton $\mathcal{A}_{C,\mathcal{T}}$ is polynomial in the size of the TBox $\mathcal{T}$ and the concept C, this approach yields an ExpTime algorithm for concept satisfiability, which is worst-case optimal [3].

3 Prototypes and Weighted Tree Automata

In general, a prototype is some kind of structure that can be compared to elements of an interpretation, distinguishing elements that are closer (more similar or related) to the prototype from elements that are further away (dissimilar or different). More specifically, one may view a prototype as a function that assigns to each element a distance value from the focal point, where small distances correspond to similar elements, and large distances to dissimilar elements.

Definition 5. *A* prototype distance function *(pdf) d is a function that assigns to each element e of an interpretation $\mathcal{I}$ a distance value $d_{\mathcal{I}}(e) \in \mathbb{N}$. The constructor $P_{\sim n}(d)$ for a threshold $n \in \mathbb{N}$ is interpreted in an interpretation $\mathcal{I}$ as the set of all elements $e \in \Delta^{\mathcal{I}}$ such that $d_{\mathcal{I}}(e) \sim n$, for $\sim \in \{<, \leq, >, \geq\}$. If D is a set of pdfs, we use $\mathcal{ALCP}(D)$ to denote the Description Logic $\mathcal{ALC}$ extended by the prototype constructor for pdfs from D.*

As explained before, we will use weighted alternating tree automata to define pdfs. These automata can express distance functions from trees (in our case, tree-shaped pointed interpretations[1]) to the non-negative integers $\mathbb{N}$. By unraveling pointed interpretations we can extend this to a function from arbitrary pointed interpretations to $\mathbb{N}$, i.e., a prototype distance function.

The main idea behind the use of weighted automata to describe pdfs is that the automaton can punish a pointed interpretation by increasing the distance value whenever a feature described by the automaton is not as expected. For example, the automaton can require the current node to be labeled with the concept name Cup, and increase the distance by some number if this is not the case. Using this idea, the most natural interpretation of the transition conditions in the weighted setting is as follows: $q_1 \wedge q_2$ will compute the sum of the distances for q_1 and q_2 (both features should be present), $\vee$ will be interpreted as the minimum (one of the feature should be present), $\exists$ will also be interpreted as the minimum (one of the successors should have the feature, i.e., we choose the best

[1] Recall that a pointed interpretation is an interpretation together with an element of the interpretation domain.

one); and $\forall$ will be interpreted as the maximum (all successors should have the feature; if not, we take the distance of the worst).

A weighted alternating parity tree automaton is nearly the same as in the unweighted case, with the exception that the transition function may also contain non-negative integers. Given an alphabet Σ and a set of states Q, a weighted transition condition $\mathrm{wTC}(\Sigma, Q)$ is one of the following: $n \in \mathbb{N}$; σ or $\neg\sigma$ for $\sigma \in \Sigma$; $q_1 \wedge q_2$ or $q_1 \vee q_2$ for $q_1, q_2 \in Q$; or $\Box q$ or $\Diamond q$ for $q \in Q$.

Definition 6 (weighted alternating parity tree automaton). *A weighted alternating parity tree automaton (wapta) $\mathcal{A}$ working on $\mathcal{P}(\Sigma)$-trees is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \delta, \Omega)$, where 1. Σ is a finite alphabet; 2. Q is a finite set of states and $q_0 \in Q$ is the initial state; 3. $\delta : Q \to wTC(\Sigma, Q)$ is the transition function; and 4. $\Omega : Q \to \mathbb{N}$ is the priority function.*

Runs are defined as in the unweighted case, where nodes labeled with a state for which the transition function yields a number do not need to satisfy any additional conditions, they can be leafs in the run. In order to define the behavior of such a weighted automaton on a tree, we need to define the $\Box$-fixation of a run, which basically chooses for a $\Box$-operator a single successor node (instead of all of them). Given a run R, a $\Box$-fixation is a tree R' with $\mathrm{dom}_{R'} \subseteq \mathrm{dom}_R$, which can be obtained from R as follows: starting with the root, we keep all the successors for nodes where the transition function does not yield a box; for nodes u labeled with a state q for which the transition function is of the form $\delta(q) = \Box q'$, the $\Box$-fixation R' keeps at most one successor $ui \in \mathrm{dom}_R$. All nodes $u \in \mathrm{dom}_{R'}$ have the same label $R'(u) = R(u)$ as in R.

