Concrete Domains and Nominals United

Carlos Areces

University of Amsterdam
The Netherlands
carlos@science.uva.nl

Carsten Lutz

Technical University Dresden
Germany
lutz@tcs.inf.tu-dresden.de

Abstract

While the complexity of concept satisfiability in both $\mathcal{ALCO}$, the basic description logic $\mathcal{ALC}$ enriched with nominals, and $\mathcal{ALC}(\mathcal{D})$, the extension of $\mathcal{ALC}$ with concrete domains, is known to be PSPACE-complete, in this article we show that the combination $\mathcal{ALCO}(\mathcal{D})$ of these two logics can have a NEXPTIME-hard concept satisfiability problem (depending on the concrete domain $\mathcal{D}$ used). The proof is by a reduction of a NEXPTIME-complete variant of the domino problem to $\mathcal{ALCO}(\mathcal{D})$-concept satisfiability.

1 Motivation

The basic description logic $\mathcal{ALC}$ (or, equivalently, the modal logic $\mathbf{K}_m$) provides a simple yet useful language for representing conceptual knowledge [12,5]. However, in many cases the expressive power of $\mathcal{ALC}$ is too weak for capturing the relevant notions of an application domain. One of the hallmarks of description logics is to offer a wealth of constructors that can be added to $\mathcal{ALC}$ in order to extend its expressive power in an appropriate way.

In this paper, we consider two such constructors. Firstly, *nominals* (sometimes in disguise of the so-called "one-of" constructor [7]) permit the naming of elements in the interpretation domain [1,2]. They can hence be used to define concepts by enumeration: for example, the set of authors of this article can be defined as $\mathsf{Authors} = \mathsf{Carlos} \sqcup \mathsf{Carsten}$, where Carlos and $\mathsf{Carsten}$ are nominals. The extension of $\mathcal{ALC}$ with nominals is called $\mathcal{ALCO}$. Secondly, *concrete domains* provide a way to integrate "concrete data" into a description logic [3,10]. More precisely, a new syntactic type called "concrete features" allows to attach elements of the concrete domain (which could, e.g., be natural

numbers) to elements of the logical domain. A new concept constructor can then be used to describe constraints on concrete data. For example, if we take a concrete domain $\mathcal{D}$ that provides the natural numbers and a unary predicate $\geq_{18}$, then we can express the condition that the authors are of age by writing $\mathsf{Authors} \rightarrow \exists \mathsf{age}. \geq_{18}$, where age is a concrete feature. The extension of $\mathcal{ALC}$ with concrete domains is called $\mathcal{ALC}(\mathcal{D})$, where $\mathcal{D}$ denotes a concrete domain that can be viewed as a parameter to the logic.

Both nominals and concrete domains are known to interact with other constructors. For example, while concept satisfiability in both $\mathcal{ALCI}$ ($\mathcal{ALC}$ plus converse roles) and $\mathcal{ALCO}$ is PSpace-complete, adding a single nominal to $\mathcal{ALCI}$ yields an ExpTime-complete concept satisfiability problem [1]. Similarly, while $\mathcal{ALC}(\mathcal{D})$-concept satisfiability is PSpace-complete if reasoning with the concrete domain $\mathcal{D}$ is in PSpace [9], the corresponding problem for $\mathcal{ALCI}(\mathcal{D})$ can be NExpTime-complete even for concrete domains $\mathcal{D}$ for which reasoning is in PTime [11].

In this paper we show that there also exists a rather strong interaction between nominals and concrete domains: $\mathcal{ALCO}(\mathcal{D})$-concept satisfiability can be NExpTime-hard even if reasoning with the concrete domain $\mathcal{D}$ is in NP. To show this, we will define a special concrete domain Dom which allows the represention of so-called domino arrays and reduce a NExpTime-complete variant of the domino problem to $\mathcal{ALCO}(\mathsf{Dom})$. The reduction strategy is rather different from those that have been previously used for other extensions of $\mathcal{ALCO}$ and $\mathcal{ALC}(\mathcal{D})$ (c.f. [1,10,11]).

2 The Description Logic $\mathcal{ALCO}(\mathcal{D})$

In the following, we formally introduce the description logic $\mathcal{ALCO}(\mathcal{D})$. Let us start with defining concrete domains:

Definition 2.1 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 $P \in \Phi_\mathcal{D}$ is associated with an arity n and an n-ary predicate $P^\mathcal{D} \subseteq \Delta_\mathcal{D}^n$.

Based on concrete domains, we can now define $\mathcal{ALCO}(\mathcal{D})$-concepts.

