XPath containment in the presence of disjunction, DTDs, and variables

Frank Neven and Thomas Schwentick

[1] University of Limburg
frank.neven@luc.ac.be
[2] Philipps-Universität Marburg
tick@mathematik.uni-marburg.de

Abstract. XPath is a simple language for navigating an XML tree and returning a set of answer nodes. The focus in this paper is on the complexity of the containment problem for various fragments of XPath. In addition to the basic operations (child, descendant, filter, and wildcard), we consider disjunction, DTDs and variables. W.r.t. variables we study two semantics: (1) the value of variables is given by an outer context; (2) the value of variables is defined existentially. We establish an almost complete classification of the complexity of the containment problem w.r.t. these fragments.

1 Introduction

XPath is a simple language for navigating an XML document and selecting a set of element nodes [6]. Actually, XPath is the main XML selection language. Indeed, XPath expressions are used, for instance, as basic patterns in several XML query languages like XQuery [3] and XSLT [2, 7]; they are used in XML Schema to define keys [8], and in XLink [10] and XPointer [9] to reference elements in external documents. In every such context an instance of the containment problem is present: optimizing XPath expressions can be accomplished by an algorithm for containment, and XSLT rule selection and inference of keys based on XPath expressions again reduces to containment. In this paper we focus on the complexity of the containment problem of various fragments of XPath. In particular, we consider disjunction, DTDs and variables. The complexity of XPath evaluation has recently been studied in [12, 13].

The XPath containment problem already attracted quite some attention [1, 11, 16, 17, 25, 23]. We next discuss the known results together with our own contributions. The initial XPath fragment consists of the following operators: /, //, [ ], $*$, | which denote child, descendant, filter, wildcard, and disjunction, respectively. We indicate the fragment we use by listing the allowed operators. For instance, XP(/,//) denotes the XPath fragment where only child and descendant are allowed.

Among other results, Miklau and Suciu obtained that containment of XP(/,//, [ ],$*$) is CONP-complete [16]. For this result it does not matter whether one considers XML documents over a finite or an infinite alphabet. We show that adding disjunction to this fragment does not make the containment problem harder when the input alphabet is infinite. The proof is an extension of their canonical model technique. However, when XML documents are restricted to a finite alphabet the containment problem turns PSPACE-complete. The reason for this complexity jump is that when the alphabet is finite, disjunction allows to express negation (like is the case with regular expressions), and negation allows for a reduction from CORRIDOR TILING [5]. The upper bound is obtained by a reduction to the containment problem of alternating tree automata on bounded trees.

Deutsch and Tannen consider XPath containment in the presence of DTDs and Simple XPath Integrity Constraints (SXICs) [11]. They obtain that this problem is undecidable in general and in the presence of bounded SXICs and DTDs. When only DTDs are present they have a PSPACE lower bound and leave the exact complexity as an open question. We show that containment testing for XP(DTD,/,//,[ ],$*$,|) is in EXPTIME and obtain that containment of XP(DTD,/,//,|) and XP(DTD,/,//,[ ],$*$) is hard for EXPTIME. The upper bound is obtained by a reduction to containment of unranked tree automata [21]. The presence of the DTD allows for a reduction from TWO-PLAYER CORRIDOR TILING [5]. We do not know much about the complexity of more restrictive fragments in the presence of DTDs. In fact, we can only prove that containment of XP(DTD,/,[ ]) is CONP-complete and containment of XP(DTD,//,[ ]) is CONP-hard. It is not clear whether or how the upper bound proof can be extended to include, for instance, the descendant operator. Further, we show that in the presence of very simple DTDs and node-set inequality the containment problem is undecidable. The DTD can be eliminated when a modest form of negation is allowed: negation that can express that a node cannot have a certain label.

The XPath recommendation allows variables to be used in XPath expressions on which equality tests can be performed. For instance, $//a[\$x = @b][\$y \neq @c]$ selects all a-descendants whose b-attribute equals the value of variable $\$x$ and whose c-attribute differs from the value of variable $\$y$. However, under the XPath semantics the value of all variables should be specified by the outer context (e.g., in the XSLT template in which the pattern is issued). So the semantics of a pattern is defined w.r.t. a variable mapping. We show that the complexity of containment is PSPACE-complete under this semantics. For the lower bound, it suffices to observe that with variables a finite alphabet can be simulated. We obtain the upper bound by reducing the containment problem to the containment of several patterns without variables.

In addition to the XPath semantics, Deutsch and Tannen [11] considered an existential semantics for variables: a pattern matches a document if there is an assignment for the variables such that the pattern matches w.r.t. this assignment. W.r.t. the existential semantics they showed that containment of XP(/,//,[ ],$*$,vars) and XP(/,//,[ ],|,vars) is Π_2^P-hard, and that containment of XP(/,//, [ ],|,vars) under fixed bounded SXICs is in Π_2^P. We extend their result by showing that containment of XP(/,//,[ ],|,vars,$\neq$), that is, inequality tests on variables and attribute values are allowed, remains in Π_2^P. Interestingly, when $*$ is added, the problem turns undecidable.

In a recent paper, Wood obtained that containment of XP(/,//,[ ],$*$) in the presence of DTDs is decidable [23]. He also studies conditions for which containment under DTDs is in PTIME. Benedikt, Fan, and Kuper study the expressive power and closure properties of fragments of XPath [1]. They also consider sound and complete axiom systems and normal forms for some of these fragments.

This paper is organized as follows. In Section 2, we define DTDs and the basic XPath fragments. In Section 3, 4, and 5 we consider disjunction, DTDs, and variables, respectively. We conclude in Section 6.

Due to space limitations we only provide sketches of proofs.

2 Preliminaries

In the present section, we define trees, DTDs, and the core of XPath.

For the rest of this paper we fix a recursively enumerable infinite alphabet Σ and a recursively enumerable infinite set of data values $\mathbf{D}$. A is always a finite set of attributes. An XML document is faithfully modelled by a finite Σ-tree where the attributes of the nodes have $\mathbf{D}$-values.

Formally, a *tree domain* τ *over* $\mathbb{N}$ is a subset of $\mathbb{N}^*$, such that if $v \cdot i \in \tau$, where $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, then $v \in \tau$. Here, $\mathbb{N}$ denotes the set of natural numbers without zero. If $i > 1$ then also $v \cdot (i-1) \in \tau$. The empty sequence, denoted by ε, represents the root. We call the elements of τ *vertices*. A vertex w is a *child* of a vertex v (and v the *parent* of w) if $vi = w$, for some i.