Then, we can define the behavior of the automaton as a function $\|\mathcal{A}\| : \mathrm{Tree}(\mathcal{P}(\Sigma)) \to \mathbb{N}$. The weight of a $\Box$-fixation R' of a run R is defined as

$$\mathrm{weight}_{\mathcal{A}}(R') = \sum_{u \in \mathrm{dom}_{R'}, R'(u)=(d,q), \delta(q,T(u))=n \in \mathbb{N}} n.$$

Note that this (possibly infinite) sum is well-defined: If infinitely many values $n > 0$ occur in R', the weight of R' is ∞; otherwise it is the finite sum of all weights in R'. The weight of a run R on T is $\mathrm{weight}_{\mathcal{A}}(R) = \sup_{R' \ \Box\text{-fixation of } R} \mathrm{weight}_{\mathcal{A}}(R')$, and the behavior of $\mathcal{A}$ is $\|\mathcal{A}\|(T) = \min_{R \text{ accepting run on } T} \mathrm{weight}_{\mathcal{A}}(R)$.

Constructions of prototype automata In the following we will give a concrete example of how a weighted automaton can be constructed from an $\mathcal{ALC}$-concept. Recall from the introduction that a prototypical cup is a small container with handles, which can hold liquids and is made of plastic or porcelain. We can express this as an $\mathcal{ALC}$-concept:

$$\mathsf{Container} \sqcap \mathsf{Small} \sqcap \exists \mathsf{hasPart}.\mathsf{Handle} \sqcap \forall \mathsf{holds}.\mathsf{Liquid} \sqcap \forall \mathsf{material}.(\mathsf{Glass} \sqcup \mathsf{Porcelain})$$

This concept can directly be translated into a complex transition condition for an alternating tree automaton:

$$\begin{aligned}&\mathsf{Container} \wedge \mathsf{Small} \wedge \Diamond(\mathsf{hasPart} \wedge \mathsf{Handle})\\ &\quad \wedge \Box(\neg\mathsf{holds} \vee \mathsf{Liquid}) \wedge \Box(\neg\mathsf{material} \vee (\mathsf{Glass} \vee \mathsf{Porcelain}))\end{aligned}$$

Now, we can add weights in order to punish missing features:

$$(\mathsf{Container} \vee 3) \wedge (\mathsf{Small} \vee 1) \wedge (\Diamond(\mathsf{hasPart} \wedge \mathsf{Handle}) \vee 1)$$
$$\wedge \Box(\neg\mathsf{holds} \vee (\mathsf{Liquid} \vee 2)) \wedge \Box(\neg\mathsf{material} \vee ((\mathsf{Glass} \vee 1) \vee (\mathsf{Porcelain} \vee 1)))$$

The meaning of this weighted transition condition is as follows: If an element is not a container, it will be punished with a weight of 3 since there cannot be a run that uses the option Container at the root. Otherwise, there is such a run, which does not contribute a weight. Accordingly, the absence of the feature small is punished with weight 1. If the cup does not have a successor that is labeled with both hasPart and Handle, then a weight of 1 is added. Finally, if there is a material-successor that is not labeled with Glass or Porcelain, then this is punished with weight 1. If the cup does not have any material-successors, or all of them are glass or porcelain, no weight is added. Similarly for holding only liquids. In general, choosing the weights appropriately allows us to punish the absence of different features by different values.

For universal restrictions, the weights of several offending successors are not added up, but rather the supremum is taken. As a consequence, equivalent concepts may not yield equivalent wapta using this approach. For example, $\forall r.(A \sqcap B) \equiv \forall r.A \sqcap \forall r.B$, but the corresponding transition conditions after adding weights may lead to different results. However, one can argue that, when viewed as prototype descriptions, these two concept descriptions do actually encode different intentions. While in the first case we want to make sure that all r-successors are instance of A and B simultaneously (and pick the weight of the worst offender if there is one), in the second case we want to enforce both features separately, and punish for the worst offenders separately.