Definition 2.2 Let $\mathsf{N_C}$, $\mathsf{N_O}$, $\mathsf{N_R}$, $\mathsf{N_{cF}}$ be pairwise disjoint and countably infinite sets of *concept names*, *nominals*, *roles*, and *concrete features*. Furthermore, let $\mathsf{N_{aF}}$ be a countably infinite subset of $\mathsf{N_R}$. The elements of $\mathsf{N_{aF}}$ are called *abstract features*. A *concrete 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 $\mathcal{ALCO}(\mathcal{D})$-concepts is the smallest set such that every concept name and every nominal is a concept, and if C and D are concepts, R is a role, g is a concrete feature, $u_1, \ldots, u_n$ are concrete paths, and $P \in \Phi_\mathcal{D}$ is a predicate of arity n, then the following expressions are also

concepts:

$$\neg C,\ C \sqcap D,\ C \sqcup D,\ \exists R.C,\ \forall R.C,\ \exists u_1, \ldots, u_n.P,\ \text{and}\ g\uparrow.$$

The description logic $\mathcal{ALCO}(\mathcal{D})$ is equipped with a Tarski-style set-theoretic semantics. Along with the semantics, we introduce the two standard inference problems: concept satisfiability and concept subsumption.

Definition 2.3 An *interpretation* $\mathcal{I}$ is a pair $(\Delta_\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta_\mathcal{I}$ is a 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 nominal N to a singleton subset $N^\mathcal{I}$ 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 concrete path, 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:

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

An interpretation $\mathcal{I}$ is a *model* of a concept C iff $C^\mathcal{I} \neq \emptyset$. A concept C is *satisfiable* iff it has a model. C is *subsumed by* a concept D (written $C \sqsubseteq D$) iff $C^\mathcal{I} \subseteq D^\mathcal{I}$ for all interpretations $\mathcal{I}$.

It is well-known that subsumption can be reduced to (un)satisfiability and vice versa: $C \sqsubseteq D$ iff $C \sqcap \neg D$ is unsatisfiable and C is satisfiable iff $C \not\sqsubseteq (A \sqcap \neg A)$, where A is an arbitrary concept name. Thus, establishing a lower complexity bound for concept satisfiability also yields such a bound for concept subsumption.

3 Domino Problems and Concrete Domains

In this section, we first define a NExpTime-complete variant of the well-known, undecidable domino problem [4,8], and then introduce a concrete domain Dom that is well-suited for reducing this domino problem to the satisfiability of $\mathcal{ALCO}(\mathsf{Dom})$-concepts.

In general, a domino problem is given by a finite set of *tile types.* All tile types are of the same size, each type having a square shape and colored edges.

An unlimited number of tiles of each type is available. In the NExpTime-hard variant of the domino problem that we use, the task is to tile a $2^{n+1} \times 2^{n+1}$-torus (i.e., a $2^{n+1} \times 2^{n+1}$-rectangle whose edges are "glued" together) without holes, overlappings, or rotation of the tiles.

Definition 3.1 A *domino system* D is a triple (T, H, V), where $T \subset \mathbb{N}$ is a finite set of *tile types* and $H, V \subseteq T \times T$ represent the horizontal and vertical matching conditions. Let D be a domino system and $a = a_0, \ldots, a_{n-1}$ be an n-tuple of tiles. A mapping $\tau : 2^{n+1} \times 2^{n+1} \to T$ is a 2^{n+1}-*solution* for D and a iff, for all $x, y < 2^{n+1}$:

- if $\tau(x, y) = t$ and $\tau(x \oplus_{2^{n+1}} 1, y) = t'$, then $(t, t') \in H$
- if $\tau(x, y) = t$ and $\tau(x, y \oplus_{2^{n+1}} 1) = t'$, then $(t, t') \in V$
- $\tau(i, 0) = a_i$ for $i < n$.

where $\oplus_i$ denotes addition modulo i.

As shown in, e.g., Corollary 4.15 of [10], it follows from results in [6] that the above variant of the domino problem is NExpTime-complete.

We now define a concrete domain Dom that will be used for reducing the NExpTime-complete domino problem to $\mathcal{ALCO}(\mathsf{Dom})$-concept satisfiability. As we shall see later, it is rather important for the reduction that the *whole* $2^{n+1} \times 2^{n+1}$-torus can be represented by a *single* element of Δ_{Dom}. To this end, Δ_{Dom} contains complex structures called "domino arrays." Moreover, we must be able to identify positions in the torus, and thus Δ_{Dom} also contains the natural numbers and $\Phi_{\mathcal{D}}$ contains predicates for accessing positions in domino arrays as well as predicates for performing simple arithmetic operations.