Definition 1. A (Σ, A)-*tree* is a triple $t = (\mathrm{dom}(t), \mathrm{lab}_t, \lambda_t)$, where $\mathrm{dom}(t)$ is a tree domain over $\mathbb{N}$, $\mathrm{lab}_t : \mathrm{dom}(t) \to \Sigma$, and, for each $a \in A$, $\lambda_t^a : \mathrm{dom}(t) \to \mathbf{D}$ are functions. Intuitively, $\mathrm{lab}_t(v)$ is the label of v, while $\lambda_t^a(v)$ is the value of v's a-attribute.

When Σ and A are clear from the context or not important, we sometimes say tree rather than (Σ, A)-tree. Of course, in real XML documents not every element has the same attributes. Further, there can be nodes with mixed content, but these can easily be modeled by using dummy intermediate nodes [2]. We only make use of attributes in Section 5.

We formalize a DTD as a context-free grammar with regular expressions on the right-hand side of rules.

Definition 2. A *DTD* is a tuple (d, S_d, Σ_d) where Σ_d is a finite subset of Σ, $S_d \in \Sigma_d$ is the start symbol, and d is a mapping from Σ_d to the set of regular expressions over Σ_d. A tree t *matches* a DTD d iff $\mathrm{lab}_t(\varepsilon) = S_d$ and for every $u \in \mathrm{dom}(t)$ with n children, $\mathrm{lab}_t(u1) \cdots \mathrm{lab}_t(un) \in L(d(\mathrm{lab}(u)))$. We denote by $L(d)$ the set of all trees that satisfy d.

Note that DTDs do not constrain the value of attributes in any way. We usually refer to a DTD by d rather than (d, S_d, Σ_d).

We next define the core fragment of XPath that we will consider in Sections 3 and 4.

Definition 3. An XP-*expression* is an expression defined by the following grammar:

$$\begin{array}{lr} p := p_1 \mid p_2 & \text{(disjunction)} \\ \quad \mid /p & \text{(root)} \\ \quad \mid //p & \text{(descendant)} \\ \quad \mid p_1/p_2 & \text{(child)} \\ \quad \mid p_1//p_2 & \text{(descendant)} \\ \quad \mid p_1[p_2] & \text{(filter)} \\ \quad \mid \sigma & \text{(element test)} \\ \quad \mid * & \text{(wildcard)} \end{array}$$

Here, $\sigma \in \Sigma$.

It remains to define the semantics of XP expressions. In brief, every XP expression p induces a mapping $[\![p]\!]_t : \mathrm{dom}(t) \mapsto 2^{\mathrm{dom}(t)}$. This mapping is defined w.r.t. a tree t. We inductively define $[\![p]\!]_t$ as follows: for all $u \in \mathrm{dom}(t)$,

– $[\![p_1 \mid p_2]\!]_t(u) := [\![p_1]\!]_t(u) \cup [\![p_2]\!]_t(u)$;
– $[\![/p]\!]_t(u) := \begin{cases} [\![p]\!]_t(\varepsilon) & \text{if } u = \varepsilon; \\ \emptyset & \text{otherwise;} \end{cases}$
– $[\![//p]\!]_t(u) := \{v \mid \exists w : v \in [\![p]\!]_t(w)\}$;
– $[\![p_1/p_2]\!]_t(u) := \{v \mid \exists w \in \mathbb{N}^*, z \in \mathbb{N} : w \in [\![p_1]\!]_t(u) \text{ and } v \in [\![p_2]\!]_t(wz)\}$;
– $[\![p_1//p_2]\!]_t(u) := \{v \mid \exists w \in \mathbb{N}^*, z \in \mathbb{N}^* \setminus \{\varepsilon\} : w \in [\![p_1]\!]_t(u) \text{ and } v \in [\![p_2]\!]_t(wz)\}$;
– $[\![p_1[p_2]]\!]_t(u) := \{v \mid v \in [\![p_1]\!]_t(u) \text{ and } [\![p_2]\!]_t(v) \neq \emptyset\}$;
– $[\![*]\!]_t(u) := \{u\}$;
– $[\![\sigma]\!]_t(u) := \begin{cases} \{u\} & \text{if } \mathrm{lab}_t(u) = \sigma; \\ \emptyset & \text{otherwise;} \end{cases}$

Obviously, XPath expressions express unary queries. However, we can also use expressions for Boolean queries by testing whether $[\![p]\!]_t(\varepsilon) \neq \emptyset$. We denote the latter also by $t \models p$ or say that t matches p. Like in [16], we can reduce containment testing of unary queries to containment of Boolean queries by introducing new labels.

Definition 4. We say that p is *contained* in q, denoted $p \subseteq q$, if for all t, $t \models p$ implies $t \models q$. For a DTD d, p is *contained* in q *w.r.t. d*, denoted $p \subseteq_d q$, if for all $t \in L(d)$, $t \models p$ implies $t \models q$.

As already mentioned in the introduction, we denote the fragment of XP under consideration by listing the allowed operators. Element test is always allowed.

In Definition 3, we allow absolute expressions, as opposed to relative ones, to appear within filter expressions. For instance, in the expression $//a[/c//b]$, $/c//b$ looks for a b starting at the children of the c-labeled root rather than starting from a. When no disjunction is present we can always move such expressions to the top and introduce a new root symbol. For instance, suppose we are given $//a[/c//b]$ and $//a$ then we take the expressions $/\#[//a][/c//b]$ and $/\#//a$. In the proofs for fragments where disjunction is present, we always go through their

DNF, that is, for every expression q we take the equivalent expression $q_1 \mid \cdots \mid q_n$ where no q_i contains disjunction. In each of the disjuncts we can move absolute expressions to the root. For this reason, we do not deal with absolute filter expression in the upper bound proofs of this paper. But it should be pointed out that we make significant use of absolute path expressions in the proof of undecidability in the presence of DTDs and node-set inequality (Theorem 10).

In some proofs, we view patterns p from XP(/,//,[ ],$*$) as *tree patterns* as described by Miklau and Suciu [16]. From this point of view a tree t matches a pattern p iff there is a homomorphism from (the tree pattern associated with) p to t, i.e., a mapping which respects labels, child and descendant, (and does not care about $*$). For example, the pattern $a/b//c[d][*/e]$ corresponds to the tree pattern in Figure 1. Single edge and double edge correspond to child and descendant relation. All nodes of the input tree that can be mapped onto the x-labelled node are selected.

