Between tree patterns and conjunctive queries: is there tractability beyond acyclicity?

Filip Murlak, Michał Ogiński, and Marcin Przybyłko

Institute of Informatics, University of Warsaw
fmurlak@mimuw.edu.pl, {M.Oginski,M.Przybylko}@students.mimuw.edu.pl

Abstract. In static analysis of queries over trees in the presence of schemas, there is an exponential complexity gap between conjunctive queries (CQs, positive existential first-order formulae without disjunction) and tree patterns (tree-like acyclic CQs). Motivated by applications in XML data management, we consider various restrictions of CQs that bring their complexity down to that of tree patterns. Most importantly, we show that vertical tree patterns can be costlessly extended with full horizontal CQs over children. We also consider restricted classes of schemas and show that under disjunction-free schemas the complexity of static analysis sometimes drops dramatically.

1 Introduction

Static analysis is a common name used in database theory for problems that do not deal with data, but only with queries. Such problems are often part of complex data management tasks, like data integration and data exchange [15, 19]. Most important static analysis problems include satisfiabiliy (Given a query q, is there a database D such that the returned set of tuples $q(D)$ is nonempty?) and query containment (Given q_1, q_2, does $q_1(D) \subseteq q_2(D)$ hold for each D?). As for first order logic these problems are undecidable, restricted query languages are considered. For relational databases, conjunctive queries (CQs, positive existential formulae without disjunction) and their unions (UCQs) are used most widely. The reason is a relatively low cost of static analysis [12], and expressive power meeting most typical needs (select-from-where SQL queries). In contrast, in XML static analysis, where problems are relativized to XML trees accepted by a given schema (often modelled as a tree automaton), the complexity of full CQs, using child and descendant relations, and sibling order, is prohibitively high [8, 9, 16]. As a remedy, more restrictive languages of acyclic CQs and tree patterns (tree-like acyclic CQs) were introduced. For instance, literature on XML data exchange and metadata management considers almost exclusively tree patterns [1–3, 14]. A fine complexity analysis for CQs and UCQs over XML trees would be useful in designing richer formalisms, based on intermediate classes of queries.

Most research on static analysis for queries over XML trees was done for fragments of XPath 1.0, which is a language allowing only acyclic queries [5, 18, 20, 21, 24, 25], or XPath 2.0, which allows path intersection, but not arbitrary joins

[11, 17]. As has been observed by Gottlob, Koch, and Shulz, each CQ on trees can be translated to a union of exponentially many polynomial tree patterns [16]. This gives an upper bound on the complexity of containment exponentially higher then that for tree patterns: 2ExpTime in general and ExpSpace under non-recursive schemas (where the depth of trees is bounded by the size of the schema), while for tree patterns they are ExpTime and PSpace, respectively [5, 20, 24]. Björklund, Martens, and Schwentick show that the exponential gap cannot be avoided in general, as even containment of CQs using only child and descendant relation is 2ExpTime-complete, and ask if there are more manageable classes of CQs, other then acyclic CQs [9]. We are most interested in results of the form: under certain restrictions, the complexity of containment of UCQs is the same as that of unions of tree patterns. For example, if only child relation is available, the general case reduces to the acyclic case, as each CQ can be rewritten as a single tree pattern of linear size (cf. [4]).

We focus on the restrictions most commonly studied in XML data exchange and metadata management: non-recursive or disjunction-free schemas (cf. nested-relational DTDs [2, 3, 6]), and limited use of horizontal or vertical relations. We first prove that, over words, containment of UCQs is PSpace-complete, just like for unions of tree patterns (Sect. 4). Then we apply these results to trees (Sect. 5) and show that the complexities match for a fairly general class of "forest-like" UCQs, combining vertical tree patterns with arbitrary horizontal CQs over children. This is further exploited to prove the same for

- UCQs that do not use the descendant relation;
- UCQs specifying labels of all mentioned nodes, under non-recursive DTDs.

Finally in Sect. 6 we show that under disjunction-free schemas the containment of UCQs without the next-sibling relation is in coNExpTime and PSpace-hard, and if additionally schemas are non-recursive, it is on the second level of the polynomial hierarchy; with next-sibling, the complexity does not drop.

We work exclusively with Boolean queries; as explained in [9], this is not a restriction. Due to space limitations some arguments are omitted. For more details see the appendix available at `www.mimuw.edu.pl/~fmurlak/papers/patsat.pdf`.

2 Preliminaries

XML documents and trees. We model XML documents as unranked labelled trees. Formally, a *tree* over a finite labelling alphabet Γ is a relational structure $\mathcal{T} = \langle T, \downarrow, \downarrow^+, \rightarrow, \overset{+}{\rightarrow}, (a^{\mathcal{T}})_{a\in\Gamma}\rangle$, where

- the set T is an unranked tree domain, i.e., a prefix-closed subset of $\mathbb{N}^*$ such that $n \cdot i \in T$ implies $n \cdot j \in T$ for all $j < i$;
- the binary relations $\downarrow$ and $\rightarrow$ are the child relation ($n \downarrow n \cdot i$) and the next-sibling relation ($n \cdot i \rightarrow n \cdot (i+1)$);
- $\downarrow^+$ and $\overset{+}{\rightarrow}$ are transitive closures of $\downarrow$ and $\rightarrow$;
- $(a^{\mathcal{T}})_{a\in\Gamma}$ is a partition of the domain T into possibly empty sets.

We write $|\mathcal{T}|$ to denote the number of nodes of tree $\mathcal{T}$. The partition $(a^{\mathcal{T}})_{a\in\Gamma}$ defines a labelling of the nodes of $\mathcal{T}$ with elements of Γ, denoted by $\ell_{\mathcal{T}}$.

Automata and DTDs. The principal schema language we use are tree automata, abstracting Relax NG [22, 23]. We are using a variant in which the state in a node v depends on the states in the previous sibling and the last child of v. Such automata are equivalent to standard automata on unranked trees, as explained in [23]. Formally, an automaton is a tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$, where Σ is the labelling alphabet (the set of element types in our case), Q is the state space with the initial state q_0 and final states F, and $\delta \subseteq Q \times Q \times \Sigma \times Q$ is the transition relation. A *run* of $\mathcal{A}$ over a tree $\mathcal{T}$ is a labelling ρ of the nodes of $\mathcal{T}$ with the states of $\mathcal{A}$ such that for each node v with children $v\cdot 0, v\cdot 1, \ldots, v\cdot k$ and previous sibling w, $\big(\rho(w), \rho(v\cdot k), \ell_{\mathcal{T}}(v), \rho(v)\big) \in \delta$. If v has no previous sibling, $\rho(w)$ in the condition above is replaced with q_0. Similarly, if v has no children, $\rho(v\cdot k)$ is replaced with q_0. The language of trees *recognized* by $\mathcal{A}$, denoted by $L(\mathcal{A})$, consists of all trees admitting an *accepting* run of $\mathcal{A}$, i.e. a run that assigns one of the final states to the root.

A simpler schema language is provided by DTDs. A *document type definition* (DTD) over a labelling alphabet Γ is a pair $D = \langle r, P_D\rangle$, where $r \in \Gamma$ is a distinguished root symbol and P_D is a function assigning regular expressions over $\Gamma - \{r\}$ to the elements of Γ, usually written as $\sigma \to e$, if $P_D(\sigma) = e$. A tree $\mathcal{T}$ *conforms to* a DTD D, denoted $\mathcal{T} \models D$, if its root is labelled with r and for each node s in $\mathcal{T}$ the sequence of labels of its children is in the language of $P_D(\ell_{\mathcal{T}}(s))$. The set of trees conforming to D is denoted by $L(D)$. It is well known (and easy to see) that there is a PTIME translation from DTDs to automata.

We shall often consider *non-recursive* schemas, DTDs or automata. A DTD D is non-recursive if in every tree conforming to D each path contains each label at most once. A schema given by a tree automaton is non-recursive if in each run (accepting or not) each path contains each state at most once. The height of trees conforming to non-recursive schemas is bounded by the size of the schema.

CQs and patterns. A conjunctive query (CQ) over alphabet Γ is a formula of first order logic using only conjunction and existential quantification, over unary predicates $a(x)$ for $a \in \Gamma$ and binary predicates $\downarrow, \downarrow^+, \rightarrow, \overset{+}{\rightarrow}$ (referred to as *child*, *descendant*, *next sibling*, and *following sibling*, respectively). Since we work only with Boolean queries, to avoid unnecessary clutter we often skip the quantifiers, assuming that all variables are by default quantified existentially.

An alternative way of looking at CQs is via patterns. A *pattern* π over Γ can be presented as $\pi = \langle V, E_c, E_d, E_n, E_f, \ell_\pi\rangle$ where ℓ_π is a partial function from V to Γ, and $\langle V, E_c \cup E_d \cup E_n \cup E_f\rangle$ is a finite graph whose edges are split into child edges E_c, descendant edges E_d, next-sibling edges E_n, and following-sibling edges E_f. By $|\pi|$ we mean the size of the underlying graph.

We say that a tree $\mathcal{T} = \langle T, \downarrow, \downarrow^+, \rightarrow, \overset{+}{\rightarrow}, (a^{\mathcal{T}})_{a\in\Gamma}\rangle$ *satisfies* a pattern $\pi = \langle V, E_c, E_d, E_n, E_f, \ell_\pi\rangle$, denoted $\mathcal{T} \models \pi$, if there exists a homomorphism $h\colon \pi \to \mathcal{T}$, i.e., a function $h : V \to T$ such that

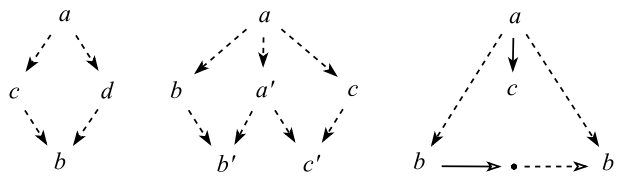

Fig. 1. Typical patterns. Void and solid heads indicate horizontal and vertical order, respectively. Dashed lines indicate transitive closure. The leftmost pattern can be expressed using path intersection operator from XPath 2.0, but the middle one cannot.

- $h : \langle V, E_c, E_d, E_n, E_f\rangle \to \langle T, \downarrow, \downarrow^+, \to, \overset{+}{\to}\rangle$ is a homomorphism of relational structures; and
- $\ell_{\mathcal{T}}(h(v)) = \ell_\pi(v)$ for all v in the domain of ℓ_π.

Each pattern can be seen as a CQ, and vice versa. In what follows we use the terms "pattern" and "CQ" interchangeably. *Tree patterns* are patterns whose underlying graph is a directed tree with edges pointing from parents to children.

Containment and satisfiability. We focus on the following satisfiability problem.

PROBLEM: BC-SAT
INPUT: Boolean combination of patterns φ, schema $\mathcal{S}$ (automaton or DTD).
QUESTION: Is there a tree $\mathcal{T} \in L(\mathcal{S})$ such that $\mathcal{T} \models \varphi$?