Definition 3.2 For every $n \in \mathbb{N}$, a function $d : \{0, \ldots, n-1\}^2 \to \mathbb{N}$ is called a *domino array* of dimension n. We use DA_n to denote the set of all domino arrays of dimension n. The concrete domain Dom is defined by setting $\Delta_{\mathsf{Dom}} := \mathbb{N} \cup \bigcup_{i>0} \mathsf{DA}_i$ and Φ_{Dom} to the (smallest) set containing the following predicates:

- a unary predicate $\top$ with $\top^{\mathsf{Dom}} = \Delta_{\mathsf{Dom}}$;
- unary predicates nat with $(\mathsf{nat})^{\mathsf{Dom}} = \mathbb{N}$ and da_k with $(\mathsf{da}_k)^{\mathsf{Dom}} = \mathsf{DA}_k$ for each $k \in \mathbb{N}$;
- for every $k \in \mathbb{N}$, a unary predicate $=_k$ with $(=_k)^{\mathsf{Dom}} = \{k\}$;
- a binary predicate $=$ with $(=)^{\mathsf{Dom}} = \{(n, n) \mid n \in \mathbb{N}\}$;
- for every $k \in \mathbb{N}$, a binary predicate incr_k with $(\mathsf{incr}_k)^{\mathsf{Dom}} = \{(n, m) \in \mathbb{N} \times \mathbb{N} \mid m = n \oplus_k 1\}$;
- a ternary predicate $+$ with $(+)^{\mathsf{Dom}} = \{(n, m, s) \in \mathbb{N}^3 \mid s = n + m\}$;
- for every $k \in \mathbb{N}$, a predicate extr_k of arity 4 with $(\mathsf{extr}_k)^{\mathsf{Dom}} = \{(d, i, j, v) \mid d \in \mathsf{DA}_k,\ i, j, v \in \mathbb{N},\ i, j < k,\ \text{and } d(i, j) = v\}$;
- for each of the above predicates p of arity n its negation, i.e., a predicate $\overline{p}$

of arity n with the extension $\Delta^n_{\mathsf{Dom}} \setminus p^{\mathsf{Dom}}$.

We assume the index $\cdot_k$ appearing in extr_k, $\overline{\mathsf{extr}_k}$, incr_k, $\overline{\mathsf{incr}_k}$, $=_k$, and $\overline{=_k}$ to be coded in binary.

The predicate extr_k will play a crucial role in the reduction. Intuitively, $\mathsf{extr_k}(d, i, j, v)$ states that, at position (i, j) of the domino array d, we find value v. The trivial predicate $\top$ and the negations of predicates are included to make the concrete domain Dom "well-behaved." More precisely, the presence of these predicates is necessary to ensure that Dom is *admissible*, a property that is usually assumed when proving decidability or complexity results for the $\mathcal{ALC}(\mathcal{D})$ family of description logics [3,9,10]. Let us define admissibility in a formal way such that this claim can be verified.

Definition 3.3 Let $\mathcal{D}$ be a concrete domain and V a set of variables. A predicate conjunction of the form

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

where P_i is an n_i-ary predicate for $i < k$ and the $x_j^{(i)}$ are variables from V, is called *satisfiable* iff 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. We say that the concrete domain $\mathcal{D}$ is called *admissible* iff its set of predicate names is closed under negation and contains a name $\top$ for $\Delta_{\mathcal{D}}$ and the satisfiability problem for finite conjunctions of predicates is decidable.

In what follows, we call finite conjunctions of predicates from Φ_{Dom} Dom-*conjunctions* and refer to the satisfiability of such conjunctions as Dom-*satisfiability*. To show that Dom is admissible, it clearly suffices to prove that Dom-satisfiability is decidable. However, Dom-satisfiability should also be of a low complexity since our aim is to demonstrate that the NExpTime-hardness of $\mathcal{ALCO}(\mathsf{Dom})$-concept satisfiability is due to the interaction between the concrete domain and nominals, and not due to the hardness of reasoning with the concrete domain Dom itself. We now show that this demand is met by proving that Dom-satisfiability is in NP.