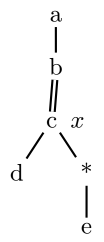


Fig. 1. The tree pattern corresponding to $a/b//c[d][*/e]$

3 The base cases: /, //, [ ], |, and $*$

Milkau and Suciu showed that XP(/,//,[ ],$*$) is CONP-complete [16]. In this section, we add | and show that containment remains in CONP. Containment is even hard for CONP when | is considered in isolation with / or //. In fact, during the write up of this paper, we noted that Miklau and Suciu already mention these results in their discussion section but do not provide proofs. For this reason we decided to include the proofs in this paper. Moreover, we want to stress that the reason there is an CONP upper bound is because the alphabet Σ is infinite. Indeed, when we restrict to a finite alphabet, then XP(/,//,|) becomes hard for PSPACE.

Theorem 5. *1. Containment of* XP(/, //, [ ], $*$, |) *expressions is in* CONP*;*
2. Containment of XP(/, |)*-expressions is* CONP*-hard.*
3. Containment of XP(//, |)*-expressions is* CONP*-hard.*

Proof. (sketch) The hardness proofs are rather straightforward and are omitted. We proceed with the proof of (1). We develop a criterion which allows to check in NP whether, for given patterns p and q, $p \not\subseteq q$. Let p and q be fixed and let $p_1|...|p_l$ and $q_1|\dots|q_{l'}$ be the disjunctive normal forms (DNFs) of p and q, respectively. Hence, each p_i and q_j is a pattern from $\mathrm{XP}(/, //, [\,], *)$. Let n and m denote the maximum number of nodes in a pattern p_i and q_j, respectively. Let $T(p, q)$ be the set of trees with at most $2n(m + 2)$ nodes that are labelled with labels that occur in p and with the new label # not occurring in p nor in q. It is possible to prove the following claim:

Claim. $p \not\subseteq q \Leftrightarrow$ there is a $t \in T(p, q)$ such that $t \models p$ but $t \not\models q$.

It remains to show how the above criterion can be used for an NP-algorithm that checks whether $p \not\subseteq q$. The algorithm simply guesses a pattern p_i from the DNF of p (by nondeterministically choosing one alternative for each $|$ in p) and a $t \in T(n, m)$. Then it checks that $t \models p_i$ and $t \not\models q$. The latter can be done in polynomial time as shown in [13]. □

When the alphabet is finite, the containment problem becomes PSPACE-complete. Actually, the finite alphabet allows us to express that an element name in the XML document does not occur in a certain set. This is the only property we need. Therefore, if we extend the formalism with an operator $*_{\not\in S}$ for a finite set S, expressing that any symbol but one from S is allowed, then containment would also be hard for PSPACE.

Theorem 6. *When Σ is finite,*

1. *containment of* $\mathrm{XP}(/, //, [\,], *, |)$*-expressions is in* PSPACE*; and*
2. *containment of* $\mathrm{XP}(/, //, |)$*-expressions is* PSPACE*-hard.*

Proof. (sketch) *Upper bound.* Let k be a natural number. We say that a tree is *k-bounded* if it has at most k non-unary nodes (that is, nodes with more than one child) and every node has rank at most k (that is, at most k children). Let p and q be two patterns in $\mathrm{XP}(/, //, [\,], *, |)$. Let $f(p)$ denote the number of filter expressions in p. It is easy to see that p is contained in q iff p is contained in q on the class of $f(p)$-bounded trees. Indeed, suppose there is a t such that $t \models p$ and $t \not\models q$. Let the DNF of p and q be $p_1|\cdots|p_n$ and $q_1|\cdots|q_m$. Note that each disjunct has at most as many of filter expressions as the original XP expression. Let i be such that $t \models p_i$. Hence, there is a $(/, //, [\,], *)$-homomorphism h from p_i to t (as defined in the proof of Theorem 5). Let V be the set of nodes on the image of h. Let s be the tree obtained from t by deleting all nodes that are not in V and are not ancestors of nodes in V. Then, clearly, s is $f(p)$-bounded, $s \models p_i$ and $s \not\models q_j$ for all j (otherwise, $t \models q_j$).

By ATA we denote the class of ranked alternating top-down automata [22]. We say that an automaton is *bounded* iff there is a k such that whenever a tree is accepted by the automaton it is k-bounded.

Given two patterns p and q in $\mathrm{XP}(/, //, [\,], *, |)$, let $n = f(p)$. The remainder of the proof consists of two steps: (1) we show that p can be transformed into an n-bounded automaton M_p such that p and M_p are equivalent on n-bounded trees; (2) we show that containment of n-bounded automata is in PSPACE. Details are omitted.

Lower bound. We make use of a reduction from CORRIDOR TILING which is known to be hard for PSPACE [5]. Let $T = (D, H, V, \bar{b}, \bar{t}, n)$ be a tiling system. Here, $D = \{a_1, \ldots, a_k\}$ is a finite set of tiles; $H, V \subseteq D \times D$ are horizontal and vertical constraints, respectively; $\bar{b} = (b_1, \ldots, b_n), \bar{t} = (t_1, \ldots, t_n)$ are n-tuples of tiles; n is a natural number in unary notation. A player places tiles on an $n \times \mathbb{N}$ board (n columns, unlimited rows). On this board the bottom row is tiled with $\bar{b}$. Every placed tile should satisfy the horizontal and vertical constraints. The top row should be tiled with $\bar{t}$. The problem is to determine whether a tiling exists.

We use a string representation of the $n\times\mathbb{N}$ board where every row is delimited by $\#$ and the last symbol is \$. The XP pattern q selects all strings that do not encode a tiling. As Σ we take $D \cup \{\#, \$\}$. By D^i we denote the pattern $(a_1|\cdots|a_k)$, repeated i times which describes i successive symbols of D. The pattern p is $//\$$ assuring that the string contains the symbol \$. The pattern q is the disjunction of the following patterns:

- a row has the wrong format: $\bigcup_{i=0}^{n-1} //\#D^i\#\cup\bigcup_{i=0}^{n-1} /D^i/\#\cup\bigcup_{i=0}^{n-1} //\#D^i/\$\cup //D^{n+1}$;
- \$ occurs inside the string: $//\$/(D \cup \{\$\} \cup \{\#\})$;
- the string does not begin with b: $\bigcup_{i=1}^{n} /b_1/\cdots/b_{i-1}/(\bigcup_{a_j\neq b_i} a_j)$;
- the string does not end with t: $\bigcup_{i=1}^{n}(\bigcup_{a_j\neq t_i} a_j)/t_{i+1}/\cdots/t_n/\$$
- some horizontal constraint is violated:$\bigcup_{(d_1,d_2)\notin H} //d_1/(D \cup \{\#\})^{n+1}/d_2$;
- some vertical constraint is violated: $\bigcup_{(d_1,d_2)\notin V} //d_1/d_2$.