For $\sigma \subseteq \{\downarrow, \downarrow^+, \to, \overset{+}{\to}, \_\}$ we write BC-SAT(σ) to denote BC-SAT restricted to patterns which use only the axes listed in σ. If the wildcard symbol, $\_$, is not present in σ, the labelling functions in patterns are required to be total, or in other words that each variable must occur in an atom of the form $a(x)$ for some $a \in \Gamma$. We use the following abbreviations: $\Downarrow$ for $\downarrow, \downarrow^+, \_$; and $\Rightarrow$ for $\to, \overset{+}{\to}, \_$.

Containment for UCQs is inter-reducible with non-satisfiability for Boolean combinations of the form $\pi \wedge \neg\pi_1 \wedge \neg\pi_2 \wedge \cdots \wedge \neg\pi_k$. Validity of a query is equivalent to non-satisfiability of its negation. Most of our lower bounds only use conjunctions of negations of patterns, thus giving dual lower bounds for validity (and containment).

3 Basic complexity bounds

In this section we briefly summarize known complexity bounds and establish some new bounds in the case of non-recursive schemas. The complexity of BC-SAT for tree patterns follows immediately from the results on XPath satisfiability and containment.

Theorem 1 ([5, 11, 24, 20]). *For tree patterns,* BC-SAT$(\Downarrow, \Rightarrow)$ *is* EXPTIME*-complete and* PSPACE*-complete under non-recursive schemas. The lower bounds hold even for containment of unions of tree patterns using only* $\downarrow$*.*

The ExpTime upper bound follows from the translation of downward XPath to automata [24], later extended to cover horizontal axes in [11] (also implicit in [20]). The PSpace upper bound follows from [5]. The lower bounds in [5, 24] rely on the availability of wildcard (or disjunction inside XPath expressions), but they can be strengthened to queries using only $\downarrow$.

As we have mentioned, for CQs the bounds are exponentially worse.

Theorem 2 ([9]). *For arbitrary patterns,* $\mathsf{BC\text{-}SAT}(\Downarrow, \Rightarrow)$ *is* 2ExpTime*-complete. The lower bound holds already for validity of a single CQ using only* $\Downarrow$*.*

To complete the background for the main results of this paper, described in Sections 4–6, we now settle the complexity of BC-SAT under non-recursive schemas. The complexity drops only slightly, but the cost of $\rightarrow$ begins to show.

Theorem 3. *Under non-recursive schemas* $\mathsf{BC\text{-}SAT}(\Downarrow)$ *and* $\mathsf{BC\text{-}SAT}(\Downarrow, \overset{+}{\rightarrow})$ *is* NExpTime*-complete and* $\mathsf{BC\text{-}SAT}(\Downarrow, \Rightarrow)$ *is* ExpSpace*-complete. The lower bounds hold already for conjunctions of negated patterns.*

The ExpSpace upper bound follows immediately by translation to tree patterns. The NExpTime upper bound is obtained via a linearly-branching model property, which relies on the fact that an unsatisfied pattern without $\rightarrow$ never becomes satisfied when a subtree is deleted. The lower bounds use an ingenious pattern construction from [9].

4 CQs over words

In the classification sketched out in the previous section, horizontal CQs were to some extent drowned in the overall complexity of patterns. We shall have a closer look at them now: we restrict our models to words. We show that under this restriction the complexity of CQs matches that of tree patterns (Theorem 4); in the next section we show how this can be applied to the tree case.

Our main building block is a procedure $\textsc{Match}_\pi$ associated with each pattern π. The procedure takes as input a word w and checks if $w \models \pi$. It reads w letter by letter, possibly storing some information in the working memory, polynomial in $|\pi|$ and independent of w (it can be seen as a DFA, exponential in π). The procedure looks for the *earliest (leftmost) matching* of π in w, as defined below. We note that earliest matchings were previously used in [7] for tree patterns.

We use $\leq$ and $+1$ for the standard order and successor on the positions of words (an initial segment of natural numbers), and define homomorphisms just like for trees. Whenever we write $h : \pi \to w$, we implicitly assume that h is a homomorphism.

Definition 1. *Let* $g, h \colon \pi \to w$ *be two homomorphisms.*

- *We write* $g \leq h$ *if* $g(v) \leq h(v)$ *for each vertex* v *of* π*.*
- *We define* $\min(g, h) \colon \pi \to w$ *as* $\min(g, h)(v) = \min(g(v), h(v))$*.*

Lemma 1. *Let* w *be a word satisfying a* $\Rightarrow$*-pattern* π*.*

1. *For all* $g, h\colon \pi \to w$, $\min(g,h)$ *is a homomorphism.*
2. *There exists* $h_{\min}\colon \pi \to w$ *such that* $h_{\min} \le h$ *for all* $h\colon \pi \to w$.
3. *For each set* X *of vertices of* π *and each* $h\colon \pi \to w$ *there is a* $\hat{h}\colon \pi \to w$ *extending* $h|_X$ *such that* $\hat{h} \le h'$ *for each* $h'\colon \pi \to w$ *extending* $h|_X$.

We call the unique $h_{\min}$ from Lemma 1 the *earliest matching* of π in w.

$\textsc{Match}_\pi(w)$ works with components of π, called *firm subpatterns*, described in Definition 3.

Definition 2. *A* $\to$-component *of* π *is a maximal connected subgraph of* $\to$*-graph of* π. *In the* graph of $\to$-components of π, *denoted* G_π, *there is an edge from a* $\to$*-component* π_1 *to a* $\to$*-component* π_2 *if there is a* $\overset{+}{\to}$ *edge in* π *from a vertex of* π_1 *to a vertex of* π_2.

Definition 3. *A pattern* π *is* firm *if* G_π *is strongly connected. In general, each strongly connected component* X *of* G_π *defines a* firm subpattern *of* π*: the subgraph of* π *induced by the vertices of* $\to$*-components contained in* X. *The* DAG of firm subpatterns *of* π, *denoted* F_π, *is the standard DAG of strongly connected components of* G_π.

For example, the pattern in Fig. 2 on page 8 is firm, but has three $\to$-components.

The matching procedure $\textsc{Match}_\pi(w)$ works as follows:

- it reads the input word w from left to right trying to match firm subpatterns of π in the topological order given by F_π;
- for each firm subpattern it finds the earliest matching that does not violate the $\overset{+}{\to}$ edges connecting it with previously matched firm subpatterns.

Since we are proceeding in the topological order, each firm subpattern is processed after all its predecessors have been matched. Hence, the algorithm always finds a correct homomorphism or none at all. Completeness of the algorithm follows from the lemma below by straightforward induction (where Y is the union of previously matched firm patterns and X is obtained by adding a new one).

Lemma 2. *Let* $h\colon \pi \to w$ *be the earliest matching and let* X *be a set of vertices of* π *such that no edge enters* X *from the outside, and the only edges leaving* X *are* $\overset{+}{\to}$. *For each* $Y \subseteq X$, *if* $g\colon \pi|_X \to w$ *is the least homomorphism extending* $h|_Y$ *to* X, *then* $h|_X = g$.

Now we need to bound the memory used by $\textsc{Match}_\pi(w)$. We claim that the algorithm only needs to remember last $|\pi|$ symbols read (plus the matching constructed so far, restricted to this suffix). It is straightforward to check that each homomorphic image of a firm pattern π_0 is a subword of length at most $|\pi_0|$. Based on this observation, we prove the claim. For $i \le |w|$, let Π_i be the set of firm subpatterns of π matched by $\textsc{Match}_\pi(w)$ after processing the first i symbols of w. Note that if a position j is not touched by the matching, all firm subpatterns matched before this position are in Π_j. By pigeon-hole principle, there is a position j between $i - |\pi|$ and i that is not touched by the matching.

By the previous comment, all patterns from $\Pi_i \setminus \Pi_{i-1}$ are matched between j and i. It follows that $\textsc{Match}_\pi(w)$ only needs to remember the last $|\pi|$ symbols.

Using the matching procedure we prove the main result of this section.

Theorem 4. *On words* $\mathsf{BC\text{-}SAT}(\Rightarrow)$ *is* PSPACE*-complete, with hardness already for conjunctions of negated tree patterns using only* $\to, \_$ *or CQs using only* $\to, \overset{+}{\to}$.

Proof. To check if a Boolean combination φ is satisfiable in a word accepted by an automaton $\mathcal{A}$, we non-deterministically generate letters of a word $w \in L(\mathcal{A})$ and feed with them $\textsc{Match}_\pi$ for each π used in φ. We accept if the split into matched and unmatched patterns satisfies φ. To prevent looping, we count the number of letters and stop when we reach certain threshold, single exponential in $|\varphi|$. To establish the threshold, recall that $\textsc{Match}_\pi$ can be seen as a DFA, exponential in $|\pi|$. The product automaton corresponding to all running copies of the matching procedure is single exponential in φ. The threshold can be set to the size of the product automaton. By Savitch theorem we can eliminate non-determinism from this algorithm.

For the lower bound we give a reduction from the following tiling problem, which is known to be PSPACE-complete: Given a set of tiles $T = \{t_1, t_2, \ldots, t_k\}$, relations $H, V \subseteq T \times T$, and a number n in unary, decide if there is a number m and an $m \times n$ matrix $(a_{i,j})$ with entries from T such that $a_{1,1} = t_1$, $a_{m,n} = t_k$, $(a_{i,j}, a_{i,j+1}) \in H$ for $1 \leq i \leq m$, $1 \leq i < n$ and $(a_{i,j}, a_{i+1,j}) \in V$ for $1 \leq i < m$, $1 \leq i \leq n$. In fact we give the reduction from the following linearised tiling to which the original problem can be easily reduced: Given T, H, V, n, decide if there is a sequence of tiles $s_1 s_2 \ldots s_\ell$ such that $s_1 = t_1$, $s_\ell = t_k$, $(s_i, s_{i+1}) \in H$ for all $i \leq \ell - 1$, and $(s_i, s_{i+n}) \in V$ for all $i \leq \ell - n$.