Since the NP algorithm makes a call to an integer programming algorithm, it seems apropriate to spend a few words on integer programming. An *integer programming problem* has the form $Ax = b$, where A is an $m \times n$-matrix of rational numbers, x is an n-vector of variables, and b is an m-vector of rational numbers [13]. A *solution* of $Ax = b$ is a mapping δ that assigns a positive integer to every variable in x such that the equality $Ax = b$ holds. Deciding the *satisfiability* of an integer programming problem means to decide whether such a problem has a solution. As has been shown in, e.g., [13], this task is NP-complete. We use this fact to prove the following theorem:

Theorem 3.4 Dom-*satisfiability is in* NP.

Proof. We sketch a non-deterministic polynomial time algorithm for Dom-satisfiability. The algorithm is based on several normalization steps, simple inconsistency checks, and a final call to an algorithm which is capable of deciding the satisfiability of integer programming problems.

Before we describe the algorithm, let us introduce some notions. For a Dom-conjunction c, we use $\gamma(c)$ to denote the set of indices k appearing in da_k, $\overline{\mathsf{da}_k}$, extr_k, and $\overline{\mathsf{extr}_k}$ predicates in c. Moreover, we use $\delta(c)$ to denote $\gamma(c) \cup \{\max(\gamma(c)) + 1\}$. The *size* of a Dom-conjunction c is the number of symbols used to write it (recall that indices $\cdot_k$ in predicate names are encoded in binary).

Now let c be a Dom-conjunction. The following steps are executed sequentially to decide the satisfiability of c:

0. Return $\mathsf{unsatisfiable}$ if c contains the $\overline{\top}$ predicate.
1. Eliminate all occurrences of the $\top$ predicate from c and call the result c_1.
2. Eliminate all occurrences of the $\overline{\mathsf{nat}}$ and $\overline{\mathsf{da}_k}$ predicates in the following way: In a predicate conjunction $\hat{c}$, an occurrence $\overline{\mathsf{nat}}(x)$ of the $\overline{\mathsf{nat}}$ predicate can be eliminated by nondeterministically replacing it with $\mathsf{da}_k(x)$ for some $k \in \delta(\hat{c})$. Similarly, an occurrence $\overline{\mathsf{da}_k}(x)$ of the $\overline{\mathsf{da}_k}$ predicate can be eliminated by nondeterministically replacing it with either $\mathsf{nat}(x)$ or with $\mathsf{da}_{k'}(x)$ for some $k' \in \delta(\hat{c})$ such that $k \neq k'$. Call the result of the elimination c_2.
3. We now introduce a new predice "$<$" with the obvious semantics

$$(<)^{\mathsf{Dom}} = \{(n, m) \mid m, n \in \mathbb{N} \text{ and } n < m\}.$$

   Replace each predicate $\neq(x, y)$ in c_2 with either $<(x, y)$, $<(y, x)$, $\mathsf{da}_k(x)$, or $\mathsf{da}_k(y)$ for some $k \in \delta(c_2)$. Call the result c_3.
4. Let $\beta_1, \ldots, \beta_n$ be all conjuncts in c_3 which are of the form $\overline{\mathsf{extr}_k}(d, x, y, r)$ and let $z_1, \ldots, z_n$ be variables not appearing in c_3. For each i with $1 \leq i \leq n$ replace $\beta_i = \overline{\mathsf{extr}_k}(d, x, y, r)$ with one of the following (non deterministically):
   - the conjuncts $\mathsf{extr}_k(d, x, y, z_i)$ and $<(z_i, r)$,
   - the conjuncts $\mathsf{extr}_k(d, x, y, z_i)$ and $<(r, z_i)$,
   - one of the conjuncts $<(k, x)$ and $<(k, y)$,
   - the conjunct $\mathsf{nat}(d)$,
   - the conjunct $\mathsf{da}_{k'}(d)$ for some $k' \in \delta(c_3)$ with $k \neq k'$,
   - the conjunct $\mathsf{da}_{k'}(x)$ for some $k' \in \delta(c_3)$,
   - the conjunct $\mathsf{da}_{k'}(y)$ for some $k' \in \delta(c_3)$, or
   - the conjunct $\mathsf{da}_{k'}(r)$ for some $k' \in \delta(c_3)$.

   In a similar way, eliminate all occurrences of $\overline{\mathsf{incr}_k}$, $\overline{=_k}$, and $\overline{+}$. Call the result c_4.

   Notice that c_4 contains only the predicates nat, da_k, $=_k$, $=$, $<$, incr_k,

$+$, and extr_k.

