Description Logics with Concrete Domains and Functional Dependencies

Carsten Lutz and **Maja Miličić**[1]

Abstract. Description Logics (DLs) with concrete domains are a useful tool in many applications. To further enhance the expressive power of such DLs, it has been proposed to add database-style key constraints. Up to now, however, only uniqueness constraints have been considered in this context, thus neglecting the second fundamental family of key constraints: functional dependencies. In this paper, we consider the basic DL with concrete domains $\mathcal{ALC}(\mathcal{D})$, extend it with functional dependencies, and analyze the impact of this extension on the decidability and complexity of reasoning. Though intuitively the expressivity of functional dependencies seems weaker than that of uniqueness constraints, we are able to show that the former have a similarly severe impact on the computational properties: reasoning is undecidable in the general case, and NEXPTIME-complete in some slightly restricted variants of our logic.

1 Introduction

Description Logics (DLs) are a family of logic-based knowledge representation formalisms designed to represent and reason about conceptual knowledge [4]. In recent application areas of DLs such as the semantic web and reasoning about database schemas, the integration of so-called concrete domains (or, synonymously, concrete datatypes) has turned out to be a crucial task. A concrete domain is provided by a set (e.g. the integers, reals, or strings), and fixed predicates on this set (e.g. $<$, $+$, prime, and is-prefix) [2, 10]. In the context of the semantic web, DLs with concrete domains thus allow to formulate ontologies that refer to concrete qualities of objects such as their size, weight, temperature, etc. [3]. In reasoning about database schemas such as ER and UML diagrams, concrete domains allow to capture integrity constraints on numerical data [12].

It has recently been suggested that the expressive power of DLs with concrete domains can be further improved by adding database-style key constraints [15]. The most important such constraints are uniqueness constraints and functional dependencies, see e.g. [1]. The former allow to describe a set of properties that uniquely determine the identity of an object, such as in "Americans are uniquely identified by their social security number". In contrast, functional dependencies allow to express that the value of a property is determined by a set of properties, e.g. as in "all books with the same ISBN number have the same title". Clearly, both kinds of key constraints are very useful in the afore mentioned application areas.

The analysis of DLs with concrete domains and key constraints performed in [15] revealed that *uniqueness* constraints can have a severe impact on computational complexity: if they are added to the basic DL with concrete domains $\mathcal{ALC}(\mathcal{D})$, then the complexity of reasoning jumps from PSPACE-complete to NEXPTIME-complete or even to undecidable—depending on the variant of uniqueness constraints admitted. To the best of our knowledge, however, the result of adding the equally natural and important functional dependencies to DLs with concrete domains has never been investigated (though this has been done for DLs *without* concrete domains, c.f. [6, 7, 9, 18]).

In this paper, we extend $\mathcal{ALC}(\mathcal{D})$ with functional dependencies such as

$$(\mathsf{isbn}\ \text{skeyfor}\ \mathsf{Book}, \mathsf{title}),$$

which states that all instances of the class Book that share the same isbn also share the same title. Formally, both isbn and title are partial functions connecting logical objects with elements of the concrete domain, say strings. The purpose of this paper is then to *analyze the impact on decidability and complexity of adding functional dependencies to description logics with concrete domains*. Intuitively, the expressive power of functional dependencies is much weaker than that of the uniqueness constraints considered in [15]: while the latter enable the simulation of nominals (as known from modal logic) and thus have a considerable impact on the model theory of the underlying logic, the former merely allow to state some constraints on concrete data that have no direct effect on the logical part of the DL. Rather surprisingly, we are nevertheless able to show that the effect of adding functional dependencies to DLs with concrete domains is similarly dramatic as in the case of uniqueness constraints: reasoning in $\mathcal{ALC}(\mathcal{D})$ with full functional dependencies turns out to be undecidable, while a slight restriction on the structure of functional dependencies brings us down to NEXPTIME-completeness.

This paper is organized as follows: In Section 2, we introduce the DL $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ which augments $\mathcal{ALC}(\mathcal{D})$ with functional dependencies. In Section 3, we show that $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ is undecidable in the general case and NEXPTIME-hard if we restrict ourselves to a certain, natural class of functional dependencies. A matching NEXPTIME upper bound is established in Section 4, where we also exhibit a tableau algorithm for the restricted logic (which, for the first time, combines concrete domains with a blocking mechanism). Finally, we give concluding remarks in Section 5.

2 The Description Logic $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$

We start with formally introducing concrete domains.