Let an instance of the linearised tiling problem be T, H, V, n. If wildcard is available, we can assume our alphabet is $T \cup \{r\}$ and take the DTD $r \to t_0 T^* t_k$ and the following combination of patterns:

$$\bigwedge_{(t_i,t_j)\notin H} \neg\exists x\, \exists y\, (x \to y) \wedge t_i(x) \wedge t_j(y) \wedge \bigwedge_{(t_i,t_j)\notin V} \neg\exists x\, \exists y\, (x \to^n y) \wedge t_i(x) \wedge t_j(y)\,.$$

Without wildcard we cannot express $\to^n$, but we can circumvent this obstacle using $\overset{+}{\to}$ if we modify our encoding properly. We encode the tile t_i as the word

$$w_i = \triangleright \bar{a} a^i b a^j \bar{a} \triangleleft$$

with $\bar{a} = a a^k$ and $i + j + 1 = k$. For the DTD we take $r \to w_0 W^* w_k$, where $W = \{w_i \mid i = 1, 2, \ldots, k\}$. The patterns are replaced with

$$x_1 \xrightarrow{w_i} x_1' \to x_2 \xrightarrow{w_j} x_2'\,,$$

$$x_1 \xrightarrow{w_i} x_1' \to x_2 \xrightarrow{w_*} x_2' \to \ldots \to x_n \xrightarrow{w_*} x_n' \to x_{n+1} \xrightarrow{w_j} x_{n+1}'\,,$$

where $x \xrightarrow{w_i} x'$ is a pattern that says that the segment of the word from position x to x' is w_i and $x \xrightarrow{w_*} y$ is the following pattern (see also Fig. 2)

$$(x \xrightarrow{\triangleright\bar{a}} x'') \wedge (x' \xrightarrow{\bar{a} b \bar{a}} y') \wedge (y'' \xrightarrow{\bar{a}\triangleleft} y) \wedge (x \overset{+}{\to} x' \overset{+}{\to} x'') \wedge (y'' \overset{+}{\to} y' \overset{+}{\to} y)\,. \qquad \square$$

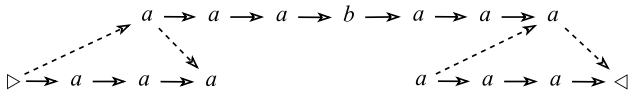

Fig. 2. Pattern $x \xrightarrow{w_*} y$ for $k = 2$

Thus BC-SAT($\Rightarrow$) is PSPACE-complete if either wildcard or $\xrightarrow{+}$ and arbitrary joins are allowed. If we forbid wildcards and restrict the use of joins, the complexity drops to NP. Using $\textsc{Match}_\pi$ and the following observation, we prove the upper bound for a relatively large class of patterns, extending tree patterns.

Lemma 3. *For all $w_1, w_2, \ldots, w_n \in \Gamma^*$ there is a linear-size deterministic automaton recognizing $\bigcup_{i=1}^{n} \Gamma^* w_i \Gamma^*$. One can compute it in* PTIME.

Proof. States of the automaton are prefixes of w_i's. After reading u, the state is the longest prefix that is a suffix of u. If the prefix is the whole word w_i, the automaton moves to a distinguished accepting state. □

Theorem 5. *For patterns whose firm sub-patterns do not contain $\xrightarrow{+}$, the problem* BC-SAT($\rightarrow, \xrightarrow{+}$) *on words is* NP*-complete.*

Proof. The lower bound is proved in [9]. To get the upper bound, we prove a polynomial model property. Given that $\Rightarrow$-patterns can be evaluated in PTIME, the proposition follows.

Let $w \in L(\mathcal{A})$ be a word satisfying a Boolean combination φ. For each pattern π in φ consider its *earliest partial matching*, i.e., the partial matching computed by $\textsc{Match}_\pi$. Clearly, π is satisfied if and only if its earliest partial matching is total. It suffices to show that the segments of w outside of the partial matches can be chosen small, without changing the matches.

Suppose that $w = u_1 v u_2$ and v is not touched by the partial matchings. A partial matching is earliest if and only if each firm sub-pattern π_0 is matched at its first occurrence after the *launching point*: the latest position i such that matching π_0 at i (regardless of labels) violates some $\xrightarrow{+}$ edge entering π_0. When shortening v we only need to make sure that we do not introduce an occurrence of a subpattern between its launching point and its original match in w. For subpatterns matched in u_1 changing v makes no difference. Suppose π_0 is matched in u_2. Where can the launching point of π_0 be? If it is enforced by a sub-pattern matched in u_1, it is in u_1. If it is enforced by a subpattern matched in u_2, it is either in u_2 or within the last $|\pi_0|$ positions of v.

Let $\pi_1, \pi_2, \ldots, \pi_k$ be all sub-patterns matched in u_2 whose launching points are in u_1. Since they contain no $\xrightarrow{+}$ nor $\_$, they can be turned into single words by merging along the $\rightarrow$ edges. Let $\mathcal{B}$ be the deterministic automaton accepting words that contain some π_i (Lemma 3). Let $v = v_1 v' v_2$, where $|v_1| = |v_2|$ is equal to the maximal size of a firm subpattern. By standard pumping we can shorten v' to at most $\|\mathcal{A}\| \cdot \|\mathcal{B}\|$, without introducing new occurrences of π_i's in

vu_2. Since we are not touching v_1, we do not introduce new occurrences of π_i's in the whole word w. Similarly, since we are not touching v_2, the patterns whose launching points are in v_2u_2 are not influenced either. □

5 Back to trees

We now lift the restriction on models and see what happens for trees. We have already seen that BC-SAT for full CQs is exponentially harder than for tree patterns: 2ExpTime versus ExpTime, and ExpSpace versus PSpace under non-recursive schemas (Theorems 1–3). Here we consider several restrictions on CQs and schemas that lower the complexity of CQs to that of tree patterns.

We show first that, for vertical tree patterns extended with arbitrary horizontal CQs over siblings, our PSpace algorithm for BC-SAT on words can be incorporated into the procedures for tree patterns without increasing their complexity. (Allowing joins with arbitrary horizontal CQs would immediately violate the intended tree structure of the vertical part of the pattern.) We say that a pattern is *forest-like* if its $\Downarrow$-subgraph is a disjoint union of trees and all vertical edges coming to the same connected $\Rightarrow$-subpattern originate in the same vertex.

Theorem 6. *For forest-like patterns,* BC-SAT$(\Downarrow, \Rightarrow)$ *is* ExpTime*-complete, and under non-recursive schemas it is* PSpace*-complete.*

Proof. For a forest-like pattern π we shall construct an equivalent deterministic automaton $\mathcal{A}_\pi$, whose states and transitions can be generated in PSpace. (Recall that a tree automaton is (bottom-up) deterministic, if for all $q_1, q_2 \in Q$ and $a \in \Sigma$ there exists exactly one state q such that $(q_1, q_2, a, q) \in \delta$.) Using this construction one can reduce BC-SAT$(\Downarrow, \Rightarrow)$ to nonemptiness of tree automata in PSpace. Both upper bounds follow, since nonemptiness of $\mathcal{A}$ over trees of depth d can be tested in space $\mathcal{O}(d \cdot \log \|\mathcal{A}\|)$, and over arbitrary trees in PTime.

A *horizontal component* of π is a connected component of the $\Rightarrow$-subgraph of π. Let $H_\pi = \langle V_\pi, \downarrow, \downarrow^+ \rangle$ be a graph over horizontal components of π, where edge $\pi_1 \downarrow \pi_2$ is present if $x \downarrow y$ for some $x \in \pi_1$ and $y \in \pi_2$, and $\pi_1 \downarrow^+ \pi_2$ is present if $x \downarrow^+ y$ for some $x \in \pi_1$ and $y \in \pi_2$, but there is no edge $\pi_1 \downarrow \pi_2$. Since π is forest-like, this graph is a forest. The subtree of H_π rooted at π_1 defines a subpattern of π, denoted by $(\pi_1)_\Downarrow$. We call such subpatterns *subtrees of* π.

The automaton $\mathcal{A}_\pi$, after reading the sequence of children of a node v, passes to v information about subtrees of π that were matched in the children of v and those that were matched in the children of some descendant of v. The automaton accepts, if the information passed from the root says that π was matched. To compute the information to be passed to v the automaton needs to aggregate the information passed from v's grandchildren to their parents. This is done by a modified version of Match working over an extended alphabet, described below.

A subtree $(\pi_1)_\Downarrow$ of π can be viewed as a horizontal pattern obtained from π_1 by including in the label of each vertex x the information about the subtrees of π to which x is connected by $\downarrow$ and $\downarrow^+$ edges. At each step Match is fed with a symbol that consists of the label of a tree node u and the information passed to

u from its sequence of children. (At the leaf level of T this information is void and MATCH works just like for words.) MATCH is only altered in this way that a vertex labelled with an extended label σ can be matched in a position labelled with an extended label τ if the original labels agree and all patterns listed in σ are also listed in τ (keeping the distinction between patterns connected by $\downarrow$ and $\downarrow^+$). It is straightforward to check that this does not influence correctness of MATCH. Observe that the extended alphabet is exponential, but each symbol can be stored in polynomial memory. Hence, MATCH still works in memory polynomial in the size of the pattern.

This procedure can be easily implemented by an exponential deterministic tree automaton. Within a sequence of children, $\mathcal{A}_\pi$ behaves like the automaton implementing MATCH($\tilde{\pi}$), where $\tilde{\pi}$ is the disjoint union of all subtrees of π. It reads the extended label from the label of the current child u and the state coming from the children of u. When the last child is read, the information about matched subtrees of π is complete and can be passed up, to the parent. □

From this result we obtain further upper bounds. For purely horizontal patterns, a standard pumping argument allows us to bound the height of the witnessing tree by the size of the schema, and with some care the algorithm for non-recursive schemas can be used even if the original schema is recursive.

Corollary 1. BC-SAT($\Rightarrow$) *is* PSPACE*-complete.*

Furthermore, as observed in [16], each pattern using no $\downarrow^+$ can be turned in PTIME into an equivalent forest-like pattern by simply merging each pair of vertices that have outgoing $\downarrow$ edges to the same connected $\Rightarrow$-subpattern. Hence, we immediately get the following corollary (hardness from Theorem 1).

Corollary 2. BC-SAT($\downarrow, \Rightarrow$) *is* EXPTIME*-complete and* PSPACE*-complete under non-recursive schemas.*

Finally, with a little more effort one can prove that in the presence of a non-recursive DTD the same holds for patterns that do not use wildcard.

Corollary 3. BC-SAT($\downarrow, \downarrow^+, \rightarrow, \overset{+}{\rightarrow}$) *is* PSPACE*-complete under non-rec. DTDs.*

The lower bound follows from Theorem 1 and the upper bound relies on the fact that in the presence of a non-recursive DTD labels come in a fixed order in the paths. For non-recursive tree automata this is no longer the case. In fact, under such schemas one can carry over the lower bounds of Theorem 3 to the case without wildcard.

6 Disjunction-free DTDs

Theorem 3 shows that under non-recursive schemas BC-SAT does not get much easier. We now introduce another restriction, often used in combination with non-recursivity in complex data management tasks [2, 3]: we limit the use of

disjunction. A DTD is *disjunction-free* if its regular expressions use only concatenation, Kleene star and the operator $\alpha^{\leq m} = (\varepsilon \mid \alpha \mid \alpha^2 \mid \ldots \mid \alpha^m)$.