5. Make the types of variables explicit: for each conjunct $\mathsf{extr}_k(d, x, y, r)$ in c_4, add the conjuncts $\mathsf{da}_k(d)$, $\mathsf{nat}(x)$, $\mathsf{nat}(y)$, $\mathsf{nat}(r)$. Proceed analogoulsy for $=_k$, $=$, $<$, incr_k, and $+$. Call the result c_5.
6. Return unsatisfiable if
   - c_5 contains conjuncts $\mathsf{nat}(x)$ and $\mathsf{da}_k(x)$ for some $k \in \mathbb{N}$ or
   - c_5 contains conjuncts $\mathsf{da}_k(x)$ and $\mathsf{da}_{k'}(x)$ for some $k, k' \in \mathbb{N}$ with $k \neq k'$.

   Then remove all occurrences of the nat and da_k predicates. Call the result c_6.
7. For each variable d appearing in the first position of an $\mathsf{extr_k}$ predicate in c_6 and every $i, j \in \mathbb{N}$, let $z_{d,i,j}$ be a variable not appearing in c_6. Replace each conjunct $\mathsf{extr}_k(d, x, y, r)$ with the new conjuncts $=_i(x)$, $=_j(y)$, $=(r, z_{d,i,j})$ for some $i, j < k$. Call the result c_7.

   Notice that c_7 only contains the predicates $=_k$, $=$, $<$, incr_k, and $+$.
8. For each $n \in \mathbb{N}$, let z_n be a variable not occurring in c_7. Replace each predicate $\mathsf{incr}_k(x, y)$ in c_7 with either $=_{k-1}(x) \wedge =_0(y)$ or $=_{k-1}(z_{k-1}) \wedge <(x, z_{k-1}) \wedge =_1(z_1) \wedge +(x, z_1, y)$.

   Call the result c_8, notice that it contains only the predicates $=_k$, $=$, $<$, and $+$.
9. It is easy to see that c_8 can be transformed into an integer programming problem (possibly using slack variables to handle "$<$"). Use a standard NP algorithm (see, e.g., [13]) to decide the satisfiability of this problem and return the result.

It is straightforward to prove the correctness of the sketched algorithm by showing that (i) each of the normalization steps preserves (un)satisfiability, (ii) each of the inconsistency checks is correct, and (iii) the reduction to integer programming is correct. Moreover, it is not hard to see that the algorithm can be executed in nondeterministic polynomial time: each of the normalization steps leads to at most a polynomial blowup of the size of the predicate conjunction. Finally, deciding the satisfiability of integer programming problems can be done in NP [13]. □

4 The Reduction

We prove NExpTime-hardness of $\mathcal{ALCO}(\mathsf{Dom})$-concept satisfiability by reduction of the NExpTime-complete domino problem introduced in the previous section. For a domino system D with initial condition a, the reduction concept $C_{\mathsf{D},a}$ is defined as

$$\begin{aligned} C_{\mathsf{D},a} := {} & \mathsf{TreeX} \sqcap \forall R^{n+1}.\mathsf{TreeY} \sqcap \forall R^{2(n+1)}.\exists f.(N \sqcap \exists \mathsf{darr}.\mathsf{da}_{2^{n+1}}) \\ & \sqcap \forall R^{2(n+1)}.(\mathsf{CompXPos} \sqcap \mathsf{CompYPos} \sqcap \mathsf{Label} \sqcap \mathsf{CheckLabel}) \\ & \sqcap \forall R^{2(n+1)}.(\mathsf{Init} \sqcap \mathsf{CheckHMatch} \sqcap \mathsf{CheckVMatch}) \end{aligned}$$

where $\forall R^n.C$ is an abbreviation for the n-fold nesting $\forall R. \cdots \forall R.C$ and TreeX, TreeY, $\mathsf{CompXPos}$, etc. are abbreviations for complex concepts that will soon be described in detail. It is interesting to note that $C_{\mathsf{D},a}$ refers to *only one* nominal N, which is indeed the only nominal used in the entire reduction.

Intuitively, the main purpose of the first line of $C_{\mathsf{D},a}$ is to enforce a tree structure of depth $2(n+1)$ whose leaves correspond to positions in the $2^{n+1} \times 2^{n+1}$-torus. More precisely, TreeX and TreeY are defined as follows:

$$\begin{aligned} \mathsf{TreeX} &:= \exists R.X_0 \sqcap \exists R.\neg X_0 \sqcap \bigsqcap_{i=1..n} \forall R^i.(\mathsf{DistX}_{i-1} \sqcap \exists R.X_i \sqcap \exists R.\neg X_i) \\ \mathsf{TreeY} &:= \exists R.Y_0 \sqcap \mathsf{DistX}_n \sqcap \exists R.\neg Y_0 \sqcap \bigsqcap_{i=1..n} \forall R^i.(\mathsf{DistY}_{i-1} \sqcap \mathsf{DistX}_n \sqcap \exists R.Y_i \sqcap \exists R.\neg Y_i) \\ \mathsf{DistX}_k &:= \bigsqcap_{i=0..k} ((X_i \rightarrow \forall R.X_i) \sqcap (\neg X_i \rightarrow \forall R.\neg X_i)) \\ \mathsf{DistY}_k &:= \bigsqcap_{i=0..k} ((Y_i \rightarrow \forall R.Y_i) \sqcap (\neg Y_i \rightarrow \forall R.\neg Y_i)) \end{aligned}$$