Definition 1 (Concrete Domain) *A* concrete domain $\mathcal{D}$ *is a pair* $(\Delta_\mathcal{D}, \Phi_\mathcal{D})$*, where* $\Delta_\mathcal{D}$ *is a set and* $\Phi_\mathcal{D}$ *a set of predicate names. Each predicate name is associated with an arity* n *and an* n*-ary predicate* $P^\mathcal{D} \subseteq \Delta_\mathcal{D}^n$*. A* $\mathcal{D}$-conjunction *is a finite predicate conjunction of the form*

$$c = \bigwedge_{i<k} (x_0^{(i)}, \ldots, x_{n_i}^{(i)}) : P_i,$$

[1] Institute for Theoretical Computer Science, TU Dresden, Germany. Emails: {lutz, milicic}@tcs.inf.tu-dresden.de

where P_i is an n_i-ary predicate for $i < k$ and the $x_j^{(i)}$ are variables. A $\mathcal{D}$-conjunction c is satisfiable *if there exists a function δ mapping the variables in c to elements of $\Delta_\mathcal{D}$ such that $(\delta(x_0^{(i)}), \ldots, \delta(x_{n_i}^{(i)})) \in P_i^\mathcal{D}$ for each $i < k$. Such a function is called a* solution *for c.*

Based on concrete domains, we define the syntax of $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$.

Definition 2 ($\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ Syntax) *Let N_C, N_R and N_{cF} be pairwise disjoint and countably infinite sets of* concept names, role names, *and* concrete features. *Furthermore, we assume that N_R contains a countably infinite subset N_{aF} of* abstract features. *A* path *u is a composition $f_1 \cdots f_n g$ of n abstract features $f_1, \ldots, f_n$ ($n \geq 0$) and a concrete feature g. Let $\mathcal{D}$ be a concrete domain. The set of $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concepts is the smallest set such that*

- *every concept name is a concept*
- *if C and D are concepts, R is a role name, g is a concrete feature, $u_1 \ldots u_n$ are paths, and $P \in \Phi_\mathcal{D}$ is a predicate of arity n, then the following expressions are also concepts:*

  $\neg C,\ C \sqcap D,\ C \sqcup D,\ \exists R.C,\ \forall R.C,\ \exists u_1, \ldots, u_n.P,$ *and* $g\!\uparrow$.

A functional dependency *is an expression*

$$(u_1, \ldots, u_k \textit{ wkeyfor } C, u) \quad \textit{(weak fun. dep.)}$$
$$\textit{or} \quad (u_1, \ldots, u_k \textit{ skeyfor } C, u) \quad \textit{(strong fun. dep.)}$$

where $u_1 \ldots u_k$ ($k \geq 1$) and u are paths and C is a concept. A finite set of functional dependencies is called key box. *If a key box contains only weak functional dependencies, it is called* weak.

We use $\top$ as abbreviation for an arbitrary propositional tautology. Intuitively, the two kinds of functional dependencies differ as follows: if there are instances x and y of C that have the same values for the paths $u_1, \ldots, u_n$, then weak f.d.s enforce the u-values of x and y to be identical only if *both* x and y are already known to have a value for u. In contrast, a strong functional dependency would enforce y to have a u-value if x has one (indeed, they have to be the same), and vice versa.

Definition 3 ($\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ Semantics) *An* interpretation *$\mathcal{I}$ is a pair $(\Delta_\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta_\mathcal{I}$ is a non-empty set, called the* domain, *and $\cdot^\mathcal{I}$ is the* interpretation function. *The interpretation function maps each concept name C to a subset $C^\mathcal{I}$ of $\Delta_\mathcal{I}$, each role name R to a subset $R^\mathcal{I}$ of $\Delta_\mathcal{I} \times \Delta_\mathcal{I}$, each abstract feature f to a partial function $f^\mathcal{I}$ from $\Delta_\mathcal{I}$ to $\Delta_\mathcal{I}$, and each concrete feature g to a partial function $g^\mathcal{I}$ from $\Delta_\mathcal{I}$ to $\Delta_\mathcal{D}$.*

If $u = f_1 \cdots f_n g$ is a path and $d \in \Delta_\mathcal{I}$, then $u^\mathcal{I}(d)$ is defined as $g^\mathcal{I}(f_n^\mathcal{I} \cdots (f_1^\mathcal{I}(d)) \cdots)$. The interpretation function is extended to arbitrary concepts as follows:

$$\begin{aligned}
(\neg C)^\mathcal{I} &:= \Delta_\mathcal{I} \setminus C^\mathcal{I} \\
(C \sqcap D)^\mathcal{I} &:= C^\mathcal{I} \cap D^\mathcal{I} \qquad (C \sqcup D)^\mathcal{I} := C^\mathcal{I} \cup D^\mathcal{I} \\
(\exists R.C)^\mathcal{I} &:= \{d \in \Delta_\mathcal{I} \mid \exists e : (d, e) \in R^\mathcal{I} \wedge e \in C^\mathcal{I}\} \\
(\forall R.C)^\mathcal{I} &:= \{d \in \Delta_\mathcal{I} \mid \forall e : (d, e) \in R^\mathcal{I} \rightarrow e \in C^\mathcal{I}\} \\
(\exists u_1, \ldots, u_n.P)^\mathcal{I} &:= \{d \in \Delta_\mathcal{I} \mid \exists x_1, \ldots, x_n \in \Delta_\mathcal{D} : \\
&\qquad u_i^\mathcal{I}(d) = x_i \textit{ and } (x_1, \ldots, x_n) \in P^\mathcal{D}\} \\
(g\!\uparrow)^\mathcal{I} &:= \{d \in \Delta_\mathcal{I} \mid g^\mathcal{I}(d) \textit{ undefined }\}.
\end{aligned}$$