BC-SAT under disjunction-free DTDs is not easier unless $\rightarrow$ is forbidden. Indeed, using $\rightarrow$ we can simulate full DTDs, e.g., a production $a \rightarrow \alpha \mid \beta$ can be simulated by $a \rightarrow \sharp(\triangleright\alpha\triangleleft)^*(\triangleright\beta\triangleleft)^*\sharp$, with conjunct $\neg\exists x\, \exists y \left(\sharp(x) \rightarrow \sharp(y)\right) \wedge \neg\exists x\, \exists y \left(\triangleleft(x) \rightarrow \triangleright(y)\right)$ added to the combination tested for satisfiability.

If $\rightarrow$ is forbidden, the complexity under disjunction-free non-recursive DTDs drops to low levels of the polynomial hierarchy, compared to NExpTime for non-recursive DTDs allowing disjunction (Theorem 3).

Theorem 7. *Under non-recursive disjunction-free DTDs* $\mathsf{BC\text{-}SAT}(\Downarrow, \overset{+}{\rightarrow})$ *is* Σ_2P*-complete and* NP*-complete for tree patterns.*

The problem is Σ_2P-hard already for Boolean combinations of the form $\pi_1 \wedge \neg\pi_2$ where π_1 is a pattern with a single node, but without π_1 it is coNP-complete.

If the non-recursivity restriction is lifted the complexity is still (potentially) below the general 2ExpTime lower bound.

Theorem 8. *Under disjunction-free DTDs* $\mathsf{BC\text{-}SAT}(\Downarrow, \overset{+}{\rightarrow})$ *is in* NExpTime *and* PSpace*-complete for tree patterns. The lower bound holds already for containment of unions of tree patterns using only* $\downarrow, \_$.

7 Conclusions

We have shown that under several independent restrictions, CQs have the same complexity of the satisfiability of Boolean combinations, and the containment of unions of queries problem, as tree patterns. Most importantly, vertical tree patterns can be extended with full horizontal CQs over children without increasing the complexity of static analysis tasks. We have also showed that under non-recursive, disjunction-free schemas the complexity of static analysis for CQs without the next-sibling relation is in low levels of the polynomial hierarchy. This could be applied in the analysis of mappings between nested-relational schemas [2]. (We point out the complexity gap for general disjunction-free schemas as an elegant theoretical challenge.) We focused on containment of UCQs, since this is the problem relevant for XML metadata management, but a finer analysis of the containment for CQs would also be desired (the 2ExpTime-lower bound of [9] holds already for validity of CQs). Similarly, patterns with data comparisons might be considered (again, some cases are settled in [9]).

Acknowledgements. This work is part of the *Querying and Managing Navigational Databases* project realized within the Homing Plus programme of the Foundation for Polish Science, co-financed by the European Union from the Regional Development Fund within the Operational Programme Innovative Economy ("Grants for Innovation"). We thank Claire David for inspiring discussions and careful reading of a preliminary version of this paper, and the anonymous referees for helpful comments motivating us to improve the presentation of the paper.

References

1. S. Amano, C. David, L. Libkin, F. Murlak. On the Tradeoff between Mapping and Querying Power in XML Data Exchange. ICDT 2010: 155–164.
2. S. Amano, L. Libkin, F. Murlak. XML schema mapping. PODS 2009: 33–42.
3. M. Arenas, L. Libkin. XML data exchange: consistency and query answering. *J. ACM* 55(2), 2008.
4. M. Benedikt, P. Bourhis, P. Sennelart. Monadic Datalog Containment. ICALP 2012 (to appear).
5. M. Benedikt, W. Fan, F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM* 55(2), 2008.
6. G. J. Bex, F. Neven, J. Van den Bussche. DTDs versus XML Schema: a practical study. WebDB 2004: 79–84.
7. H. Björklund, W. Gelade, W. Martens. Incremental XPath evaluation. *ACM Trans. Database Syst.* 35(4): 29 (2010)
8. H. Björklund, W. Martens, T. Schwentick. Conjunctive query containment over trees. DBPL 2007: 66–80.
9. H. Björklund, W. Martens, T. Schwentick. Optimizing conjunctive queries over trees using schema information. MFCS 2008: 132–143.
10. M. Bojańczyk, L. A. Kołodziejczyk, F. Murlak. Solutions in XML data exchange. ICDT 2011: 102–113.
11. B. ten Cate, C. Lutz. The Complexity of Query Containment in Expressive Fragments of XPath 2.0. *J. ACM* 56(6): 1–48.
12. A. K. Chandra, P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. STOC 1977: 77–90.
13. C. David. Complexity of data tree patterns over XML documents. MFCS 2008: 278–289.
14. C. David, L. Libkin, F. Murlak. Certain answers for XML queries. PODS 2010: 191–202.
15. R. Fagin, Ph. Kolaitis, R. Miller, L. Popa. Data exchange: semantics and query answering. *Theor. Comp. S.* 336 (2005), 89–124.
16. G. Gottlob, C. Koch, K. Schulz. Conjunctive queries over trees. *J. ACM* 53 (2006), 238–272.
17. J. Hidders. Satisfiability of XPath expressions. DBPL 2003: 21–36.
18. Y. Ishihara, T. Morimoto, S. Shimizu, K. Hashimoto, T. Fujiwara. A Tractable Subclass of DTDs for XPath Satisfiability with Sibling Axes. DBPL 2009: 68–83.
19. M. Lenzerini. Data integration: a theoretical perspective. PODS 2002: 233–246.
20. M. Marx. XPath with conditional axis relations. EDBT 2004: 477–494.
21. G. Miklau, M. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM* 51(1): 2–45, 2004.
22. M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4): 1–45, 2005.
23. F. Neven. Automata Theory for XML Researchers. *SIGMOD Record* 31(3): 39–46 (2002).
24. F. Neven, T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Meth. in Comp. Sci.* 2(3): 1–30, 2006.
25. P. T. Wood. Containment for XPath fragments under DTD constraints. ICDT 2003.

A Lower bounds of Theorem 1

A.1 Syntax for tree patterns

Tree patterns can be specified in the following natural syntax [2]:

$$\begin{array}{ll} \pi ::= \sigma[\lambda] & \text{patterns} \\ \lambda ::= \varepsilon \mid \pi \mid //\pi \mid \mu \mid \lambda, \lambda & \text{lists} \\ \mu ::= \pi \mid \pi \to \mu \mid \pi \xrightarrow{+} \mu & \text{sequences} \end{array}$$

where $\sigma \in \Gamma \cup \{\_\}$. That is, a tree pattern is given by its root node with label σ and a list of subtrees λ. A subtree can be rooted at a child of the root (corresponding to π in the definition of λ), or its descendant (corresponding to $//\pi$). The subtrees rooted in the children of the root can be additionally connected into sequences μ by means of the next sibling relation (represented by $\to$) or the following sibling relation (represented by $\xrightarrow{+}$). The wildcard symbol $\_$ is used to denote a vertex without a label. We write σ instead of $\sigma[\,]$.

A.2 BC-SAT(↓) for tree patterns is PSpace-hard under non-recursive DTDs

We reduce the following problem of acceptance by Alternating Turing Machine (ATM) in polynomial time: Given ATM $\mathcal{M}$ and number n in unary, does $\mathcal{M}$ accept empty input in no more than n steps?

An ATM can be presented as a tuple $\langle Q, A, q_0, \delta\rangle$, where the state space Q is split into universal states $Q_\forall$, existential states $Q_\exists$, the accepting state q_a and rejecting state q_r. The initial state is $q_0 \in Q_\exists$. Alphabet A contains special blank symbol $\flat$ and $\delta \subset Q \times A \times Q \times A \times \{-1, 0, 1\}$ is the transition relation.

Without loss of generality we can assume that both accepting and rejecting configurations are halting configurations (have no successor configurations) and that every non-halting configuration has exactly two different successor configurations. Moreover we can assume that $\mathcal{M}$ is normalized that is transitions from universal states affect only the state of the machine (tape is not changed). Since the computation takes no more than n steps, $\mathcal{M}$ uses no more than n memory cells.

Acceptance for alternating Turing machine is based on computation trees. An accepting computation tree is finite and has the accepting state q_a in each leaf. We encode each configuration as a sequence of children of the *configuration header* node. The configuration header nodes form a binary tree. The root is labeled with c_1. If a configuration header node labelled with c_i or d_i stores a universal state, then it has two children, c_{i+1} and d_{i+1}. If it stores an existential node, it has just one child, c_{i+1}. If it stores a final state (accepting or rejecting), it has no children.

Our reduction will enforce that every tree satisfying the Boolean combination of patterns encodes an accepting run (computation tree) of the machine. DTD will provide the skeleton for the encoding and patterns will enforce that the

computation is accepting, starts in correct configuration, and that transitions are consistent with δ.

The root symbol of the DTD is S and the productions are as follows:

$$\begin{aligned} S &\to c_1 \\ c_i, d_i &\to a_1, ..., a_n(q_a|q_r|Q_\exists c_{i+1}|Q_\forall c_{i+1} d_{i+1}) \qquad \text{for } 0 < i < n \\ c_n, d_n &\to a_1, ..., a_n(q_a|q_r) \\ a_i &\to \Sigma \end{aligned}$$

where elements of $\Sigma = A \times \{\top, \bot\}$ encode the content of cells, $\top$ indicates the presence of the head. By $\hat{\Sigma}$ we will denote encodings of not empty cells, i.e., $\hat{\Sigma} = \Sigma \setminus (\{\flat\} \times \{\top, \bot\})$ and C denotes the set of configuration headers $\{c_i, d_i \mid i = 1, \ldots, n\}$.

Note that provided schema ensures that in a computation an existential configuration chooses one of possible successors, universal two and that final configurations are halting (ends the computation).

The correctness of the encoding is expressed by patterns as follows. Universal configuration has two different child configurations ($lab \in C; q_\forall \in Q_\forall; q \in Q; 1 \leq i \leq n$):

$$\neg\ lab\big[q_\forall, c_i[q], d_i[q]\big]$$

After an empty cell there are only empty cells ($lab \in C, l \in \hat{\Sigma}, h \in \{\top, \bot\}, 1 \leq i < j \leq n$):

$$\neg\ lab\big[a_i[\langle \flat, h\rangle], a_j[l]\big]$$

There is only one head in each configuration ($a, b \in A; lab \in C; 1 \leq i < j \leq n$):

$$\neg\ lab\big[a_i[\langle a, \top\rangle], a_j[\langle b, \top\rangle]\big]$$

The head does not disappear (or does not move further than one position) when configuration changes ($1 < i < n$; $lab, lab' \in C$; $a, b, c, d \in A$):

$$\neg\ lab\big[a_i[\langle a, \top\rangle], lab'[a_{i-1}[\langle b, \bot\rangle], a_i[\langle c, \bot\rangle], a_{i+1}[\langle d, \bot\rangle]]\big]$$

and for $i = 1$ or $i = n$ we drop $a_{i-1}[\langle b, \bot\rangle]$ or $a_{i+1}[\langle d, \bot\rangle]$, respectively.