Now, T has a solution iff $p \not\subseteq q$. Clearly, if T has a solution then we can take the string encoding of the tiling as a counter example for the containment of p and q. Conversely, if $p \not\subseteq q$ then there is a, not necessarily unary, tree t with one branch s ending on a \$ such that $s \models p$ and $s \not\models q$. So, this branch encodes a solution for T. $\square$

4 Containment in the presence of DTDs

Deutsch and Tannen obtained a PSPACE lower bound on the complexity of containment in the presence of DTDs. In the general case, we prove that the complexity is EXPTIME-complete. We also have a modest NP-completeness result on the fragment using only / and [ ]. We do not know how to extend the upper bound proof to include // or $*$. Finally, we show that adding nodeset comparisons w.r.t. $=$ and $<$ leads to undecidability. In fact, when a modest form of negation is introduced expressing that certain labels cannot appear as a child of a node, the DTD can be dispensed with. The results of this section are summarized in Table 1.

We start with a fragment in P. The lower bounds in Theorem 5 and in Theorem 8 show that this is the largest fragment whose complexity of containment w.r.t. DTDs is in P.

DTD	/	//	[ ]	|	$*$	complexity
+	+	+			+	in P
+	+		+			CONP-complete
+		+	+			CONP-hard
+	+	+	+	+	+	EXPTIME-complete
+	+	+		+		EXPTIME-complete
+	+	+	+		+	EXPTIME-complete
+	+	+	+		+	undecidable with nodeset comparisons

Table 1. The complexity of containment in the presence of DTDs.

Theorem 7. *Containment of* XP$(DTD, /, //, *)$*-expressions is in* P.

Proof. (sketch) Let d be a DTD and p, q be patterns of XP$(DTD, /, //, *)$. Note that although p and q only can match paths it is not sufficient to reason about single paths in trees. It might be the case that whenever a tree t has a path which matches p, the DTD forces the tree to have a different path which matches q.

We construct a non-deterministic top-down tree automaton A that accepts a tree t if and only if (1) t conforms to d, (2) $t \models p$, and (3) $t \not\models q$. Once A is constructed it only remains to check that A does not accept any tree to conclude that $p \subseteq_d q$. This can be tested in polynomial time in $|A|$. Further, the construction time and the size of $|A|$ are polynomial in the overall size of d, p, and q. Hence, the algorithm is indeed polynomial. Further details are omitted. □

Next, we consider a fragment in CONP. It is open whether XP$(DTD, /, [\,])$ is a maximal fragment whose complexity of containment w.r.t. DTDs is in CONP.

Theorem 8. *1. Containment testing for* XP$(DTD, /, [\,])$ *is in* CONP.
2. Containment testing for XP$(DTD, /, [\,])$ *is* CONP*-hard.*
3. Containment testing for XP$(DTD, //, [\,])$ *is* CONP*-hard.*

Proof. (sketch) We only prove (1). We present a nondeterministic algorithm `CheckPnotq` which checks, given a DTD d, a non-terminal s of d, an XP$(/, [\,])$-pattern q and a set $P = \{p_1, \ldots, p_n\}$ of XP$(/, [\,])$-patterns, whether there is a tree t with root symbol s which conforms to d, matches the patterns in P but does not match q. Clearly, invoking this algorithm with d, q, $P = \{p\}$ and $s = \text{root}(d)$ checks whether $p \not\subseteq_d q$.

A complication arises from the fact that the smallest counter example tree t might be of exponential size due to the constraints from d. Hence, we can not simply guess such a counter example.

We make use of two algorithms with slightly simpler tasks. Algorithm `CheckP` checks on input d, s, P whether there is a tree t with root s conforming to d which contains all the patterns from P. Algorithm `Checknotq` checks on input d, q whether there is a tree conforming to d with a root labelled by the root symbol of q which does *not* match q. The construction of the two algorithms is omitted. Both work non-deterministically in polynomial time. In both algorithms and below, we make use of the following notation. For a DTD d let $U(d)$ be the set of non-terminals a of d that are useful in the sense that there is a tree t with root label a that conforms to d. $U(d)$ can be computed in polynomial (even linear) time from d by using standard methods.

Algorithm `CheckPnotq` proceeds as follows. Let $d, s, P = \{p_1, \dots, p_n\}, q$ be an input.

- First, it checks whether all patterns in P have the root symbol s. If this is not the case it returns FALSE.
- Next, it checks whether q has the root symbol s. If this is not the case it calls `CheckP` with parameters $d, s, P = \{p_1, \dots, p_n\}$ and returns TRUE iff `CheckP` does.
- It guesses a string u of length at most $(|d| + 1)(l + 1)$ and verifies that u conforms to the regular expression of s in d and that all non-terminals in u are in the set $U(d)$ of useful symbols.
- It guesses a child of the root of q. Let q' be the pattern rooted at this child.
- For each $i \in \{1, \dots, n\}$, it guesses a mapping f_i from the children $v_1, \dots, v_m$ of the root of p_i to the positions of u.
- For each position j of u, which is in the image of at least one of the mappings f_i, it does the following
  - Let P' be the vertices that are mapped to j.
  - If u_j is the symbol of the root of q' then call `CheckPnotq` recursively with parameters d, u_j, P', q'.
  - Otherwise call `CheckP` with parameters d, u_j, P'.
  - Let s' be the label at the root of q'. If s' does not occur in P' but in u it calls `Checknotq` with parameters d, q'.
- It returns TRUE iff all the subcomputations return TRUE.

Clearly, this algorithm checks nondeterministically in polynomial time whether there is a counter example conforming to d which matches all patterns in P but not q. The reasoning for the correctness is similar to the case of `CheckP`.

We note that in the above theorem, (2) and (3) were also obtained by Wood [24].