An interpretation $\mathcal{I}$ is a model *of a concept C iff $C^\mathcal{I} \neq \emptyset$. We say that $\mathcal{I}$* satisfies *a weak functional dependency $(u_1, \ldots, u_k$ wkeyfor $C, u)$ if, for all $a, b \in C^\mathcal{I}$ with $u_i^\mathcal{I}(a) = u_i^\mathcal{I}(b)$ for $1 \leq i \leq n$ and $u^\mathcal{I}(a)$ and $u^\mathcal{I}(b)$ defined, we have $u^\mathcal{I}(a) = u^\mathcal{I}(b)$. $\mathcal{I}$* satisfies *a strong functional dependency $(u_1, \ldots, u_k$ skeyfor $C, u)$ if, for all $a, b \in C^\mathcal{I}$ with $u_i^\mathcal{I}(a) = u_i^\mathcal{I}(b)$ for $1 \leq i \leq n$ and $u^\mathcal{I}(a)$ defined, we have $u^\mathcal{I}(b)$ defined and $u^\mathcal{I}(a) = u^\mathcal{I}(b)$.*

$\mathcal{I}$ is a model *of a key box $\mathcal{K}$ iff $\mathcal{I}$ satisfies all functional dependencies in $\mathcal{K}$. A concept C* is satisfiable w.r.t. a key box *$\mathcal{K}$ iff C and $\mathcal{K}$ have a common model.*

Due to space limitations, we have to refer the reader to [16]. for examples of $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concepts and key boxes.

3 Lower Bounds

In this section, we prove undecidability of $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ as introduced in the previous section, and NEXPTIME-hardness of a variant of $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ that is obtained by putting a mild restriction on functional dependencies. Full proof details can be found in [16].

Undecidability of $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ is proved by reduction of the well-known *Post Correspondence Problem (PCP)* [17]. Recall that an instance of the PCP is given by a list $(\ell_1, r_1), \ldots, (\ell_k, r_k)$ of pairs of words over some alphabet Σ, and that a *solution* to such an instance is a non-empty sequence of integers $i_1, \ldots, i_n$ from the range $1, \ldots, k$ such that the "left concatenation" $\ell_{i_1} \cdots \ell_{i_n}$ is identical to the "right concatenation" $r_{i_1} \cdots r_{i_n}$. To reduce this problem to $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concept satisfiability w.r.t. key boxes, we use the concrete domain W defined in [13]: Δ_W is Σ^*, and there is a binary concatenation predicate conc_w for each $w \in \Sigma^*$ such that $(x, y) \in \mathsf{conc}_w^\mathsf{W}$ iff $y = xw$. Intuitively, the reduction works as follows: given a PCP instance P, we define a reduction concept C_P and *weak* key box $\mathcal{K}_P$ such that their common model $\mathcal{I}$ has the shape of a k-ary infinite tree. Each node represents a different sequence of indices $i_1 \cdots i_n$, i.e. a potential solution for P. Additionally, there are concrete features ℓ and r such that each node z representing $i_1 \cdots i_n$ satisfies $\ell^\mathcal{I}(z) = \ell_{i_1} \cdots \ell_{i_n}$ and $r^\mathcal{I}(z) = r_{i_1} \cdots r_{i_n}$. Finally, the definition of C_P ensures that $\ell^\mathcal{I}(z) \neq r^\mathcal{I}(z)$ for every node z, and thus no potential solution is a solution. Hence, P has a solution if and only if C_P is unsatisfiable w.r.t. $\mathcal{K}_P$. Note that, without key boxes, it is impossible to enforce an infinite tree as needed. In [13], it is shown that W itself is computationally very simple, i.e. that the satisfiability of W-conjunctions can be decided in PTIME.

Theorem 4 *There exists a concrete domain W such that satisfiability of W-conjunctions is in* PTIME *and $\mathcal{ALC}(\mathsf{W})^{\mathcal{FD}}$-concept satisfiability w.r.t. weak key boxes is undecidable.*

It is interesting to note that we do not need strong functional dependencies for the reduction. Moreover, we only need singleton key boxes whose functional dependency has only a single concrete feature on the left- and right-hand side. As discussed in [14], the concrete domain W can be replaced by more natural ones, e.g. $\Delta_\mathcal{D} = \mathbb{N}$ and predicates $=_0$, $=$, $\neq$, $+$, and $*$.