Memory change involves head ($1 \leq i \leq n$; $lab, lab' \in C$; $a, b \in A$; $h \in \{\bot, \top\}$):

$$\neg\ lab\big[a_i[\langle a, \bot\rangle], lab'[a_i[\langle b, h\rangle]]\big] \qquad \text{for } a \neq b\,.$$

Transition is consistent with δ ($0 \leq i \leq n$; $lab, lab' \in C$):

$$\neg\ lab\left[q_1, a_i[l_1], lab'\left[q_2, a_i[l_2], a_{i+d}[l_3]\right]\right]$$

for

- $l_1 = \langle a, \top\rangle, l_2 = \langle b, \top\rangle, l_3 = l_2, d = 0$, and $\langle q_1, a, q_2, b, d\rangle \notin \delta$
- $l_1 = \langle a, \top\rangle, l_2 = \langle b, \bot\rangle, l_3 \in A \times \{\top\}, d \in \{-1, 1\}, a, b \in A$, and $\langle q_1, a, q_2, b, d\rangle \notin \delta$

For $i = 1$ and $i = n$ we drop variable $a_{i-1}[\dots]$ or $a_{i+1}[\dots]$, respectively.

Computation is accepting (no rejecting configurations):

$$\neg\, q_r,$$

starts with an empty tape, with the head over the first cell:

$$\neg\, S\,[c_1[a_1[\langle a, h\rangle]]] \qquad \text{for } a \neq \flat \text{ or } h \neq \top$$

and in initial state:

$$\neg\, S\,[c_1[q]] \qquad \text{for } q \neq q_0.$$

It is easy to check that $\mathcal{M}$ accepts empty input if and only if the conjunction of the negated patterns above is satisfiable with respect to the given DTD.

A.3 BC-SAT(↓) for tree patterns is ExpTime-hard

We obtain a reduction from the acceptance by an alternating Turing machine in polynomial space by a simple modification of the reduction given in the previous subsection. We skip the indices in the configuration headers to get the following DTD:

$$\begin{aligned} S &\to c \\ c, d &\to a_1, ..., a_n(q_a|q_r|Q_\exists c|Q_\forall cd) \\ a_i &\to \Sigma \end{aligned}$$

and allow arbitrarily long computations which use n memory cells. Similar change in patterns is enough to obtain the desired reduction.

B Proof of Theorem 3

B.1 BC-SAT(⇓, →) is ExpSpace-hard under non-recursive schemas

We give a reduction from the acceptance by Turing machine in exponential space: given Turing machine $M = \langle A, Q, \delta, q_0, q_f\rangle$ and number n in unary, does M accept empty input using no more than 2^n memory cells? We assume that A contains the blank symbol $\flat$ and that there is only one initial state q_0 and one accepting state q_f.

The idea is to enforce a tree whose designated sub-trees encode consecutive configurations of an accepting computation of M. DTD will create the skeleton to code computation, and patterns will take care of correctness. A configuration consists of a state ($q \in Q$) of the machine, head position, and tape content. Single configuration will be encoded in a tree with c-labelled root. Its leaves will represent tape cells and will have labels from set $C = A \times (Q \cup \{\bot\})$. If in a given configuration the label of a cell-encoding node is of form $\langle a, q\rangle \in A \times Q$ then the head is over that cell. W.l.o.g. we can assume that every configuration of M is of length 2^n.

For given n the skeleton of the computation is fixed by a DTD D of the form:

$$\begin{aligned}
r &\to R \\
R &\to \dagger c^* \ddagger \\
c &\to l_1 r_1 \\
l_i &\to 0 b_i && 1 \le i \le n \\
r_i &\to 1 b_i && 1 \le i \le n \\
b_i &\to l_{i+1} r_{i+1} && 1 \le i < n \\
b_n &\to C \\
0 &\to 1
\end{aligned}$$

We will say that node is on level i if its label has index i.

To express correctness of the run, we shall need some auxiliary patterns. We construct them step by step.

The formula $is\_root(x)$ states that x is one of the roots: either the real one, labelled with r, or the fake one, labelled with R:

$$is\_root(x) = \exists\, y \colon (x \downarrow^{2n+2} y)\,.$$

The correctness of the formula follows from the fact that the height of each tree conforming to the DTD D is $2n+3$, and a path of length $2n+2$ exists only from the node labelled with r or the one labelled with R.

Next, we write a formula $same\_turn_i$ saying that two nodes x, y are on level i and both are labelled with l_i or both are labelled with r_i. Since we cannot use disjunction, it requires a trick using the fake root.

$$same\_turn_i(x,y) = \exists x, y, root, x_c, y_c, x_d, y_d :$$

$$b_i(x_c) \wedge b_i(y_c) \wedge 1(x_d) \wedge 1(y_d) \wedge is\_root(root) \wedge$$

$$\wedge\ (x \downarrow x_c) \wedge (x \downarrow^+ x_d) \wedge (y \downarrow y_c) \wedge (y \downarrow^+ y_d) \wedge$$

$$\wedge\ (root \downarrow^{2i+2} x_d) \wedge (root \downarrow^{2i+2} y_d)$$

We know that $root$ is labelled with r or R. If it is r, then x_d and y_d must be children of x and y, respectively. By the definition of the DTD D, x and y must have label r_i. In the second case, $root$ is labelled with R and x_d and y_d are grandchildren of x and y, respectively. Then, x and y, both have label l_i.

Now we can write a formula saying that x, y encode the same cell in two subsequent configurations. We express this by saying that they can be reached from their c-labelled ancestors along paths of length $2n+1$ that take the same turns, left or right:

$$same\_cell(x,y) = \exists x_c, x_1, .., x_n, y_c, y_1, ..y_n :$$

$$c(x_c) \wedge c(y_c) \wedge (x_c \to y_c) \wedge (x_c \downarrow x_1) \wedge (y_c \downarrow y_1) \wedge$$

$$\wedge \bigwedge_{i=1}^{n-1} (x_i \downarrow^2 x_{i+1}) \wedge (y_i \downarrow^2 y_{i+1}) \wedge$$

$$\wedge (x_n \downarrow^2 x) \wedge (y_n \downarrow^2 y) \wedge$$

$$\wedge \bigwedge_{i=1}^{n} \mathit{same\_turn}_i(x_i, y_i)$$

Second, we need a formula $\mathit{next\_cell}_k(x, y)$ expressing that x and y are neighbouring leaves (x is to the left of y) and have the closest common ancestor on level k (labelled b_k):

$$\mathit{next\_cell}_k(x, y) = \exists z, x_{k+1}, .., x_n, y_{k+1}, .., y_n :$$

$$\bigwedge_{i=k+1}^{n-1} (x_i \downarrow^2 x_{i+1} \wedge y_i \downarrow^2 y_{i+1}) \wedge$$

$$\wedge (z \downarrow x_{k+1}) \wedge (z \downarrow y_{k+1}) \wedge (x_n \downarrow^2 x) \wedge (y_n \downarrow^2 y) \wedge$$

$$\wedge b_k(z) \wedge l_{k+1}(x_{k+1}) \wedge r_{k+1}(y_{k+1}) \wedge \bigwedge_{i=k+2}^{n} (r_i(x_i) \wedge l_i(y_i))$$

Now we can express that transition between two consecutive configurations is compatible with δ. More specifically we exclude every incorrect transition. To find an error we need 3 neighbouring cells in two consecutive configurations. When we grasp them, we simply say that their labels encode an incorrect transition, i.e., that one of the following happened:

- cell content changed without the head being involved,
- the head disappeared,
- the change of head position, state, or cell content is inconsistent with transition relation δ.

This is expressed as follows:

$$\neg\exists x_1, x_2, x_3, y_1, y_2, y_3 :$$

$$\mathit{next\_cell}_k(x_1, x_2) \wedge \mathit{next\_cell}_{k'}(x_2, x_3) \wedge$$

$$\wedge \mathit{next\_cell}_k(y_1, y_2) \wedge \mathit{next\_cell}_{k'}(y_2, y_3) \wedge$$

$$\wedge \mathit{same\_cell}(x_1, y_1) \wedge \bigwedge_{i=1}^{3} (a_i(x_i) \wedge b_i(y_i)),$$

where a_1, a_2, a_3 and b_1, b_2, b_3 range over triples of labels from C corresponding to incorrect transitions, and k, k' ranges over $1, 2, \ldots, n-1$.

It remains make sure that the computation begins and ends correctly. The first configuration should contain empty tape and have the head in the first cell,

in the initial state. This is enforced by the conjunction of the following formulae ($\langle\sigma, q\rangle \in C \setminus \{\langle\flat, q_0\rangle\}$ and $\langle\sigma', q'\rangle \in C \setminus \{\langle\flat, \bot\rangle\}$)):

$$\neg\, \exists x, y \colon \mathit{first\_conf}(x) \wedge (x \downarrow^+ y) \wedge \mathit{first\_cell}(x) \wedge \langle\sigma, q\rangle(y)\,,$$

$$\neg\, \exists x, y, z : \mathit{first\_conf}(x) \wedge (x \downarrow^+ y) \wedge \mathit{next\_cell}_k(y, z) \wedge \langle\sigma', q'\rangle(z)\,,$$

where k ranges over $1, 2, \ldots, n$ and the auxiliary patterns are defined as follows:

$$\mathit{first\_conf}(x) = \exists y : \dagger(y) \wedge (y \to x)\,,$$

$$\mathit{first\_cell}(x) = \exists y_1, \ldots, y_n : \left(\bigwedge_{i=1}^{n} l_i(y_i)\right) \wedge \left(\bigwedge_{i=1}^{n-1} (y_i \downarrow^2 y_{i+1})\right) \wedge (y_n \downarrow^2 y)\,.$$

To check that the last configuration is accepting we use conjunction of the following formulae ($q \in Q \setminus \{q_f\}$ and $a \in A$):

$$\neg\exists x, y : \mathit{last\_conf}(x) \wedge (x \downarrow^+ y) \wedge \langle a, q\rangle(y)\,,$$

where $\mathit{last\_conf}(x) = \exists z : (x \to z) \wedge \ddagger(z)$.

Disjunction-free DTDs The DTD used in the reduction only contains disjunction in the production $b_n \to C$. We can replace this production with $b_n \to \sharp(\triangleright c_1 \triangleleft)^*(\triangleright c_2 \triangleleft)^* \ldots (\triangleright c_m \triangleleft)^* \sharp$, where $\{c_1, c_2, \ldots, c_m\} = C$, and adding conjunct $\neg\exists x\, \exists y \left(\sharp(x) \to \sharp(y)\right) \wedge \neg\exists x\, \exists y \left(\triangleleft(x) \to \triangleright(y)\right)$ to the combination tested for satisfiability. As children of b_n nodes are never tested for $\to$, this does not interfere with other patterns.

Similarly, in the 2ExpTime reduction of [9], the patterns do not use horizontal axes at all, so eliminating disjunction in a similar fashion does not influence other patterns.

B.2 BC-SAT($\Downarrow, \overset{+}{\to}$) is in NExpTime under non-recursive schemas

We exploit the following simple observation [2, 3].