From the above example, it should be clear how a translation from $\mathcal{ALC}$-concepts to wapta works in general. On the other hand, one can also create prototypes from finite pointed interpretations. For this, one introduces a state for each element of the interpretation, and as transition condition for each state one simply conjoins all the concept names the element is instance of, negations of all concept names it is not instance of, and a $\Diamond$-transition for each successor in the interpretation, labeled with both the role name and the state of the successor-element. If one also introduces a $\Box$-transition with a disjunction of all possible successor-states and adds positive weights as in the above example, this weighted automaton will only give distance 0 to pointed interpretations that are bisimilar to the prototypical interpretation, and otherwise punish each difference by increasing the distance accordingly.

4 Reasoning with Prototype Automata in $\mathcal{ALC}$

To reason in $\mathcal{ALC}$ with prototypes, we have to achieve two things: First, for each prototype constructor $P_{\leq n}(\mathcal{A})$, we have to transform the wapta $\mathcal{A}$ into an unweighted automaton that accept exactly those trees T for which $\|\mathcal{A}\|(T) \leq n$. Then we need to combine the alternating tree automaton $\mathcal{A}_{C,\mathcal{T}}$ from Section 2 with the unweighted automata for the prototypes such that the resulting automaton accepts exactly the tree models of C w.r.t. $\mathcal{T}$. An emptiness test can then be used to decide (un-)satisfiability.

Cut-point automata Given a weighted alternating parity tree automaton $\mathcal{A}$ and a threshold value $n \in \mathbb{N}$, we want to construct an unweighted automaton $\mathcal{A}_{\leq n}$ that accepts exactly the cut-point language, i.e. $L(\mathcal{A}_{\leq n}) = \{T \in \text{Tree}(\mathcal{P}(\Sigma)) \mid \|\mathcal{A}\|(T) \leq n\}$. In this cut-point automaton, each state needs to keep track of both the weight and the current state of the corresponding weighted automaton. However, instead of tracking the weight that has already been accumulated, it needs to track the weight that the automaton is still allowed to spend. The reason for this is that for trees each state can have multiple successors, and thus we have to budget the allowed weight for each of the successors so that the sum is not greater than the threshold.

Definition 7. *Given a wapta $\mathcal{A} = (\Sigma, Q, q_0, \delta, \Omega)$, the cut-point automaton $\mathcal{A}_{\leq n} = (\Sigma, Q', q'_0, \delta', \Omega')$ for the threshold $n \in \mathbb{N}$ is an apta defined as follows:*

$$Q' = \{(q,i) \in Q \times \mathbb{N} \mid i \leq n\} \cup \{q'_0\} \quad \textit{and} \quad \Omega'((q,i)) = \Omega(q)$$

$$\delta'(q'_0) = \bigvee_{0 \leq i \leq n} (q_0, i) \qquad \delta'((q,i)) = \delta(q) \;\; \textit{if}\; \delta(q) = \sigma, \neg\sigma$$

$$\delta'((q,i)) = \textsf{true} \;\; \textit{if}\; \delta(q) = j \leq i \qquad \delta'((q,i)) = \textsf{false} \;\; \textit{if}\; \delta(q) = j > i$$

$$\delta'((q,i)) = \Diamond(q',i) \;\; \textit{if}\; \delta(q) = \Diamond q' \qquad \delta'((q,i)) = \Box(q',i) \;\; \textit{if}\; \delta(q) = \Box q'$$

$$\delta'((q,i)) = (q_1,i) \vee (q_2,i) \;\; \textit{if}\; \delta(q) = q_1 \vee q_2$$

$$\delta'((q,i)) = \bigvee_{0 \leq j \leq i} (q_1,j) \wedge (q_2, i-j) \;\; \textit{if}\; \delta(q) = q_1 \wedge q_2$$

Proposition 8. *Let $\mathcal{A}$ be a wapta and $\mathcal{A}_{\leq n}$ the cut-point automaton derived from $\mathcal{A}$ using the threshold $n \in \mathbb{N}$. Then $\mathcal{A}_{\leq n}$ accepts the cut-point language, i.e., $L(\mathcal{A}_{\leq n}) = \{T \in \mathit{Tree}(\mathcal{P}(\Sigma)) \mid \|\mathcal{A}\|(T) \leq n\}$.*