The TreeX concept enforces that, in every model of $C_{\mathsf{D},a}$, there exists a binary tree of depth $n+1$. Moreover, the DistX concepts (there exists one for each $k \in \{0, \ldots, n\}$) ensure that the leaves of this tree are binarily numbered (from 0 to $2^{n+1}-1$) by the concept names $X_0, \ldots, X_n$. More precisely, for a domain object $d \in \Delta^{\mathcal{I}}$, set

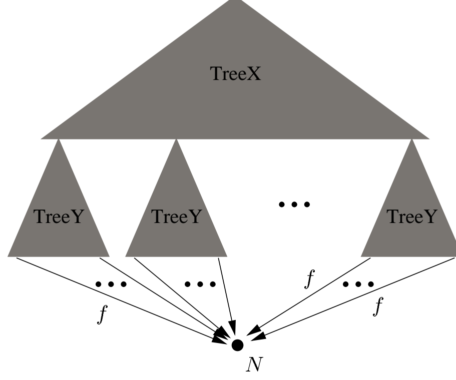
$$\mathsf{xpsn}(d) = \Sigma_{i=0}^{n} \alpha_i(d) * 2^i \quad \text{where} \quad \alpha_i(d) = \begin{cases} 1 \text{ if } d \in X_i^{\mathcal{I}} \\ 0 \text{ otherwise.} \end{cases}$$

The TreeX and DistX concepts ensure that there exist leaves of the tree $d_0, \ldots, d_{2^{n+1}-1}$ such that $\mathsf{xpsn}(d_i) = i$. Intuitively, this numbering represents the horizontal positions in the $2^{n+1} \times 2^{n+1}$-torus. The vertical positions are coded in a similar way by the $Y_0, \ldots, Y_n$ concept names. More specifically, the TreeY, DistX, and DistY ensure that every d_i ($0 \leq i \leq 2^{n+1}-1$) is the root of another tree, in which (i) every node has the same "$X_0, \ldots, X_n$-configuration" as the root node, and (ii) the leaves are numbered binarily using the concept names $Y_0, \ldots, Y_n$ (note that the TreeY concept appears in $C_{\mathsf{D},a}$ inside a $\forall R^{n+1}$ value restriction). Define

$$\mathsf{ypsn}(d) = \Sigma_{i=0}^{n} \beta_i(d) * 2^i \quad \text{where} \quad \beta_i(d) = \begin{cases} 1 \text{ if } d \in Y_i^{\mathcal{I}} \\ 0 \text{ otherwise.} \end{cases}$$

Taking together the leafs of all the trees enforced by the TreeY concept, we obtain a node $e_{i,j}$ for each $0 \leq i,j < 2^{n+1}$ such that $\mathsf{xpsn}(e_{i,j}) = i$ and $\mathsf{ypsn}(e_{i,j}) = j$, i.e., each $e_{i,j}$ represents a position in the $2^{n+1} \times 2^{n+1}$-torus.

Due to last conjunct in the first line of $C_{\mathsf{D},a}$, all the $e_{i,j}$ are connected via the concrete feature f to the domain element identified by the nominal N. Thus, very roughly, the models of $C_{\mathsf{D},a}$ look as follows:



To make the grid positions accessible by the concrete domain, we translate the positions encoded by $X_0, \dots, X_n$ and $Y_0, \dots, Y_n$ into integer values and "store" them in the concrete features xpos and ypos such that $\mathsf{xpos}^{\mathcal{I}}(e_{i,j}) = i$ and $\mathsf{ypos}^{\mathcal{I}}(e_{i,j}) = j$. This is done by the concepts $\mathsf{CompXPos}$ and $\mathsf{CompYPos}$ using the auxiliary concrete features $x_0, \dots, x_n$, $s_0, \dots, s_n$, $y_0, \dots, y_n$, and $s'_0, \dots, s'_n$.