Lemma 4. *Let π be a pattern that does not use $\to$. Assume that $\mathcal{T} \models \pi$ and let $h : \pi \to \mathcal{T}$ be a homomorphism witnessing the satisfiability. If $\mathcal{T}'$ is obtained from $\mathcal{T}$ by removing a subtree disjoint from $h(\pi)$; or adding a subtree, then $\mathcal{T}' \models \pi$.*

Proof. The proof is straightforward: none of above operations interferes with h so in the new tree h still proves satisfiability. □

Using Lemma 4 we show that if a Boolean combination of patterns is satisfiable with respect to a schema $\mathcal{S}$, then there is a witnessing tree of exponential size.

Take any $\mathcal{T} \in L(\mathcal{S})$ such that $\mathcal{T} \models \varphi$. W.l.o.g. we may assume that φ is of form $\varphi = \pi \wedge \bigwedge_{i=1}^{k} \neg\pi_i$, for some $k \leq |\varphi|$, and $\mathcal{S}$ is represented by non-deterministic

automaton $\mathcal{A}$. Let $h : \pi \to \mathcal{T}$ be a witnessing homomorphism. Consider a node u in $\mathcal{T}$ with children $u \cdot 0, \ldots, u \cdot m$. Let a be the label of u and q the state mapped by r. Let $u_0, .., u_l$ be an ascending sub-sequence u's children, such that $u_0 = u0$, $u_l = u \cdot m$ and every other node in the sequence either is in $h(\pi)$ or is an ancestor of a node in $h(\pi)$. Then, $l \leq |\pi| + 2$. By Lemma 4 and standard pumping we can assume that between each two consecutive u_i's there is no more than $\|\mathcal{A}\|$ nodes. Hence, u has no more than $\|\mathcal{A}\|(|\varphi| + 2)$ children. Thus there exists a witnessing tree with polynomial branching. Since the height of trees is bounded by $\|\mathcal{S}\|$, we obtain an exponential witness.

The NExpTime upper bound follows immediately. Indeed, each pattern can be can be translated to an non-deterministic tree automaton of exponential size [9]. We guess a "valuation" of patterns that satisfies the Boolean combination, and guess a witnessing tree of exponential size. Then we evaluate the automata obtained from patterns on the guessed tree checking if the patterns are satisfied or not satisfied, according to the valuation. (Note that we can run an exponential non-deterministic automaton over an exponential tree in ExpTime.)

B.3 BC-SAT(⇓) is NExpTime-hard

We reduce the acceptance of non-deterministic Turing machines in exponential time: Given a non-deterministic Turing Machine M, and an integer n in unary, decide if M accepts the empty input in exactly 2^n steps.

The reduction is very similar to the one used in Section B.1, but since $\to$ is not allowed in patterns, we have to find new way to express that two configurations are consecutive. To do this, we replace $R \to \dagger c^* \ddagger$ with a binary tree with configurations in its leaves. Then we build a formula *next_conf* similar to *next_cell*. Note that it limits the number of configurations exponentially, as opposed to potentially super-exponential number of configurations in the DTD from the reduction in Section B.1.

Like before, $C = A \times (Q \cup \{\bot\})$. The DTD is

$$\begin{aligned}
r &\to R \\
R &\to l_1 r_1 \\
l_i &\to 0 b_i && 1 \leq i \leq 2n \\
r_i &\to 1 b_i && 1 \leq i \leq 2n \\
b_i &\to l_{i+1} r_{i+1} && 1 \leq i < 2n \\
b_{2n} &\to C \\
0 &\to 1
\end{aligned}$$

The auxiliary formulae *is_root*(x), *same_turn*$_i(x, y)$, and *next_cell*$_k(x, y)$ require cosmetic changes related with the fact that trees have now additional $2n$ levels, c-nodes are gone, and their role is taken by nodes with labels b_n:

$$\mathit{is\_root}(x) = \exists\, y\colon (x \downarrow^{4n+1} y)$$

$$same\_turn_i(x,y) = \exists x, y, root, x_c, y_c, x_d, y_d :$$
$$b_i(x_c) \wedge b_i(y_c) \wedge 1(x_d) \wedge 1(y_d) \wedge is\_root(root) \wedge$$
$$\wedge\ (x \downarrow x_c) \wedge (x \downarrow^{+} x_d) \wedge (y \downarrow y_c) \wedge (y \downarrow^{+} y_d) \wedge$$
$$\wedge\ (root \downarrow^{2i+1} x_d) \wedge (root \downarrow^{2i+1} y_d)$$

$$next\_cell_k(x,y) = \exists z, x_{k+1}, .., x_{2n}, y_{k+1}, .., y_{2n} :$$
$$\bigwedge_{i=k+1}^{2n-1} (x_i \downarrow^2 x_{i+1} \wedge y_i \downarrow^2 y_{i+1}) \wedge$$
$$\wedge\ (z \downarrow x_{k+1}) \wedge (z \downarrow y_{k+1}) \wedge (x_{2n} \downarrow^2 x) \wedge (y_{2n} \downarrow^2 y) \wedge$$
$$\wedge\ b_k(z) \wedge l_{k+1}(x_{k+1}) \wedge r_{k+1}(y_{k+1}) \wedge \bigwedge_{i=k+2}^{2n} (r_i(x_i) \wedge l_i(y_i))$$

The formula $same\_cell_k(x,y)$ uses a subformula $next\_conf_k(x_c, y_c)$ instead of $x_c \to y_c$, to express that x_c and y_c are roots of subtrees encoding two subsequent configuration:

$$same\_cell_k(x,y) = \exists x_c, x_1, .., x_n, y_c, y_1, ..y_n :$$
$$b_n(x_c) \wedge b_n(y_c) \wedge next\_conf_k(x_c, y_c) \wedge (x_c \downarrow x_1) \wedge (y_c \downarrow y_1) \wedge$$
$$\wedge \bigwedge_{i=1}^{n-1} (x_i \downarrow^2 x_{i+1}) \wedge (y_i \downarrow^2 y_{i+1}) \wedge$$
$$\wedge\ (x_n \downarrow^2 x) \wedge (y_n \downarrow^2 y) \wedge$$
$$\wedge \bigwedge_{i=1}^{n} same\_turn_i(x_i, y_i)$$

The formula $next\_conf_k(x,y)$ is simply the original version of $next\_cell_k(x,y)$, as defined in Section B.1.

The patterns enforcing correctness of the run are identical to the ones from Section B.1, except that the lower indices k, k' range over $n+1, n+2, \ldots, 2n$ (not $1, 2, \ldots, n$) and the auxiliary formulas *first_conf*, *last_conf*, and *first_cell* are modified as follows:

$$first\_conf(x) = \exists y_1, \ldots, y_n : \left(\bigwedge_{i=1}^{n} l_i(y_i) \right) \wedge \left(\bigwedge_{i=1}^{n-1} (y_i \downarrow^2 y_{i+1}) \right) \wedge (y_n \downarrow y)\,,$$

$$\mathit{last\_conf}(x) = \exists y_1, \ldots, y_n : \left(\bigwedge_{i=1}^{n} r_i(y_i) \right) \wedge \left(\bigwedge_{i=1}^{n-1} (y_i \downarrow^2 y_{i+1}) \right) \wedge (y_n \downarrow y)\,,$$

$$\mathit{first\_cell}(x) = \exists y_1, \ldots, y_n : \left(\bigwedge_{i=n+1}^{2n} l_i(y_i) \right) \wedge \left(\bigwedge_{i=1}^{n-1} (y_i \downarrow^2 y_{i+1}) \right) \wedge (y_n \downarrow^2 y)\,.$$

C Proofs of lemmas from Section 4

Lemma 1. *Let w be a word satisfying a $\Rightarrow$-pattern π.*

1. *For all $g, h \colon \pi \to w$, $\min(g, h)$ is a homomorphism.*
2. *There exists $h_{\min} \colon \pi \to w$ such that $h_{\min} \leq h$ for all $h \colon \pi \to w$.*
3. *For each set X of vertices of π and each $h \colon \pi \to w$ there is a $\hat{h} \colon \pi \to w$ extending $h|_X$ such that $\hat{h} \leq h'$ for each $h' \colon \pi \to w$ extending $h|_X$.*

Proof. Let $f = \min(g, h)$. Since g and h preserve unary relations, so does f. Let us see f also preserves binary relations. If $x \to y$ for some vertices x, y of π, then $f(y) = \min(g(y), h(y)) = \min(g(x) + 1, h(x) + 1) = 1 + \min(g(x), h(x)) = 1 + f(x)$. Suppose $x \xrightarrow{+} y$. W.l.o.g. $g(x) \leq h(x)$. Since g and h are homomorphisms, $g(x) < g(y)$ and $h(x) < h(y)$, and it follows that $g(x) < h(y)$. In consequence, $f(x) = g(x) < \min(g(y), h(y)) = f(y)$. Items (2) and (3) follow immediately. □

Lemma 2. *Let $h \colon \pi \to w$ be the earliest matching and let X be a set of vertices of π such that no edge enters X from the outside, and the only edges leaving X are $\xrightarrow{+}$. For each $Y \subseteq X$, if $g \colon \pi|_X \to w$ is the least homomorphism extending $h|_Y$ to X, then $h|_X = g$.*

Proof. By definition, $g \leq h|_X$. It remains to prove the converse inequality. Let $\tilde{g} : \pi \to w$ be a function defined by extending g to π according to h. We shall prove that $\tilde{g}$ is a homomorphism. From this it follows by minimality of h that $\tilde{g} = h$ and in consequence $g = h|_X$. Unary relations and binary relations between pairs of vertices in X or in the complement of X are preserved by $\tilde{g}$, because they are preserved by g and h. It remains to consider $x \in X$ and $y \notin X$. By the hypothesis of the lemma, the only possible relation is $x \xrightarrow{+} y$. Since $g \leq h|_X$ and h is a homomorphism, we have $\tilde{g}(x) = g(x) \leq h(x) < h(y) = \tilde{g}(y)$. □

D The greedy matching procedure $\mathrm{Match}_{\pi}(w)$

The procedure $\textsc{Match}_\pi$ is given in Algorithm 1. Note that in the main loop (lines 6–19) new firm subpatterns are added to LAUNCHED. The loop runs until all of them are processed.