We now investigate a mild restriction of the logic $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ that is obtained by considering only a "safe" form of key boxes:

Definition 5 *A key box is called* safe *if none of the concepts inside functional dependencies in it has a subconcept of the form $\exists u_1, \ldots, u_n.P$.*

As safe key boxes suffice to model the simple example in the introduction as well as the more complex ones in [16], we believe that this restriction is not very severe. Also note that it is considerably more relaxed than the "Boolean" key boxes from [15]. As will be shown in Section 4, $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concept satisfiability w.r.t. safe key boxes is decidable in NEXPTIME. Here, we establish a matching

lower bound. This time, we reduce a NEXPTIME-complete variant of the *tiling problem* [5, 11], which is formulated as follows: a *domino system* $\mathcal{D} = (T, H, V)$ is given by a finite set of tile types T and horizontal and vertical matching relations $H, V \subseteq T \times T$. The task is to cover a $2^{n+1} \times 2^{n+1}$-torus (i.e., a rectangle whose parallel edges are "glued" together) with tiles such that adjacent tiles "match" according to H and V. For the reduction, it suffices to use a very simple concrete domain D such that $\Delta_{\mathsf{D}} = \{0, 1\}$ and $\Phi_{\mathcal{D}}$ contains predicates $=_0$ and $=_1$. The definition of the reduction concept ensures that its models have the shape of a binary tree such that, for every position (x, y) in the torus, there is a leaf of the tree that corresponds to position (x, y). This correspondence is established via the concrete domain D, which is used to binarily encode the x- and y-components of positions in the torus. In a similar way, we encode the tile types. The key box is then used to ensure that leaves corresponding to the same position are labeled identically. Once this is done, the matching conditions are easily enforced.

Theorem 6 *$\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concept satisfiability w.r.t. weak safe key boxes is* NEXPTIME*-hard if $\{0, 1\} \subseteq \Delta_{\mathcal{D}}$ and there are predicates $=_0$ and $=_1$.*

Again, we only need weak key boxes for the reduction and all paths in the employed functional dependencies are of length one (i.e. concrete features). However, this time we *do* need more than one functional dependency and more than one concrete feature on the left-hand side. As discussed in [16], these restrictions can be alleviated by going to slightly more powerful concrete domains.

4 A Tableau Algorithm for $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$

In this section, we use a tableau algorithm to show that $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concept satisfiability w.r.t. safe key boxes (c.f. Definition 5) is decidable. As a by-product of the algorithm, we also prove that this problem is in NEXPTIME.

In general, tableau algorithms for description logics are very popular because they can often be implemented in an efficient manner [8]. To decide whether a concept is satisfiable, tableau algorithms try to construct a model for it. To this end, they start with some initial data structure representing a fraction of a model, and then repeatedly apply completion rules that gradually augment the model. Eventually, either a contradiction is obtained or a contradiction-free (representation of a) model is found to which no more rules are applicable. The algorithm returns "unsatisfiable" in the former case and "satisfiable" in the latter.

Since we want the tableau algorithm to be independent of a specific concrete domain, we need a clean interface to a concrete domain reasoner whose job is to detect inconsistencies related to the concrete domain. Traditionally, this interface is established by requiring that satisfiability of $\mathcal{D}$-conjunctions is decidable (*admissibility* of $\mathcal{D}$ [2, 10]). However, due to the functional dependencies this is not enough: the concrete domain reasoner should also inform the tableau algorithm about which variables have to be mapped to the same value in order to satisfy a certain $\mathcal{D}$-conjunction. Thus, we employ the stronger condition of *key-admissibility* introduced in [15]. As discussed e.g. in [16], every admissible concrete domain which provides for an equality predicate is also key-admissible.

Definition 7 (key-admissible) *A concrete domain $\mathcal{D}$ is* key-admissible *iff it satisfies the following properties:*

1. *$\Phi_{\mathcal{D}}$ contains a name $\top_{\mathcal{D}}$ for $\Delta_{\mathcal{D}}$;*
2. *$\Phi_{\mathcal{D}}$ is closed under negation, i.e., for each n-ary predicate $P \in \Phi_{\mathcal{D}}$, there is a predicate $\overline{P} \in \Phi_{\mathcal{D}}$ of arity n such that $\overline{P}^{\mathcal{D}} = \Delta_{\mathcal{D}}^n \setminus P^{\mathcal{D}}$;*
3. *there exists an algorithm that takes as input a $\mathcal{D}$-conjunction c, returns* clash *if c is unsatisfiable, and otherwise non-deterministically outputs an equivalence relation $\sim$ on the set of variables V used in c such that there exists a solution δ for c with the following property: for all $v, v' \in V$*

$$\delta(v) = \delta(v') \text{ iff } v \sim v'.$$

Before presenting the tableau algorithm, we need some more prerequisites. A concept is in *negation normal form (NNF)* if negation occurs only in front of concept names. If $\mathcal{D}$ is a key-admissible concrete domain, then every $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concept can be converted into an equivalent one in NNF, see [16] for details. We use $\doteq\!C$ to denote the result of converting the concept $\neg C$ into NNF. A key box $\mathcal{K}$ is in NNF if all concepts occurring in functional dependencies in $\mathcal{K}$ are in NNF. From now on, we assume that all input concepts and key boxes are in NNF, and that the concrete domain $\mathcal{D}$ is key-admissible.