$$
\begin{aligned}
\mathsf{CompXPos} := & \bigsqcap_{i=0..n} \big((X_i \rightarrow \exists x_i.\, =_{2^i}) \sqcap (\neg X_i \rightarrow \exists x_i.\, =_0)\big) \\
& \sqcap \exists s_0, x_0.= \sqcap \big(\bigsqcap_{i=1..n} \exists s_{i-1}, x_i, s_i.+\big) \sqcap \exists s_n, \mathsf{xpos}.= \\
\mathsf{CompYPos} := & \bigsqcap_{i=0..n} \big((Y_i \rightarrow \exists y_i.\, =_{2^i}) \sqcap (\neg Y_i \rightarrow \exists y_i.\, =_0)\big) \\
& \sqcap \exists s'_0, y_0.= \sqcap \big(\bigsqcap_{i=1..n} \exists s'_{i-1}, y_i, s'_i.+\big) \sqcap \exists s'_n, \mathsf{ypos}.=
\end{aligned}
$$

A domino type is assigned to each domain element $e_{i,j}$ (i.e., to each position in the grid), and stored in the concrete feature label. This is done by the Label concept.

$$
\begin{aligned}
\mathsf{Label} := & \bigsqcup_{d \in \mathcal{D}} \exists \mathsf{label}.=_d \\
\mathsf{CheckLabel} := & \exists (f \circ \mathsf{darr}, \mathsf{xpos}, \mathsf{ypos}, \mathsf{label}).\mathsf{extr}_{2^{n+1}}
\end{aligned}
$$

It remains to ensure that the tiling satisfies the initial condition a as well as the horizontal and vertical matching conditions. To do this, we "copy" the grid as represented by the domain elements $e_{i,j}$ into a single domino array from the concrete domain Dom. This is where the nominal N comes into play: as has already been mentioned, there exists a domain element z identified by the nominal N such that $f^{\mathcal{I}}(e_{i,j}) = z$ for every $e_{i,j}$. Due to the last conjunct of

$C_{\mathsf{D},a}$, a domino array of size 2^{n+1} is associated with z via the concrete feature darr. The $\mathsf{CheckLabel}$ concept, of which the domain elements $e_{i,j}$ are required to be instances, ensures that this domino array stores a copy of the grid as represented by the domain elements $e_{i,j}$. Note that the use of the nominal N is crucial at this point: since N identifies the unique domain element z, the path $f \circ \mathsf{darr}$ reaches one and the same domino array starting from *any* of the $e_{i,j}$.

The copy of the grid stored in $\mathsf{darr}^{\mathcal{I}}(z)$ can now be used to check that the tiling is correct, which is done by the following three concepts:

$$
\begin{aligned}
\mathsf{Init} &:= \bigsqcap_{i=0..n-1} \big((\exists \mathsf{ypos}.{=_0} \sqcap \exists \mathsf{xpos}.{=_i}) \rightarrow \exists \mathsf{label}.{=_{a_i}}\big) \\
\mathsf{CheckHMatch} &:= \exists \mathsf{xpos}, \mathsf{xsucc}.\mathsf{incr}_{2^{n+1}} \sqcap \exists (f \circ \mathsf{darr}, \mathsf{xsucc}, \mathsf{ypos}, \mathsf{xslabel}).\mathsf{extr}_{2^{n+1}} \\
&\quad \sqcap \bigsqcup_{(d,d') \in H} (\exists \mathsf{label}.{=_d} \sqcap \exists \mathsf{xslabel}.{=_{d'}}) \\
\mathsf{CheckVMatch} &:= \exists \mathsf{ypos}, \mathsf{ysucc}.\mathsf{incr}_{2^{n+1}} \sqcap \exists (f \circ \mathsf{darr}, \mathsf{xpos}, \mathsf{ysucc}, \mathsf{yslabel}).\mathsf{extr}_{2^{n+1}} \\
&\quad \sqcap \bigsqcup_{(d,d') \in V} (\exists \mathsf{label}.{=_d} \sqcap \exists \mathsf{yslabel}.{=_{d'}})
\end{aligned}
$$

The Init concept ensures that the initial condition $a = a_0, \ldots, a_{n-1}$ is respected. The $\mathsf{CheckHMatch}$ and $\mathsf{CheckVMatch}$ concepts, of which the domain elements $e_{i,j}$ are instances, enforce the matching conditions. Let us focus on $\mathsf{CheckHMatch}$ since $\mathsf{CheckVMatch}$ works analogously. Fix a domain element $e_{i,j}$. The first conjunct of $\mathsf{CheckHMatch}$ computes the horizontal position of the horizontal neighbor of $e_{i,j}$ (this position is obviously $i+1 \mod 2^{n+1}$) and uses the concrete feature xsucc to store it. The second conjunct extracts the label of this horizontal neighbor from the copy of the grid stored in the darr-successor of the domain element z. This neighbor's label is stored using the concrete feature $\mathsf{xslabel}$. All that remains to be done is to ensure that the label-successor and the $\mathsf{xslabel}$-successor of $e_{i,j}$ satisfy the horizontal matching condition, which is done by the third conjunct of $\mathsf{CheckHMatch}$.