Why remembering last $|\pi|$ symbols is enough? Consider the moment when π_j is added to LAUNCHED after π_i has been matched. We will say that π_i was

Algorithm 1 $\textsc{Match}_\pi(w)$

```
    compute F_π = ({π_1, ..., π_k}, ⁺→)
    MATCHED := ∅
    LAUNCHED := set of firm subpatterns of π that have no predecessors in F_π
    read first |π| letters of w into BUFFER
    ∀i CONSTRAINTS_i := ∅
 6: for all π_i ∈ LAUNCHED do
      try to find the earliest matching h: π_i → BUFFER satisfying CONSTRAINTS_i
      if matching h is found then
        move π_i from LAUNCHED to MATCHED
        for all successors π_j of π_i in F_π do
          for all x ⁺→ y with x ∈ π_i and y ∈ π_j do
12:           add y > h(x) to CONSTRAINTS_j
          end for
          if all predecessors of π_j in F_π are in MATCHED then
            add π_j to LAUNCHED
          end if
        end for
18:   end if
    end for
    if no more letters then
      if LAUNCHED = ∅ then accept else reject
    else
      ∀i scale down CONSTRAINTS_i by 1 (drop negative ones)
24:   move BUFFER one letter to the right; go to 6
    end if
```

launched by π_j. As there is an $\xrightarrow{+}$ edge from π_i to π_j, the rightmost position of π_j's match must not come before the leftmost position of π_i's match. Suppose that π_i was in LAUNCHED when the loop started. Then it could not be matched within the previous buffer, which means that its match uses the last letter of the current buffer. Hence, each match of π_j satisfying CONSTRAINTS$_j$ fits within the last $|\pi_i| + |\pi_j| \leq |\pi|$ positions of the current buffer. If π_i was also added to LAUNCHED within the current buffer, we can again find its launching subpattern, and moving backwards this way, we finally reach a firm subpattern that was in LAUNCHED before the loop started. As each launching pattern must start before the launched one ends, the whole chain of launching subpatterns must fit within a suffix of BUFFER of length bounded by the sum of their sizes, that is by $|\pi|$.

E Proof of Theorem 6 (**BC-SAT** for forest-like patterns)

E.1 PSpace algorithm for non-recursive schemas

The idea is similar to the one for words, where a word was generated non-deterministically and checked on-line for matches of patterns used in the Boolean combination. This time we generate nodes of a tree T in the depth-first order and feed them to copies of TreeMatch(π) for all patterns π used in the combination φ tested for satisfiability (and a tree automaton $\mathcal{A}_\mathcal{S}$ checking conformance to the schema). The procedure TreeMatch(π) will run a deterministic tree automaton $\mathcal{A}_\pi$ equivalent to the pattern π. (Recall that a tree automaton is (bottom-up) deterministic, if for all $q_1, q_2 \in Q$ and $a \in \Sigma$ there exists exactly one state q such that $(q_1, q_2, a, q) \in \delta$.) Just like in the word case, $\mathcal{A}_\pi$ will be exponential in $|\pi|$, but its states and transitions will be constructed on-the-fly in PTime. The procedure TreeMatch(π) will compute the states of $\mathcal{A}_\pi$ in the nodes of the tree in the post-order fashion, i.e., the node will get its state directly after all its children have had their states assigned. As the tree is given in the depth-first order, for each node on the path from the root to the current node TreeMatch(π) needs to remember the state assigned to its sibling immediately to the left (the initial state in case there is no sibling).

Since the schema $\mathcal{S}$ is non-recursive, the height of the generated tree can be bounded by $\|\mathcal{S}\|$. By standard pumping we bound the branching by $\prod_{\pi \text{ in } \varphi} \|\mathcal{A}_\pi\|$ (still single exponential in $|\varphi|$). To avoid looping the generating procedure also remembers the path from the root to the current node, and for each node on the path it remembers the number of children of this node already generated, making sure it does not exceed the exponential bound. By Savitch theorem we can eliminate non-determinism from this algorithm. It remains to see how to construct the automaton $\mathcal{A}_\pi$.

A *horizontal component* of π is any connected component of the $\Rightarrow$-subgraph of π. Let $H_\pi = \langle V_\pi, \downarrow, \downarrow^+ \rangle$ be a graph over horizontal components of π, where edge $\pi_1 \downarrow \pi_2$ is present if $x \downarrow y$ for some $x \in \pi_1$ and $y \in \pi_2$, and $\pi_1 \downarrow^+ \pi_2$ is present if $x \downarrow^+ y$ for some $x \in \pi_1$ and $y \in \pi_2$, but there is no edge $\pi_1 \downarrow \pi_2$. Since

π is forest-like, this graph is a forest. The subtree of H_π rooted at π_1 defines a subpattern of π, which we denote $(\pi_1)_\Downarrow$. Subpatterns of π of this form are called *subtrees of* π.

The automaton $\mathcal{A}_\pi$ works as follows. From the sequence of children to their parent v it passes information about:

- subtrees of π that were matched in the children of v,
- subtrees of π that were matched in the children of some descendant of v.

The automaton accepts, if the information passed from the root says that π was matched. To compute the information to be passed to a node v the automaton needs to aggregate the information passed from v's grandchildren to their respective parents. This is done by a modified version of Match working over an extended alphabet, described below.

A subtree $(\pi_1)_\Downarrow$ of π can be viewed as a horizontal pattern obtained from π_1 by including in the label of each vertex x the information about the subtrees of π to which x is connected by $\downarrow$ and $\downarrow^+$ edges. At each step Match is fed with a symbol that consists of the label of a tree node v' and the information passed to v' from its sequence of children. (At the leaf level of T this information is void and Match works just like for words.) Match is only altered in this way that a vertex labelled with an extended label σ can be matched in a position labelled with an extended label τ if the original labels agree and all patterns listed in σ are also listed in τ (keeping the distinction between patterns connected by $\downarrow$ and $\downarrow^+$). It is straightforward to check that this does not influence correctness of Match. Observe that the extended alphabet is exponential, but each symbol can be stored in polynomial memory. Hence, Match still works in memory polynomial in the size of the pattern, and the automaton implementing it still has exponentially many states.

The described procedure can be easily implemented by a tree automaton. Within a sequence of children, $\mathcal{A}_\pi$ behaves like the automaton implementing $\textsc{Match}(\tilde{\pi})$, where $\tilde{\pi}$ is obtained by taking a disjoint union of all subtrees of π. It reads the extended label from the label of the current child v' and the state coming from the children of v'. When the last child of the sequence is read, the information about matched subtrees of π is updated and can be passed up, to the parent.

E.2 ExpTime algorithm for arbitrary schemas

In the previous subsection we proved that for each forest-like pattern π there is an exponential deterministic automaton $\mathcal{A}_\pi$ equivalent to π. The automaton can easily be computed in ExpTime. From this it follows immediately that for each Boolean combination of patterns φ one can compute in ExpTime an equivalent automaton $\mathcal{A}_\varphi$. To test if φ is satisfiable with respect to a schema $\mathcal{S}$ it suffices to check if $L(\mathcal{A}_\varphi) \cap L(\mathcal{A}_\mathcal{S}) \neq \emptyset$, where $\mathcal{A}_\mathcal{S}$ is a polynomial automaton equivalent to $\mathcal{S}$. This can be done in time polynomial in $\|\mathcal{A}_\varphi\| \cdot \|\mathcal{A}_\mathcal{S}\|$.

F Proof of Corollary 3

We prove that $\mathsf{BC\text{-}SAT}(\downarrow, \downarrow^+, \rightarrow, \overset{+}{\rightarrow})$ on non-recursive DTDs is in PSPACE by showing how to convert patterns without wildcard to forest-like patterns.

Consider a pattern π using no wildcard. Without loss of generality we can assume it is a DAG. If its not a forest-like pattern, there exist vertices $x \neq y$, x', y' such that $x \downarrow_1 x'$, $y \downarrow_2 y'$ for some $\downarrow_1, \downarrow_2 \in \{\downarrow, \downarrow^+\}$, and x', y' are in the same connected component of the $\Rightarrow$-subgraph of π. It necessarily follows that x and y are on the same branch; the question is: in which order? Under non-recursive DTDs this can be always determined as follows.

Suppose that the labels of x and y are the same. Since the DTD is non-recursive, we know that on each path nodes have unique labels. Hence, x and y are always matched to the same node. We unify them into one node, removing one redundant edge if $x' = y'$. Note, that any binary relation between x and y in this case means that the patten is not satisfiable.

Suppose that the labels of x and y are different. Again, since the DTD is non-recursive, the order of labels is determined on each branch: for every two different labels we can say which one comes first by examining the DTD. W. l. o. g. x has an "earlier" label. Then we can add to π a $\downarrow^+$ relation from x to y. Now vertical relations between x and z became redundant (and should be removed) or contradictory, in which case the whole pattern is unsatisfiable. Similarly, any horizontal relation between a pair of nodes other then x', y' indicates that the pattern is not satisfiable.

We repeat these operations until π becomes a forest-like pattern. The number of steps is linearly limited by the number of vertices in π.

G Proof of Theorem 7

Lemma 5. *Under non-recursive disjunction-free DTDs* $\mathsf{BC\text{-}SAT}(\Downarrow)$ *is* Σ_2^p*-hard.*

Proof. We shall reduce validity of Σ_2 quantified Boolean formulae to $\mathsf{BC\text{-}SAT}(\Downarrow)$ making sure the DTD is nested-relational. Without loss of generality we can assume that the formula has the form

$$\varphi = \exists x_1 \exists x_2 \ldots \exists x_k \, \forall x_{k+1} \forall x_{k+2} \forall x_{n+\ell} \, D_1 \vee D_2 \vee \cdots \vee D_m,$$

where each D_i is a conjunction of variables or negations of variables from the set $\{x_1, x_2, \ldots, x_{k+\ell}\}$ using each variable at most once.

We shall modify the reduction in the proof of Theorem **??**. We change the DTD in two ways. First, we replace disjunction $x_i | \, \bar{x}_i$ with ${x_i}^* \; {\bar{x}_i}^*$ (a pattern will enforce that at least one is selected). Second, to each production of the form $x_i \to \alpha$ we add the list of all these disjuncts D_j that are made false by setting

x_i to true; dually for $\bar{x}_i$. The resulting DTD D is

$$
\begin{aligned}
r &\to {x_1}^* \, {\bar{x}_1}^*, \\
x_{i-1} &\to {x_i}^* \, {\bar{x}_i}^* \, \{D_j \mid D_j \text{ contains } \bar{x}_{i-1}\} && \text{for } i = 2, 3, \ldots, k, \\
\bar{x}_{i-1} &\to {x_i}^* \, {\bar{x}_i}^* \, \{D_j \mid D_j \text{ contains } x_{i-1}\} && \text{for } i = 2, 3, \ldots, k, \\
x_{i-1} &\to x_i \, \bar{x}_i \, \{D_j \mid D_j \text{ contains } \bar{x}_{i-1}\} && \text{for } i = k+1, k+2, \ldots, k+\ell, \\
\bar{x}_{i-1} &\to x_i \, \bar{x}_i \, \{D_j \mid D_j \text{ contains } x_{i-1}\} && \text{for } i = k+1, k+2, \ldots, k+\ell, \\
x_{k+\ell} &\to b \,, \\
\bar{x}_{k+\ell} &\to b \,,
\end{aligned}
$$

where $\{D_j \mid D_j \text{ contains } \bar{x}_i\}$ is interpreted as the sequence of labels from this set, ordered by the index.