When disjunction, or filter and wildcard come in to play, the complexity raises from P and CONP to EXPTIME.

Theorem 9. *1. Containment testing for* $\mathrm{XP}(DTD, /, //, [\,], *, |)$ *is in* EXPTIME.
2. Containment testing for $\mathrm{XP}(DTD, /, //, |)$ *is hard for* EXPTIME.
3. Containment testing for $\mathrm{XP}(DTD, /, //, [\,], *)$ *is hard for* EXPTIME.

Proof. (sketch) The upper bound is shown by a translation to emptiness of an unranked tree automaton whose size is exponential in the input. See [18–20] for an overview of unranked tree automata.

First of all, we indicate that, for each $\mathrm{XP}(/, //, [\,], *)$-pattern p, one can construct in exponential time an exponential size deterministic tree automaton A_p such that A_p accepts a tree if and only if it matches p. Let t_p be the tree pattern for the expression p. The states of A_p are pairs (S_1, S_2) of sets of nodes of t_p. The intended meaning is as follows. If $v \in S_1$ then the subtree of the input tree rooted at the current node matches the subpattern of t_p rooted at v. If $v \in S_2$ then there is a node below the current node of the input tree which matches the subpattern rooted at v. S_2 is used to handle descendant edges. Note that there are two sources for the exponential size of A_p. First of all, there is possibly an exponential number of states. Second, the regular expressions (or finite automata) that describe the transitions of A_p from the children of a node to the node itself might be of exponential size. E.g., if a vertex in t_p has k children then the associated regular expression might be of size about $k!$.

Let now d be a DTD and let p and q be $\mathrm{XP}(/, //, [\,], *, |)$-patterns. Let $p = p_1 \mid \cdots \mid p_m$ and $q = q_1 \mid \cdots \mid q_l$ be disjunctive normal forms. Note that m and l might be exponential in $|p|$ and $|q|$, respectively (but not more). Let A_d be a nondeterministic top-down automaton checking conformance with d. Let A be the product automaton of A_d, the automata A_{p_i} and the automata A_{q_j} such that A accepts, if A_d accepts, at least one of the A_{p_i} accepts and all the A_{q_j} reject. The latter is possible as the A_{q_j} are all deterministic. Clearly, A is of exponential size.

Now $p \subseteq q$ if and only if A does not accept any tree. This concludes the proof of (1)

The proofs of the lower bound make use of a reduction from TWO-PLAYER CORRIDOR TILING [5]. The latter problem is the extension of CORRIDOR TILING, used in the proof of Theorem 6 (1), to two players (`I` and `II`). Again the game is played on an $n \times \mathbb{N}$ board. Each player places tiles in turn. While player `I` tries to construct a corridor tiling, player `II` tries to prevent it. It is known that it is EXPTIME-complete to determine whether player `I` has a winning strategy no matter how player `II` plays. Given such a tiling system, we construct a DTD d and two patterns p and q such that $p \not\subseteq_d q$ iff player `I` has a winning strategy. Intuitively, the DTD defines the set of all strategy trees, p selects every tree, and q checks whether a possible strategy tree contains an error. Details are omitted. $\square$

The core fragment XP of XPath, defined in the previous section, leaves out many features of XPath: node-set equality, location paths, the many functions in the function library, among others. When operators from the function library like arithmetical operators or string concatenation are allowed, Moerkotte already showed that containment is undecidable [17].

It is an interesting open question to pinpoint exactly the minimal XPath fragments that have an undecidable containment problem. In the present section we show that containment already becomes undecidable in the presence of very simple DTDs when we allow node-set equality and inequality with the additional $<$ operator. In addition, we show that we can get rid of the simple DTDs when a certain kind of negation, already present in full XPath, over child labels is allowed.

We define $\mathrm{XP}^{\mathrm{ns}}$ as XP extended with the following rules: if p, q_1 and q_2 are $\mathrm{XP}^{\mathrm{ns}}$ expressions then $p[q_1 = q_2]$, $p[q_1 < q_2]$, $p[not(q_1 = q_2)]$, and $p[not(q_1 < q_2)]$

are XP$^{\mathrm{ns}}$ expressions. To define their semantics, we introduce some notation. For a tree t and a node $v \in \mathrm{dom}(t)$, define $\mathrm{yield}(t_v)$ as the string obtained by concatenating from left to right the labels of the leave nodes that are descendants of v. Note that this definition is in conformance with the definition of the string-value of an element node in the XPath data model [6]. For instance, if t is the tree $a(b, a(c), d)$ then $\mathrm{yield}(t_\varepsilon)$ is bcd. We assume an ordering $<$ on Σ. The semantics is defined as follows, for $* \in \{=, <\}$,

$$\begin{aligned}\llbracket p[q_1 * q_2] \rrbracket_t(u) := \{v \mid\ & v \in \llbracket p \rrbracket_t(u) \text{ and} \\ & \exists v_1 \in \llbracket q_1 \rrbracket_t(v), \exists v_2 \in \llbracket q_2 \rrbracket_t(v) \text{ such that } \mathrm{yield}(t_{v_1}) \ * \ \mathrm{yield}(t_{v_2})\};\end{aligned}$$

$$\begin{aligned}\llbracket p[not(q_1 * q_2)] \rrbracket_t(u) := \{v \mid\ & v \in \llbracket p \rrbracket_t(u) \text{ and} \\ & \forall v_1 \in \llbracket q_1 \rrbracket_t(v), \forall v_2 \in \llbracket q_2 \rrbracket_t(v) \text{ such that } \neg(\mathrm{yield}(t_{v_1}) \ * \ \mathrm{yield}(t_{v_2}))\}.\end{aligned}$$

A *simple* DTD is a DTD where every rule is of the form $a \to b$ or $a \to c(b_1 + \cdots + b_n)$. The next proof is an involved reduction from PCP, we only provide a rough sketch.

Theorem 10. *Containment of XP^{ns} expressions w.r.t. simple DTDs is undecidable.*

Proof. (sketch) We use a reduction from Post's Correspondence Problem (PCP) which is well-known to be undecidable [14]. An *instance* of PCP is a sequence of pairs $(x_1, y_1), \ldots, (x_n, y_n)$, where $x_i, y_i \in \{a, b\}^*$ for $i = 1, \ldots, n$. This instance has a *solution* if there exist $m \in \mathbb{N}$ and $\alpha_1, \ldots, \alpha_m \in \{1, \ldots, n\}$ such that $x_{\alpha_1} \cdots x_{\alpha_m} = y_{\alpha_1} \cdots y_{\alpha_m}$.