Using the above considerations, the correctness of the reduction is readily checked. Moreover, the size of $C_{\mathsf{D},a}$ is at most polynomial in n. To see this, recall that indizes in Dom-predicates are coded in binary. Summing up, the described reduction yields the following result:

Theorem 4.1 *Satisfiability of $\mathcal{ALCO}(\mathsf{Dom})$-concept satisfiability is* NExpTime-*hard.*

Since, as was noted in Section 2, unsatisfiability can be reduced to subsumption, $\mathcal{ALCO}(\mathsf{Dom})$-concept subsumption is co-NExpTime-hard. Note that a *single* nominal is sufficient for these hardness results.

5 Conclusion

In this paper, we have shown that combining concrete domains and nominals may have rather dramatic effects on the complexity of reasoning: although $\mathcal{ALCO}$-concept satisfiability is in PSpace [1] and $\mathcal{ALC}(\mathcal{D})$-concept satisfiability is PSpace-complete if $\mathcal{D}$-satisfiability is in PSpace [9], $\mathcal{ALCO}(\mathcal{D})$-concept satisfiability may be NExpTime-hard, depending on the concrete domain $\mathcal{D}$ used. To show this, we have defined a concrete domain Dom that provides for so-called domino arrays—a data structure that comes very handy for representing domino problems. Such a concrete domain is not too natural for knowledge representation and other application areas of description logics. Nevertheless, our result indicates that one has to carefully investigate the impact on the complexity of reasoning when combining nominals and concrete domains. Moreover, it implies that the general PSpace-upper bound for $\mathcal{ALC}(\mathcal{D})$ mentioned above cannot be extended to $\mathcal{ALCO}(\mathcal{D})$.

The reader may wonder why we use the somewhat artifical domino arrays as part of the concrete domain, instead of using a *single* integer to represent the whole torus. The reason is that, to access the individual positions of the torus, we would then need a ternary multiplication predicate. Moreover, it is not hard to see that a concrete domain $\mathcal{D}$ which provides equality to one, binary equality, ternary addition and multiplication is powerful enough to capture Hilbert's tenth problem. Thus, the satisfiability of finite $\mathcal{D}$-conjunctions is undecidable which clearly implies that all description logics incorporating this concrete domain are also undecidable which prohibits a fine-grained complexity analysis.

References

[1] C. Areces, P. Blackburn, and M. Marx. A road-map on complexity for hybrid logics. In J. Flum and M. Rodríguez-Artalejo, editors, *Computer Science Logic*, number 1683 in Lecture Notes in Computer Science, pages 307–321. Springer-Verlag, 1999.

[2] C. Areces and M. de Rijke. From description logics to hybrid logics, and back. In F. Wolter, H. Wansing, M. de Rijke, and M. Zakharyaschev, editors, *Advances in Modal Logics Volume 3*. CSLI Publications, Stanford, CA, USA, 2001.

[3] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 452–457, Sydney, Australia, 1991.

[4] R. Berger. The undecidability of the dominoe problem. *Memoirs of the American Mathematical Society*, 66, 1966.

[5] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic.* Cambridge University Press, 2001.

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

[7] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick, and A. Borgida. Living with classic: When and how to use a KL-ONE-like language. In J. F. Sowa, editor, *Principles of Semantic Networks – Explorations in the Representation of Knowledge*, chapter 14, pages 401–456. Morgan Kaufmann, 1991.

[8] D. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

[9] C. Lutz. Reasoning with concrete domains. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 90–95. Morgan Kaufmann, 1999.

[10] C. Lutz. *The Complexity of Reasoning with Concrete Domains.* PhD thesis, Teaching and Research Area for Theoretical Computer Science, RWTH Aachen, 2001.

[11] C. Lutz. NExpTime-complete description logics with concrete domains. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR'01)*, number 2083 in Lecture Notes in Artifical Intelligence, pages 45–60. Springer-Verlag, 2001.

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

[13] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, Chichester, UK, 1986.