Proof (sketch). We have to prove both directions. Given a tree $T \in L(\mathcal{A}_{\leq n})$, and an accepting run R of $\mathcal{A}_{\leq n}$ on T, we can construct a run R' of $\mathcal{A}$ on T by removing all weights from the labels of R. By induction on the weight i, we can then show that whenever we have $R(u) = (v, (q,i))$ for some node $u \in \text{dom}_R$, all $\Box$-fixations of R' starting from u will have a weight at most i. This follows from the claim that the sum of the weights of the children of a node v is never larger than the weight of v itself for all $\Box$-fixations. Since the first successor of the root of R is labeled with $R(0) = (\varepsilon, (q_0, n))$, this means that $\text{weight}_{\mathcal{A}}(R') \leq n$.

Similarly, if we have a tree $T \in \text{Tree}(\mathcal{P}(\Sigma))$ with $\|\mathcal{A}\|(T) \leq n$, and a run R of $\mathcal{A}$ on T with $\text{weight}_{\mathcal{A}}(R) \leq n$, we can construct a run R' of $\mathcal{A}_{\leq n}$ on T by setting $R'(u) = (v, (q,i))$ where $R(u) = (v,q)$ and i is the weight assigned by $\mathcal{A}$ to the subtree of R rooted at u, starting in state q. It can then be shown that the run R' obtained this way is an accepting run of $\mathcal{A}_{\leq n}$ on T. $\Box$

The cut-point automaton $\mathcal{A}_{\leq n}$ has $O(n \cdot q)$ states, where q is the number of states of the weighted automaton $\mathcal{A}$. Thus, if n is encoded in unary, this construction is polynomial, otherwise it is exponential.

Combined reasoning using alternating automata We want to combine the cut-point automata constructed from prototype concepts with the automaton from Definition 3 in order to decide the concept satisfiability problem in $\mathcal{ALCP}$(wapta); more specifically, we want to construct an automaton $\mathcal{A}$ that accepts all those (tree-shaped) pointed interpretations that are instances of an $\mathcal{ALCP}$(wapta)-concept w.r.t. an $\mathcal{ALCP}$(wapta)-TBox.

For $\mathcal{ALCP}$(wapta)-concepts, one can again define a normal form. This extends the negation normal form used in Section 2 by requiring that prototype constructors occur only in the form $P_{\leq n}(\mathcal{A})$, possibly negated. For example, one can transform $P_{\geq n}(\mathcal{A})$ for $n \geq 1$ into negation normal form by replacing it with $\neg P_{\leq n-1}(\mathcal{A})$; $P_{\geq 0}(\mathcal{A})$ can be replaced by $\top$. The set of subconcepts now contains such prototype concepts as well.

In case a prototype constructor occurs negated, the complement automaton $\bar{\mathcal{A}}$ for a cut-point automaton $\mathcal{A}$ can be constructed in linear time, by exchanging true and false, $\vee$ and $\wedge$, $\Box$ and $\Diamond$, and σ and $\neg\sigma$ for all $\sigma \in \Sigma$ in all transition conditions, as well as adding one to the priority of all states [11].

Definition 9. *Let $\mathcal{T}$ be an $\mathcal{ALCP}$(wapta)-TBox of the form $\{\top \sqsubseteq C_\mathcal{T}\}$ and C an $\mathcal{ALCP}$(wapta)-concept, with both C and $C_\mathcal{T}$ in negation normal form, and let $\mathcal{A}_{i,\leq n}$ be the cut-point automaton of the wapta $\mathcal{A}_i$ for each prototype constructor $P_{\leq n}(\mathcal{A}_i)$ occurring in C or $\mathcal{T}$.*

The apta $A_{\mathcal{P},C,\mathcal{T}}$ is the disjoint union of $\mathcal{A}_{C,\mathcal{T}}$ from Definition 3, all automata $\mathcal{A}_{i,\leq n}$ for prototypes $P_{\leq n}(\mathcal{A}_i)$ occurring in C or $C_\mathcal{T}$, and all automata $\bar{\mathcal{A}}_{i,\leq n}$ for negated prototypes $\neg P_{\leq n}(\mathcal{A}_i)$ occurring in C or $C_\mathcal{T}$, such that the transition function additionally is defined for subconcepts of the form $P_{\leq n}(\mathcal{A}_i)$ and $\neg P_{\leq n}(\mathcal{A}_i)$ as follows:

$$\delta(q_{P_{\leq n}(\mathcal{A}_i)}) = q_i \text{ where } q_i \text{ is the initial state of } \mathcal{A}_{i,\leq n}$$
$$\delta(q_{\neg P_{\leq n}(\mathcal{A}_i)}) = q_i \text{ where } q_i \text{ is the initial state of } \bar{\mathcal{A}}_{i,\leq n}$$

The following theorem is an easy consequence of Proposition 4 and Proposition 8.