We construct a DTD d, and two XPath expressions p_1 and p_2 such that $p_1 \subseteq_d p_2$ iff the PCP instance has a solution.

We consider XML trees that are almost unary trees or, equivalently, simply strings. They are of the form $u\$v$, where \$ is a delimiter and u, v are strings representing a candidate solution $(x_{\alpha_1}, \ldots, x_{\alpha_m}; y_{\beta_1}, \ldots, y_{\beta_m})$ for the PCP instance in a suitable way. To check whether such a candidate is indeed a solution, we roughly have to check whether

1. $\alpha_i = \beta_i$ for each i, that is, corresponding pairs are taken; and
2. both strings are the same, that is, corresponding positions in $x_{\alpha_1} \cdots x_{\alpha_m}$ and $y_{\alpha_1} \cdots y_{\alpha_m}$ carry the same symbol.

To check the correspondences mentioned in (1) and (2), we make use of a double indexing system based on string-values of children nodes of the nodes of u and v (and therefore the trees are not literally unary). Details are omitted. $\square$

To get rid of simple DTDs, we allow a modest form of negation. Let XP$^{ns}(not)$ be the extension of XP^{ns} extended with the rules $p[not(a)]$ and $p[not(c) \mid not(b_1 \mid \cdots \mid b_n)]$ where $a, c, b_1, \ldots, b_n \in \Sigma$, expressing that there is a node selected by p that does not have an a child, and does not have a c-labeled child or does not have one with a label coming from $b_1, \ldots, b_n$, respectively.

Lemma 11. *Let d be a simple DTD, let p and q be two XP^{ns} expressions, then there is an XP(/,//,[ ],not) expression q_d such that $p \subseteq_d q$ iff $p \subseteq q \mid q_d$.*

Proof. For every rule $a \rightarrow b$, let q_a be the expression $//a[not(b)]$. For every rule $a \rightarrow c(b_1 + \cdots + b_n)$, let q_a be the expression $//a[not(c) \mid not(b_1 \mid \cdots \mid b_n)]$. Define q_d as the union of all such expressions. □

Corollary 12. *Containment of $XP^{ns}(not)$ expressions is undecidable.*

5 Containment in the presence of data values

In the present section, we add attribute comparisons to our XPath fragment. Formally, we add the following rules to Definition 3:

$$\begin{array}{l} \mid p[\$x = @a] \text{ (variables)} \\ \mid p[\$x \neq @a] \text{ (inequalities)} \end{array}$$

Here, a is an attribute, and $\$x, \$y$ are variables. The presence of the former rule is indicated by 'vars' the presence of the latter by '$\neq$'.

We consider two semantics. The first one corresponds to the XPath semantics and we refer to it in that way. The variable binding is defined in an outer context, not by matching the pattern with the tree. In particular, the value of a pattern is defined w.r.t. a variable assignment $\rho : X \rightarrow \mathbf{D}$ where X is the set of all variables. Formally,

$$[\![p[\$x = @a]]\!]^{\rho}_t(u) := \{v \mid v \in [\![p]\!]^{\rho}_t(u) \text{ and } \rho(x) = \lambda^a_t(v)\}; \text{ and}$$

$$[\![p[\$x \neq @a]]\!]^{\rho}_t(u) := \{v \mid v \in [\![p]\!]^{\rho}_t(u) \text{ and } \rho(x) \neq \lambda^a_t(v)\}.$$

So, p is contained in q w.r.t. ρ, denoted $p \subseteq_\rho q$, iff $[\![p]\!]^{\rho}_t(\varepsilon) \neq \emptyset$ implies $[\![q]\!]^{\rho}_t(\varepsilon) \neq \emptyset$.

Theorem 13. *(a) Containment testing for* XP$(/, //, [\,], |, *, vars, \neq)$ *under the XPath semantics of variables is in* PSPACE.
(b) Containment testing for XP$(/, //, |, *, vars, \neq)$ *under the XPath semantics of variables is* PSPACE*-hard.*

Proof. (sketch)

(a) For the upper bound, we basically show that the problem can be reduced to the case without variables. Let p and q be two expressions with variables $\{x_1, \ldots, x_k\}$. Let us first consider only variable assignments which assign a different value to each variable. Hence, for an attribute a of a node u of a tree t it is only relevant whether the value of a is equal to (exactly) one of the x_i or whether it is different from all of them. Let t be a tree such that $t \models p$ and $t \not\models q$ under assignment ρ. We add, for each attribute a of a node v, a new child of v labelled by a which itself has a child which is labeled by one of $x_1, \ldots, x_k$ or with `none`. So, if the value of a is $\rho(x_i)$ in t, for some i, then it is labelled x_i, and `none` if this is not the case, for all i. Let us call the

resulting tree t'. In p we replace each $\$x_i = @a$ by a/x_i and each $\$x_i \neq @a$ by $a/(x_1 \mid \cdots \mid x_{i-1} \mid x_{i+1} \mid \cdots \mid x_k \mid \texttt{none})$. We call the resulting pattern p'. Finally we construct q' from q in the same way. It is easy to see that, for each u in t: $[\![p]\!]_t^\rho(u) = [\![p']\!]_{t'}(u)$, where we identify the original nodes of t' with their counterparts in t.
We conclude that if ρ is a variable assignment with pairwise different values, then $p \subseteq_\rho q$ if and only if $p' \subseteq q'$ on all trees of the form t'. Note that these trees have partially a restriction to a finite alphabet. By using a similar proof as for Theorem 6, it can be shown that this test can be done in PSPACE.
The algorithm now cycles through all possible equality types of the variable assignment ρ. If, for a particular equality type, two variables get the same value then one of them is replaced by the other in p and in q. Hence, we get possibly fewer variables which are again pairwise different and we can apply the above algorithm. The resulting algorithm is in PSPACE.