Let C be an $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concept and $\mathcal{K}$ a key box. We use $\mathsf{sub}(C)$ to denote the set of subconcepts of C and $\mathsf{con}(\mathcal{K})$ to denote the set of concepts appearing on the right-hand side of functional dependencies in $\mathcal{K}$. We use $\mathsf{cl}(C, \mathcal{K})$ as abbreviation for the set

$$\mathsf{sub}(C) \cup \{D, \doteq\!D \mid \text{there is an } E \in \mathsf{con}(\mathcal{K}) \text{ with } D \in \mathsf{sub}(E)\}.$$

Our algorithm works on completion trees, whose nodes represent elements of the interpretation domain. Due to the presence of concrete domains, trees have two types of nodes: abstract ones that represent individuals of the logic domain $\Delta_{\mathcal{I}}$, and concrete ones representing values of the concrete domain.

Definition 8 (Completion system) *Let O_a and O_c be disjoint and countably infinite sets of* abstract *and* concrete nodes. *A* completion tree *for an $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concept C and a key box $\mathcal{K}$ is a finite, labeled tree $T = (V_a, V_c, E, \mathcal{L})$ with nodes $V_a \cup V_c$, such that $V_a \subseteq O_a$, $V_c \subseteq O_c$, and all nodes from V_c are leaves. The tree is labeled as follows:*

1. *each node $a \in V_a$ is labeled with a subset $\mathcal{L}(a)$ of $\mathsf{cl}(C, \mathcal{K})$;*
2. *each edge $(a, b) \in E$ with $a, b \in V_a$ is labeled with a role name $\mathcal{L}(a, b)$ occurring in C or $\mathcal{K}$;*
3. *each edge $(a, x) \in E$ with $a \in V_a$ and $x \in V_c$ is labeled with a concrete feature $\mathcal{L}(a, x)$ occurring in C or $\mathcal{K}$*

A node $b \in V_a$ is an R-successor of a node $a \in V_a$ if $(a, b) \in E$ and $\mathcal{L}(a, b) = R$, while an $x \in V_c$ is a g-successor of a if $(a, x) \in E$ and $\mathcal{L}(a, x) = g$. The notion u-successor for a path u is defined in the obvious way.

A completion system *for an $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concept C and a key box $\mathcal{K}$ is a tuple $(T, \mathcal{P}, \sim)$ where*

- *$T = (V_a, V_c, E, \mathcal{L})$ is a completion tree for C and $\mathcal{K}$,*
- *$\mathcal{P}$ maps each $\mathcal{P} \in \Phi_{\mathcal{D}}$ of arity n in C to a subset of V_c^n,*
- *$\sim$ is an equivalence relation on V_c.*

Let $\mathcal{D}$ be a key-admissible concrete domain. To decide the satisfiability of an $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concept C_0 w.r.t. a safe key box $\mathcal{K}$, the tableau algorithm is started with the initial completion system $S_{C_0} := (T_{C_0}, \mathcal{P}_0, \emptyset)$, where the initial completion tree is

$$T_{C_0} := (\{a_0\}, \emptyset, \emptyset, \{a_0 \mapsto \{C_0\}\}),$$

and $\mathcal{P}_0$ maps each P occurring in C_0 to $\emptyset$.

We will define the completion rules after some prerequisites. Let us first introduce an operation that is used by completion rules to add new nodes to completion trees. The operation respects the functionality of abstract and concrete features.

Definition 9 ($\oplus$ Operation) *An abstract or concrete node is called* fresh *w.r.t. a completion tree T if it does not appear in T. Let $S = (T, \mathcal{P}, \sim)$ be a completion system with $T = (V_a, V_c, E, \mathcal{L})$. We use the following operations:*

- *$S \oplus aRb$ ($a \in V_a$, $b \in O_a$ fresh in T, $R \in N_R$) yields a completion system obtained from S in the following way:*
  - *if $R \notin N_{aF}$ or $R \in N_{aF}$ and a has no R-successors, then add b to V_a, (a, b) to E and set $\mathcal{L}(a, b) = R$, $\mathcal{L}(b) = \emptyset$.*
  - *if $R \in N_{aF}$ and there is a $c \in V_a$ such that $(a, c) \in E$ and $\mathcal{L}(a, c) = R$ then rename c in T with b.*
- *$S \oplus agx$ ($a \in V_a$, $x \in O_c$ fresh in T, $g \in N_{cF}$) yields a completion system obtained from S in the following way:*
  - *if a has no g-successors, then add x to V_c, (a, x) to E and set $\mathcal{L}(a, x) = g$;*
  - *if a has a g-successor y, then rename y in T, $\mathcal{P}$, and $\sim$ with x.*

Let $u = f_1 \cdots f_n g$ be a path. With $S \oplus aux$, where $a \in V_a$ and $x \in O_c$ is fresh in T, we denote the completion system obtained from S by taking distinct nodes $b_1, \ldots, b_n \in O_a$ which are fresh in T and setting $S' := S \oplus af_1b_1 \oplus \cdots + b_{n-1}f_nb_n \oplus b_ngx$.

Now we define what is meant by an obvious inconsistency.