Theorem 10. *Given an $\mathcal{ALCP}$(wapta)-TBox $\mathcal{T}$ and an $\mathcal{ALCP}$(wapta)-concept C, the concept C is satisfiable w.r.t. $\mathcal{T}$ iff $L(\mathcal{A}_{\mathcal{P},C,\mathcal{T}}) \neq \emptyset$.*

Because of the size of the cut-point automata and the ExpTime-emptiness test for alternating tree automata, concept satisfiability can thus be deciding in ExpTime if the numbers are given in unary. This is worst-case optimal. If the numbers are given in binary, the complexity of the algorithm increases to 2ExpTime. It is an open problem whether this second exponential blowup can be avoided.

5 Conclusions

We have introduced an extension to Description Logics that allows to define prototypes and reason over them. In particular, we have introduced the prototype constructors $P_{\sim n}(d)$ that are interpreted as the set of all elements of the interpretation with distance $\sim n$ according to the prototype distance functions d. We have shown that pdfs can be defined using waptas, and that reasoning in $\mathcal{ALCP}$(wapta) has he same complexity as reasoning in $\mathcal{ALC}$ (if the threshold numbers n are coded in unary).

Of course, this approach has some limitations. As mentioned in Section 3, the pdfs obtained through a straightforward translation of $\mathcal{ALC}$-concepts into waptas are not equivalence invariant. This is due to the fact that we use the supremum rather than the sum to combine the weights obtained from different $\Box$-fixations. However, replacing supremum by sum has the disadvantage that the cut-point language need no longer be recognizable by an apta. We conjecture that in that case, the cut-point language can actually be accepted by a graded apta [4], but the construction to be developed would definitely be considerably more complex than the one used in this paper. More generally, one could of course also look at weighted automata using other domains for weights and other operations combining them.

Finally, we are interested in adding prototypes to other DLs. Since prototypes can be used to express negation, considering less expressive DLs does not make sense. But adding nominals and quantified number restrictions would be interesting, as would be considering the instance problem and answering conjunctive queries.

References

1. Baader, F.: Description logics. In: Reasoning Web: Semantic Technologies for Information Systems, 5th International Summer School, LNCS, vol. 5689, pp. 1–39. Springer (2009)
2. Baader, F., Brewka, G., Fernandez Gil, O.: Adding threshold concepts to the description logic $\mathcal{EL}$. In: Proc. FroCoS 2015. LNCS, vol. 9322, pp. 33–48. Springer (2015)
3. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
4. Bonatti, P.A., Lutz, C., Murano, A., Vardi, M.Y.: The complexity of enriched μ-calculi. In: Proc. ICALP 2006. LNCS, vol. 4052, pp. 540–551. Springer (2006)
5. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata. Springer (2009)
6. Gärdenfors, P.: Conceptual spaces - the geometry of thought. MIT Press (2000)
7. Hladik, J.: To and Fro Between Tableaus and Automata for Description Logics. Dissertation, TU Dresden, Germany (2007), http://www.qucosa.de/fileadmin/data/qucosa/documents/846/1201792812059-1908.pdf
8. Labov, W.: The boundaries of words and their meanings. In: Bailey, C.J.N., Shuy, R.W. (eds.) New ways of analyzing variation in English. Georgetown University Press (1973)
9. Rosch, E., Lloyd, B.B.: Cognition and Categorization. L. Erlbaum Associates (1978)
10. Sattler, U., Vardi, M.Y.: The hybrid μ-calculus. In: Proc. IJCAR 2001. LNCS, vol. 2083, pp. 76–91. Springer (2001)
11. Wilke, T.: Alternating tree automata, parity games, and modal μ-calculus. Bull. Belg. Math. Soc. 8, 359–391 (2001)