(b) We use basically the same construction as in Theorem 6. Let $D = \{\sigma_1, \ldots, \sigma_k\}$ be the alphabet used in that construction and assume w.l.o.g. that $k = 2^l$, for some l. We use attributes $a_1, \ldots, a_l$ and one variable x to encode the symbols of D. E.g., if all a_i of a node v have the same value as x we consider it as labelled with σ_1. If the value of a_l is different from that of x but all other a_i have the value of x, for some v then we interpret this as symbol σ_2 and so on. In this way, the k symbols correspond to the k different equality types of attributes relative to x. In the expressions p and q the element tests are replaced by the wildcard symbol together with the respective attribute comparisons. □

Deutsch and Tannen considered a different semantics which does not assume an external variable binding but rather allows a choice of values for the variables that makes the expression match. More formally, $[\![p]\!]_t(u)$ is defined as the set $\{v \in [\![p]\!]_t^\rho(u) \mid \rho : X \to \mathbf{D}\}$. In particular, for Boolean patterns this means that a tree t matches a pattern p under the semantics of Deutsch and Tannen if there exists a variable assignment ρ such that t matches p relative to ρ. We will refer to this sematics as the *existential semantics*.

Deutsch and Tannen showed that containment of $\mathrm{XP}(/, //, [\,], *, |, \mathrm{vars})$-expressions under existential semantics is Π_2^P-complete [11] (Theorem 2.3 and 3.3). Further, they show that containment of $\mathrm{XP}(/, //, [\,], \mathrm{vars})$-expressions is CONP-complete.

Our main results about the existential semantics are the following. When $\neq$ is added to $\mathrm{XP}(/, //, [\,], *, |, \mathrm{vars})$, then containment is undecidable. However, when $\neq$ is added but $*$ is removed, then containment remains in Π_2^P. Hardness follows immediately as containment of conjunctive queries (CQs) with inequalities is already Π_2^P-hard and containment of CQs is reducible to XPath containment.

The results of this section are summarized in Table 2.

Theorem 14. *Containment testing for* $\mathrm{XP}(/, //, [\,], |, vars, \neq)$ *under existential semantics is in* Π_2^P*.*

Proof. (sketch) We show that

/	//	[]	$*$	|	vars	$\neq$	complexity
+	+	+			+		CONP-complete [11]
+	+				+		CONP-complete
+	+	+	+	+	+		Π_2^p-complete [11]
+		+			+	+	Π_2^p-complete
+	+	+		+	+	+	Π_2^p-complete
+	+	+	+	+	+	+	undecidable

Table 2. The complexity of containment in the presence of data values under existential semantics. Under XPath semantics the complexity for the full fragment is PSPACE.

(a) $p \not\subseteq q$ if and only if there is a tree t of polynomial size in $|p| + |q|$ such that $t \models p$ but $t \not\models q$, and
(b) Whether $t \models p$ can be tested in NP.

Hence, the algorithm *Guess a tree t of polynomial size and check that $t \models p$ but $t \not\models q$* is a Σ_2-algorithm for the complement of containment testing.

We omit the proof of (a). To show (b), we remark that whether $t \models p$ for a pattern p in XP$(/, //, [\,], |, \text{vars}, \neq)$ can be tested as follows. First, a disjunct p_i of the disjunctive normal form of p is guessed. Next, a homomorphism from p_i to t and a value assignment for the variables of p_i are guessed (with values $\leq |p_i|$) and it is checked whether all conditions hold. $\square$

A proof of the next theorem is again a reduction from PCP and is omitted due to space restrictions.

Theorem 15. *Containment testing for XP$(/,//,[\,],*,|,vars,\neq)$ under existential semantics is undecidable.*

6 Discussion

We have studied the complexity of the containment problem for a large class of XPath patterns. In particular, we considered disjunction, DTDs and variables. Unfortunately, the complexity of almost all decidable fragments lies between CONP and EXPTIME. On the other hand, the size of XPath expressions is rather small. As pointed out, Deutsch and Tannen, and Moerkotte already obtained undecidability results for XPath containment. We added two more: presence of node-set equality and modest negation or variables with the existential semantics. It would be interesting to have a precise classification of which combination of features makes the problem undecidable.

In a next step, also navigation along the other axes of XPath should be investigated.

Acknowledgment

We thank Stijn Vansummeren for comments on a previous version of this paper. We thank the anonymous referees for valuable suggestions.

References

1. M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. ICDT 2003, this volume.
2. G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.
3. D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanascu. XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery/`, 2002.
4. A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.
5. B. S. Chlebus. Domino-tiling games. *Journal of Computer and System Sciences*, 32(3):374–392, 1986.
6. J. Clark. XML Path Language (XPath). `http://www.w3.org/TR/xpath`.
7. James Clark. XSL transformations version 1.0. http://www.w3.org/TR/WD-xslt, august 1999.
8. World Wide Web Consortium. XML schema. `http://www.w3.org/XML/Schema`.
9. S. DeRose, E. Maler, and R. Daniel. XML pointer language (XPointer) version 1.0. `http://www.w3.org/TR/xptr/`, 2001.
10. S. DeRose, E. Maler, and D. Orchard. XML linking language (XLink) version 1.0. `http://www.w3.org/TR/xlink/`, 2001.
11. A. Deutsch and V. Tannen. Containment and integrity constraints for xpath. In Maurizio Lenzerini, Daniele Nardi, Werner Nutt, and Dan Suciu, editors, *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB 2001)*, number 45 in CEUR Workshop Proceedings, 2001.
12. G. Gottlob and C. Koch. Monadic queries over tree-structured data. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, 2002.
13. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. of 28th Conf. on VLDB*, 2002.
14. J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
15. H. Lewis and C. Papadimitriou. *Elements of the theory of computation*. Prentice-Hall, 2 edition, 1997.
16. G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proc. 21th Symposium on Principles of Database Systems (PODS 2002)*, pages 65–76, 2002.
17. G. Moerkotte. Incorporating XSL processing into database engines. In *Proc. of 28th Conf. on VLDB*, 2002.
18. F. Neven. Automata, logic, and XML. In J. C. Bradfield, editor, *CSL*, volume 2471 of *Lecture Notes in Computer Science*, pages 2–26. Springer, 2002.
19. F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3), 2002.
20. F. Neven and T. Schwentick. Automata- and logic-based pattern languages for tree-structured data. Unpublished, 2001.
21. F. Neven and T. Schwentick. Query automata on finite trees. *Theoretical Computer Science*, 275:633–674, 2002.
22. G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41(2–3):305–318, 1985.
23. P. T. Wood. Containment for XPath fragments under DTD constraints. ICDT 2003, this volume.
24. P. T. Wood. Minimising simple XPath expressions. WebDB informal proceedings, 2001.