Definition 10 (Clash) *Let $S = (T, \mathcal{P}, \sim)$ be a completion system for a concept C and a key box $\mathcal{K}$ with $T = (V_a, V_c, E, \mathcal{L})$. We say that S contains a* clash *iff*

1. *there is an $a \in V_a$ and an $A \in N_C$ such that $\{A, \neg A\} \subseteq \mathcal{L}(a)$,*
2. *there are $a \in V_a$ and $x \in V_c$ such that $g\uparrow \in \mathcal{L}(a)$ and x is a g-successor of a,*
3. *S is not* concrete domain satisfiable, *i.e. the following conjunction is not satisfiable:*

$$\zeta_S = \bigwedge_{P \text{ used in } C} \bigwedge_{(x_1,\ldots,x_n)\in\mathcal{P}(P)} P(x_1, \ldots, x_n) \wedge \bigwedge_{x\sim y} = (x, y)$$

In order to ensure the termination of the algorithm, we need a cycle detection mechanism. Informally, we detect nodes in the completion tree that are "similar" to one of their ancestors and "block" them, i.e. we do not apply completion rules to such nodes.

Definition 11 ($\approx$ relation, Blocking) *Let $S = (T, \mathcal{P}, \sim)$ be a completion system for a concept C_0 and a key box $\mathcal{K}$ with $T = (V_a, V_c, E, \mathcal{L})$. Let u be a path. We say that nodes $a, b \in V_a$* have similar u-successors *(written $a \approx_u b$) if the following holds:*

- *if a has a u-successor x, then b has a u-successor y and $x \sim y$;*
- *if b has a u-successor x, then a has a u-successor x and $x \sim y$.*

With $\mathsf{suff}(C_0, \mathcal{K})$ we denote the set of all suffixes of paths that appear in a $\exists u_1, \ldots, u_n.P \in \mathsf{sub}(C_0)$ or in a functional dependency (either on the left- and right-hand side) in the key box $\mathcal{K}$. We call abstract nodes a and b similar *(written $a \approx b$) if*

- *$\mathcal{L}(a) = \mathcal{L}(b)$; and*
- *$a \approx_u b$ for all $u \in \mathsf{suff}(C_0, \mathcal{K})$.*

An abstract node $a \in V_a$ is directly blocked *by its ancestor $b \in V_a$ if $a \approx b$. An abstract node is* blocked *if it or one of its ancestors is directly blocked.*

R$\sqcap$ if $C_1 \sqcap C_2 \in \mathcal{L}(a)$, a is unblocked, and $\{C_1, C_2\} \not\subseteq \mathcal{L}(a)$
then $\mathcal{L}(a) := \mathcal{L}(a) \cup \{C_1, C_2\}$

R$\sqcup$ if $C_1 \sqcup C_2 \in \mathcal{L}(a)$, a is unblocked, and $\{C_1, C_2\} \cap \mathcal{L}(a) = \emptyset$
then $\mathcal{L}(a) := \mathcal{L}(a) \cup \{C\}$ for some $C \in \{C_1, C_2\}$

R$\exists$ if $\exists R.C \in \mathcal{L}(a)$, a is unblocked, and there is no R-successor of a such that $C \in \mathcal{L}(b)$
then set $S := S \oplus aRb$ for a fresh $b \in O_a$ and $\mathcal{L}(b) := \{C\}$

R$\forall$ if $\forall R.C \in \mathcal{L}(a)$, a is unblocked, and b is an R-successor of a such that $C \notin \mathcal{L}(b)$
then set $\mathcal{L}(b) := \mathcal{L}(b) \cup \{C\}$

R$\exists_c$ if $\exists u_1, \ldots, u_n.P \in \mathcal{L}(a)$, a is unblocked, and there exist no $x_1, \ldots, x_n \in V_c$ such that x_i is a u_i-successor of a, $1 \le i \le n$, and $(x_1, \ldots, x_n) \in \mathcal{P}(P)$
then set $S := (S \oplus au_1x_1 \oplus \cdots \oplus au_nx_n)$, $x_1, \ldots, x_n \in O_C$ fresh and $\mathcal{P}(P) := \mathcal{P}(P) \cup \{(x_1, \ldots, x_n)\}$

Rch if $(u_1, \ldots, u_n$ keyfor $C, u) \in \mathcal{K}$ with "keyfor" either wkeyfor or skeyfor, and there exist $x_1, \ldots, x_n \in V_c$ such that x_i is an unblocked u_i-successor of a for $1 \le i \le n$, and $\{C, \doteq\neg C\} \cap \mathcal{L}(a) = \emptyset$
then set $\mathcal{L}(a) := \mathcal{L}(a) \cup \{D\}$ for some $D \in \{C, \doteq\neg C\}$

Rwk if $C \in \mathcal{L}(a) \cap \mathcal{L}(b)$, $(u_1, \ldots, u_n$ wkeyfor $C, u) \in \mathcal{K}$, a and b are unblocked, a has u_i-successor x_i, b has u_i-successor y_i, and $x_i \sim y_i$ for $1 \le i \le n$, there is a u-successor x of a and a u-successor y of b, and $(x, y) \notin \sim$
then set $\sim := (\sim \cup \{(x, y)\})^*$

Rsk if $C \in \mathcal{L}(a) \cap \mathcal{L}(b)$, $(u_1, \ldots, u_n$ skeyfor $C, u) \in \mathcal{K}$, a and b are unblocked, a has u_i-successor x_i, b has u_i-successor y_i, and $x_i \sim y_i$ for $1 \le i \le n$, there is a u-successor x of a, and there is no u-successor z of b such that $(x, z) \in \sim$
then set $S := S \oplus buy$ with $y \in O_c$ fresh and $\sim := (\sim \cup \{(x, y)\})^*$

R$\sim$ if $\sim' = \mathsf{check}(\zeta_S) \not\subseteq \sim$
then set $\sim := \sim'$

Figure 1. The completion rules.