The pattern $\pi_1 = \exists u \; x_{k+1}(u)$ expresses the fact that the tree encodes at least one valuation of the existentially quantified variables of φ, and

$$\pi_2 = \exists u \; b(u) \wedge \bigwedge_{j=1}^{m} \exists v \, \exists v' \; (v \downarrow^+ u) \wedge (v \downarrow v') \wedge D_j(v')$$

states that for some valuation of variables each disjunct is false. Clearly, $\pi_1 \wedge \neg\pi_2$ is satisfiable with respect to D if and only if φ is valid. □

The *support* of a h, denoted $\mathrm{supp}(h)$, is the subtree of T obtained by keeping only the nodes that can be reached from the image of h by going up, left, and right.

Lemma 6. *Under non-recursive, disjunction-free DTDs* $\mathsf{BC\text{-}SAT}(\Downarrow, \overset{+}{\rightarrow})$ *is in* Σ_2^P *and for tree patterns it is in* NP.

Proof. We can assume that formula φ is of form $\varphi = \pi \wedge \bigwedge_{i=1}^{k} \neg\pi_i$ (we can first guess the patterns to be satisfied and the ones to be not satisfied, and combine the satisfied ones into one). Suppose that a tree $\mathcal{T} \in L(\mathcal{S})$ satisfies φ, and let $h \colon \pi \to \mathcal{T}$ be a witnessing homomorphism.

Since D is disjunction-free, for each label σ there is a unique minimal tree $D(\sigma)$ conforming to D with the root symbol changed to σ. Note that $D(\sigma)$ is at most exponential in D. (Otherwise it would be infinite; such labels can be eliminated by easy preprocessing.) The tree is obtained by replacing all subexpressions of the form α^* and α^α with ε.

Using Lemma 4 we can trim $\mathcal{T}$ by removing all nodes that are not required by the DTD and are outside of the strict (root-preserving) subtree of $\mathcal{T}$ induced by the image of h. In the resulting tree each node has polynomialy many children and for each node v that is a leaf in the support of h, the subtree rooted at v is $D(\ell^{\mathcal{T}}(v))$. This allows us to recreate $\mathcal{T}$ from support of h. Since the height of $\mathcal{T}$ is limited by $\|\mathcal{S}\|$, the support of h has polynomial size, but $\mathcal{T}$ itself can be exponentially large and we cannot recreate it in PTime. We can however retrieve full knowledge about it whenever we need it.

We claim that $\mathrm{supp}(h)$ and $h\colon \pi \to \mathrm{supp}(h)$ can be used to witness satisfiability of φ with respect to $\mathcal{S}$. Checking that the support conforms to $\mathcal{S}$ and verifying that h is a homomorphism is straightforward. It remains to see that the minimal tree extending $\mathrm{supp}(h)$ satisfies none of the patterns π_i. This can be done by universally guessing a homomorphism h' from π_i to the minimal tree extending $\mathrm{supp}(h)$ and checking that it is incorrect. Of course it is enough to guess the support of h' – a polynomial object.

Tree patterns can be evaluated in PTime. Hence, for a combination of tree patterns, the witness can be verified in PTime as well, and the whole algorithm is in NP.

H Proof of Theorem 8

H.1 NExpTime upper bound for CQs

Just like in Section B.2, the NExpTime upper bound follows immediately from the following lemma.

Lemma 7. *If a Boolean combination φ of patterns that do not use $\to$ is satisfiable with respect to a disjunction-free DTD D, there is a witnessing tree of exponential size.*

Proof. Let $\mathcal{T}$ be a tree conforming to D and satisfying φ. Without loss of generality we can assume that φ is of the form $\pi \wedge \neg\pi_1 \wedge \neg\pi_2 \wedge \cdots \wedge \neg\pi_k$ (we consider a "valuation" of patterns that satisfies φ, and we combine the "true" patterns into a single pattern). Let $h\colon \pi \to \mathcal{T}$ be a witnessing homomorphism. By Lemma 4 and a standard pumping argument for regular languages we can assume that $\mathcal{T}$ has branching bounded by $|\pi| \cdot \|D\|$.

Just like in the proof of Lemma 6, we can now remove all nodes outside $\mathrm{supp}h$ that are not not enforced by the DTD. After we have done that, for each node v in the support, either all v's children are in the support or none of them, and in the second case $\mathcal{T}.v$ (the subtree of $\mathcal{T}$ rooted at v) is equal to $D(\ell^{\mathcal{T}}(v))$. In other words, $\mathcal{T}$ is $\mathrm{supp}h$ extended at each leaf v with $D(\ell^{\mathcal{T}}(v))$. Let S be the strict (root-preserving) subtree of $\mathcal{T}$ induced by the image of h. It has at most $|\pi|$ branches. The support of h is obtained from S by including all siblings. The size of the support is proportional to the height of S (at most $|\pi|^2 \cdot \|D\|$ times larger). Since each $D(\sigma)$ is at most exponential, an exponential bound on the height of S gives an exponential bound on the size of $\mathcal{T}$.

For two nodes u, v of a tree U such that $u \downarrow^+ v$, let $U_{u,v}$ denote the context obtained by removing from $U.u$ the subtree $U.v$. Let us take $u, v \in S$ such that $u \downarrow^+ v$ and $S_{u,v}$ is has empty intersection with the image of h. It follows that $S_{u,v}$ is a single path. If we manage to show an exponential bound on the length of $S_{u,v}$, we are done.

Let $M = 1 + \max_i |\pi_i|$. Suppose that there are $u_1 \downarrow^{M+1} v_1 \downarrow^+ u_2 \downarrow^{M+1} v_2$ in $S_{u,v}$ such that $(\mathrm{supp}h)_{u_1,v_1} = (\mathrm{supp}h)_{u_2,v_2}$. Since the subtrees of $\mathcal{T}$ rooted in the leaves of $\mathrm{supp}h$ are determined by the leaves' labels, we conclude that

$\mathcal{T}_{u_1,v_1} = \mathcal{T}_{u_2,v_2}$. Let $\mathcal{T}'$ be obtained from $\mathcal{T}$ by removing the context $\mathcal{T}_{v_1,v_2}$. Obviously, $\mathcal{T}'$ still satisfies π. Let us see that it does not satisfy any of π_i's. Towards contradiction, assume there is a homomorphism $h' : \pi_i \to \mathcal{T}'$. Since the path in $\mathcal{T}'$ from u_1 to v_2 (excluding v_2) has length M there is a node w in this path that is not in the image of h'. Let us split the veritces of π_i into the ones mapped to $\mathcal{T}'.w$, denote them by *Down*, and the remaining ones, *Up*. The only relation possible between vertices coming from two different parts is $x \downarrow^+ y$ for some $x \in$ *Up* and $y \in$ *Down*. But then we can define a homomorphism $h'' \colon \pi \to \mathcal{T}$, by leaving the image of *Up* where it is, and moving the image of *Down* to $\mathcal{T}_{u_2,v_2}$. Clearly, this does not break the $\xrightarrow{+}$ relations between nodes from *Up* and *Down*, and relations within *Up* and within *Down* are not influenced at all.

The number of different values that $(\mathrm{supp}h)_{u_1,v_1}$ can take is roughly $|\Gamma|^{M\cdot|\pi|\cdot\|D\|}$ ($M \cdot |\pi| \cdot \|D\|$ is the length of S_{u_1,v_1} times branching). By the pigeon-hole principle it follows that we can bound the length of $S_{u,v}$ by $M \cdot |\Gamma|^{M\cdot|\pi|\cdot\|D\|}$. □

H.2 PSpace upper bound for tree patterns

The argument uses notation and objects defined in the proof of Lemma 7.

Being a fragment of XPath, tree patterns can be translated to alternating top-down tree automata in PTime [11]. Hence, so can each Boolean combination φ. Let $\mathcal{A}_\varphi$ be this automaton. Observe that for a disjunction-free DTD D over Γ, a label $\sigma \in \Gamma$, and a state q, one can decide in PSpace if $D(\sigma)$ is accepted from q.

The algorithm testing if φ is satisfiable with respect to D works as follows. First it guesses a homomorphic image of π (without paths realising $\downarrow^+$ or sequences of children realising $\xrightarrow{+}$). This is a polynomial object. We can also guess the siblings of the nodes in the image (still polynomial, by the argument in the proof of Lemma 7). Now we will be guessing the support of the homomorphism, level by level, and at the same time we will be guessing an accepting run of $\mathcal{A}_\varphi$ on the minimal tree subsuming the support and conforming to the DTD D. Recall that this tree is obtained from the support by replacing each leaf labelled with σ by $D(\sigma)$. Hence, it is enough to guess the run on the support, and in each leaf v of the support test if $\mathcal{A}_\varphi$ accepts $D(\ell^{\mathcal{T}}(v))$ from each state assigned to v. At each moment we only keep in memory a single level of the support, which means only polynomially many nodes: recall that the strict subtree induced by the image of π has polynomially many branches, and in the support we only extend this tree with siblings, again polynomially many for each node.

The algorithm counts the number of levels guessed so far. If the bound from Lemma 7 is reached before the automaton accepts, the algorithm aborts.

H.3 Lower bound

Lemma 8. *Under disjunction-free DTDs* $\mathsf{BC\text{-}SAT}(\downarrow, \_)$ *for tree patterns is* PSpace*-hard, even for a fixed DTD.*

Proof. We modify the reduction from the linearised tiling problem used in Theorem 4. Let an instance of the problem be $T = \{t_1, t_2, \ldots, t_k\}, H, V, n$. Our alphabet is T and the DTD has productions $t_i \to t_1^* t_2^* \ldots t_k^*$ for all i and the root symbol is t_1. Consistency of the encoding is enforced by the following combination of patterns:

$$\exists x\, t_k(x) \wedge \bigwedge_{(t_i,t_j)\notin H} \neg\exists x\, \exists y\, (x \downarrow y) \wedge t_i(x) \wedge t_j(y) \wedge \bigwedge_{(t_i,t_j)\notin V} \neg\exists x\, \exists y\, (x \downarrow^n y) \wedge t_i(x) \wedge t_j(y)\,.$$

The combination simply enforces that that all branches of the tree yield consistent sequences of tiles, and that some branch contains t_k. Cutting this branch at the first occurrence of t_k we obtain the solution to the linearised tiling problem. One can modify the reduction described here to use a fixed DTD $\sharp, 0, 1 \to \sharp^* 0^* 1^*$ with the root symbol $\sharp$, by encoding the tiles as 0-1 sequences of length $\lceil \log |T| \rceil$ separated by $\sharp$, and enforcing with patterns that the separators cut each branch into segments of length $\lceil \log |T| \rceil$ (except maybe the segment after the last $\sharp$, which might be shorter) and that segments unused in the encoding of tiles do not occur. □

The second reduction from Theorem 4 can be altered in a similar way to obtain PSPACE-hardness of $\mathsf{BC\text{-}SAT}(\downarrow, \downarrow^+)$ under disjunction-free DTDs.