An intuitive explanation of the blocking mechanism is given after we have introduced the completion rules, which are displayed in Figure 1. In the figure, we use check to refer to the function that computes a concrete equivalence for a given $\mathcal{D}$-conjunction (c.f. Point 3 of Definition 7). Moreover, ρ^* denotes the reflexive, symmetric, and transitive closure of the binary relation ρ.

Among the rules, there are two non-deterministic ones, namely R$\sqcup$ and Rch. The rules R$\sqcap$, R$\sqcup$, R$\exists$, R$\forall$, and R$\exists_c$ are variants of the corresponding rules from the classical algorithm for $\mathcal{ALC}(\mathcal{D})$-concept satisfiability [2]. The rules Rch, Rwk, and Rsk deal with key boxes and deserve some comments.

Rch is a so-called "choice" rule: if there is a functional dependency $(u_1, \ldots, u_n$ w/skeyfor $C, u) \in \mathcal{K}$ and there is an abstract node a with all appropriate u_i-successors, then the rule nondeterministically adds C or $\doteq\neg C$ to $\mathcal{L}(a)$. This is necessary since both possibilities may induce clashes after further rule applications that would otherwise go unnoticed. Observe that, without blocking, this rule can cause infinite runs of algorithm, e.g. due to a dependency $(u$ wkeyfor $\exists R.\top, u) \in \mathcal{K}$ and "bad guessing". The first part of the blocking condition deals with this problem.

The Rwk rule deals with weak functional dependencies. Suppose there is a $(u_1, \ldots, u_n$ wkeyfor $C, u) \in \mathcal{K}$ and two abstract nodes a

and b, both having successors for the paths $u, u_1, \ldots, u_n$, and whose u_i-successors are related via $\sim$, i.e. represent the same element of the concrete domain. Then the Rwk rule makes sure that the u-successors of a and b are also related by $\sim$.

Analogously, the Rsk rule deals with strong functional dependencies. The difference to Rwk is that it is applied even if b does not already have a u-successor. In this case, the necessary u-successor of b is created. This rule endangers the termination of the algorithm. To see this, consider satisfiability of

$$C_0 = \exists g.{=_0} \sqcap \exists (fg).{=_0} \quad \text{w.r.t.} \quad \mathcal{K} = \{(g \text{ skeyfor } \top, fg)\}.$$

Without blocking, applications of Rsk will generate an infinite f-chain such that each element has a g-successor that is zero. The second part of the blocking condition deals with this effect.

Finally, the R$\sim$ rule computes an update of the concrete equivalence $\sim$ by calling the check function with argument ζ_S as defined in Definition 10. This is necessary since the rule R$\exists_c$ adds new tuples into $\mathcal{P}(P)$, and the rules Rwk and Rsk add new tuples into $\sim$, thus modifying the $\mathcal{D}$-conjunction ζ_S. The update is performed *during* the run of the algorithm—in contrast to the original $\mathcal{ALC}(\mathcal{D})$ algorithm, where a single call to the concrete domain reasoner at the end of the computation is sufficient. The "interleaving" approach of our algorithm is essential since the equality of concrete nodes detected by the concrete domain reasoner can trigger further applications of the Rwk and Rsk rules.

The algorithm repeats the following two steps: first, it checks whether the completion system contains a clash, returning unsatisfiable if this is the case. Otherwise, it checks whether a completion rule is applicable. If no, then it returns satisfiable. If yes, then it applies the rule and starts over. Note that checking for clashes prior to rule application ensures that the function check(ζ_S) in R$\sim$ does not return clash.

In [16], it is proved that this algorithm terminates on any input, and that it is sound and complete.

Theorem 12 *If $\mathcal{D}$ is a key-admissible concrete domain, the tableau algorithm decides satisfiability of $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ concepts w.r.t. safe key boxes.*

Moreover, it follows from the soundness proof in [16] that $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ has a *bounded model property*: if an $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ concept C_0 is satisfiable w.r.t. a safe key box $\mathcal{K}$, then C_0 and $\mathcal{K}$ have a common model whose size is at most exponential in the size of C_0 and $\mathcal{K}$. This result almost immediately yields a NEXPTIME upper bound for $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ with safe key boxes if extended $\mathcal{D}$-satisfiability is in NP, i.e. if the (non-deterministic) algorithm for checking $\mathcal{D}$-satisfiability described in Definition 7 runs in polynomial time. Note that we say "almost immediately" since we have to ensure an appropriately-sized representation of concrete domain elements when "guessing" models of exponential size. More details can be found in [16].

Theorem 13 *If $\mathcal{D}$ is a key-admissible concrete domain and extended $\mathcal{D}$-satisfiability is in* NP*, then $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$-concept satisfiability w.r.t. safe key boxes is in* NEXPTIME.

5 Conclusion

In this paper, we have completed the investigation of description logics with concrete domains and key constraints that was begun in [15]. In particular, we have shown that the impact on computational complexity of adding functional dependencies is just as dramatic as for the seemingly more powerful uniqueness constraints. Still, we were able to come up with a tableau algorithm for $\mathcal{ALC}(\mathcal{D})^{\mathcal{FD}}$ with safe key boxes that should be amenable to known optimization techniques [8] and thus has the potential to be implemented efficiently. It should also be noted that, for the first time, we have combined full concrete domains (as introduced in [2]) with a blocking mechanism.

For future work, it would be interesting to combine both uniqueness and functional dependencies in a single DL with concrete domains. We conjecture that the upper bounds are preserved and that the more restricted Boolean key boxes in [15] can be replaced by our safe ones. It is also worthwhile to integrate functional dependencies into more expressive DLs such as $\mathcal{SHOQ}(\mathcal{D})$ or $\mathcal{ALC}(\mathcal{D})$ with (acyclic) TBoxes.

REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*, Addison-Wesley, (1995).

[2] F. Baader and P. Hanschke, 'A scheme for integrating concrete domains into concept languages', in *Proc. of the Twelfth Int. Joint Conf. on AI (IJCAI-91)*, pp. 452–457, Sydney, Australia, (1991).

[3] F. Baader, I. Horrocks, and U. Sattler, 'Description logics as ontology languages for the semantic web', in *Festschrift in honor of Jörg Siekmann*, LNAI. Springer-Verlag, (2003).

[4] F. Baader, D.L. McGuiness, D. Nardi, and Peter Patel-Schneider, *The Description Logic Handbook: Theory, implementation and applications*, Cambridge University Press, (2003).

[5] E. Börger, E. Grädel, and Y. Gurevich, *The Classical Decision Problem*, Perspectives in Mathematical Logic, Springer-Verlag, (1997).

[6] A. Borgida and G.E. Weddell, 'Adding uniqueness constraints to description logics (preliminary report)', in *Proc. of the 5th Int. Conf. on Deductive and Object-Oriented Databases (DOOD97)*, volume 1341 of *LNCS*, pp. 85–102. Springer, (1997).

[7] D. Calvanese, G. Giacomo, and M. Lenzerini, 'Identification constraints and functional dependencies in description logics', in *Proc. of the Seventeenth Int. Joint Conf. on AI (IJCAI'01)*, pp. 155–160. Morgan-Kaufmann, (2001).

[8] I. Horrocks and P. F. Patel-Schneider, 'Optimising description logic subsumption', *Jour. of Logic and Computation*, **9**(3), 267–293, (1999).

[9] V.L. Khizder, D. Toman, and G.E. Weddell, 'On decidability and complexity of description logics with uniqueness constraints', in *Proc. of the 8th Int. Conf. on Database Theory (ICDT2001)*, volume 1973 of *LNCS*, pp. 54–67. Springer, (2001).

[10] C. Lutz, 'Description logics with concrete domains—a survey', in *Advances in Modal Logics Vol. 4*, 265–296. King's College Publications, (2003).

[11] C. Lutz, *The Complexity of Reasoning with Concrete Domains*, Ph.D. dissertation, LuFG Theoretical Computer Science, RWTH Aachen, Germany, (2002).

[12] C. Lutz, 'Reasoning about entity relationship diagrams with complex attribute dependencies', in *Proc. of the Int. Workshop on Description Logics 2002 (DL2002)*, number 53 in CEUR-WS (http://ceur-ws.org/), pp. 185–194, (2002).

[13] C. Lutz, 'NExpTime-complete description logics with concrete domains', *ACM Transactions on Computational Logic*, (2003). To appear.

[14] C. Lutz, C. Areces, I. Horrocks, and U. Sattler, 'Keys, nominals, and concrete domains', LTCS-Report 02-04, Technical University Dresden, (2002). see http://lat.inf.tu-dresden.de/research/reports.html.

[15] C. Lutz, C. Areces, I. Horrocks, and U. Sattler, 'Keys, nominals, and concrete domains', in *Proc. of the Eighteenth Int. Joint Conf. on AI (IJCAI'03)*, pp. 349–354. Morgan-Kaufmann, (2003).

[16] C. Lutz and M. Milicic, 'Description logics with concrete domains and functional dependencies', LTCS-Report 04-06, TU Dresden, (2004). see http://lat.inf.tu-dresden.de/research/reports.html.

[17] E.M. Post, 'A variant of a recursively unsolvable problem', *Bulletin of the American Mathematical Society*, **52**, 264–268, (1946).

[18] D. Toman and G.E. Weddell, 'On reasoning about structural equality in XML: A description logic approach', in *Proc. of the 9th Int. Conf. on Database Theory*, number 2572 in LNCS, pp. 96–110, (2002).