25. P. T. Wood. On the equivalence of XML patterns. In Lloyd et al., editor, *Computational Logic – CL 2000*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1152–1166. Springer, 2000.

Appendix

Proof of Theorem 5 (continued) (1) We prove the claim:

$p \not\subseteq q \Leftrightarrow$ there is a $t \in T(p,q)$ such that $t \models p$ but $t \not\models q$.

$\Leftarrow$: trivial.

$\Rightarrow$: Assume there is an s matching p but not q. s has to match one of the p_i, hence there is a $(/, //, [\,], *)$-homomorphism h from p_i to s, i.e., h maps the nodes of the tree pattern of p_i to the nodes of s such that (1) $h(v)$ has the same label as v unless v carries a wildcard, (2) $h(v)$ is a child (descendant) of $h(u)$ if and only if v is a child (descendant) of u. We construct t by transforming s in several steps. Let V denote the set of nodes of s in the image of h. We delete all nodes in s that are neither in V nor an ancestor of a node in V. The remaining tree has at most as many leaves as p_i. We replace the label of all remaining nodes not in V by $\#$. Let V' be the set of branching nodes of the tree obtained so far, i.e. of those nodes that have more than one child. The set V' contains at most n vertices. Let a pure path be a path that does neither contain any node from V nor from V'. In particular, the nodes of a pure path are all labelled with $\#$. In the last step we replace all maximal pure paths with $> m+1$ inner nodes by a path with $m+1$ $\#$-labelled inner nodes. We call the resulting tree t.

It is easy to see that $t \in T(p,q)$, that $t \models p_i$ and that t contains at most $m+2$ times $|V| + |V'|$, hence $\leq 2n(m+2)$, many nodes.

We have to show that $t \not\models q$. Towards a contradiction assume that $t \models q_j$, for some j. Hence, there is a homomorphism $h' : q_j \to t$. Next, we show how h' can be modified to obtain a homomorphism from q_j to s which leads to the desired contradiction. Observe first, that the only nodes of q_j that can be mapped to nodes outside V are nodes that are labelled with a $*$. Note also that there is a natural embedding of all those parts of t into s that were not obtained by replacing a long pure path by a shorter one. Hence, for these parts of t the homomorphism h' can be easily modified to get a (partial) homorphism from q_j to s. On the other hand, let $v_1, \ldots, v_k$ be some nodes in the image of h' on a pure path of length $m+1$ with endpoints u and v in $V \cup V'$, ordered from the root to the leaves. By the choice of m it holds that $k \leq m$, therefore there must be an i such that v_{i+1} is not a child of v_i or v_1 is not the first node of the path or v_k is not the last node of the path. In either case there is a mapping from $\{u, v, v_1, \ldots, v_k\}$ to the corresponding original simple path in s which maps u and v to the endpoints of that path and which respects the child and descendant relation.

By composing h' with these mappings and the above mentioned natural embedding we get a $(/, //, [\,], *)$-homomorphism from q_j to s, the desired contradiction. This finishes the proof of the claim.

(2) The hardness proof is the same proof that shows that containment of regular expressions is coNP-hard [15]. We give it for completeness sake and because the next proof depends on it.We use a reduction from validity of propositional logic formulas in disjunctive normal form which is known to be complete for coNP [15]. Let $\varphi = \bigvee_{i=1}^{m} C_i$ be a propositional formula in disjunctive normal form over the variables $x_1, \ldots, x_n$. Here, each C_i is a conjunction of literals. For a disjunct C let $\tilde{C}$ be the XP expression $a_1/\cdots/a_n$ where

$$a_i := \begin{cases} 0 & \text{if } \neg x_i \text{ occurs in } C; \\ 1 & \text{if } x_i \text{ occurs in } C; \\ (0|1) & \text{otherwise.} \end{cases}$$

Let $\tilde{q}$ be the disjunction of the expressions $\tilde{C}_i$, $i = 1, \ldots, m$. Further, let p be the expression $(0|1)/\cdots/(0|1)$ where $(0|1)$ is repeated n times. Clearly, $p \subseteq \tilde{q}$ iff φ is valid.

(3) The reduction is similar to the one above except that we define $\bar{C}$ as $//a_1//a_2//\cdots//a_n$, $\bar{q}$ as the disjunction of the expressions $\bar{C}_i$, $i = 1, \ldots, m$, and p as $(0|1)//\cdots//(0|1)$. We show that $p \subseteq \bar{q} \Leftrightarrow \phi$ is valid. Suppose $p \subseteq \bar{q}$, then in particular $\bar{q}$ matches everey 0-1-string of length n, hence, φ is valid. Therefore, suppose φ is valid. If p matches a branch in the tree then there are in particular n positions with 0 or 1. The ith such position can be seen as a truth assignment to x_i. As φ is valid all possible assignments are accounted for by $\bar{q}$, and $\bar{q}$ matches that branch. □

Proof of Theorem 6 (continued). We start with the definition of alternating automata on finite trees [22]

Definition 16. Fix a natural number k. An *alternating tree automaton (ATA)* is a tuple $A = (k, Q, \Sigma, q_0, \delta, F)$ where Q is a finite set of states, Σ is the alphabet, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\delta : Q \times \Sigma \times \{0, 1, \ldots, k\} \to \mathcal{B}^+(\{0, 1, \ldots, k\} \times Q)$ is the transition function. Here, $\mathcal{B}^+(\{1, \ldots, k\} \times Q)$ denotes the set of positive Boolean formulas over the set $\{1, \ldots, k\} \times Q$.

A configuration on a tree t is a tuple $[u, q]$ where $u \in \mathrm{dom}(t)$ and $q \in Q$. An *accepting run* of A on t is a tree s where nodes are labeled with configurations such that:

- the root of s is labeled with $[\varepsilon, q_0]$;
- let u be a node of s labeled with $[v, q]$ with n children labeled $[v_1, q_1], \ldots, [v_n, q_n]$. Note that v_i is a child of v or v itself. For notational convenience we mean v when we say $v0$. Let ρ be the truth assignment that assigns true to (ℓ, q') iff $q \in F$ or there is an i such that $v_i = v\ell$ and $q_i = q'$. We then require that $\delta(q, \mathrm{lab}_t(v), m)$, where m is the number of children of v, is true under the assignment ρ.

A tree is accepted by A if there is an accepting run. By $L(A)$ we denote the set of trees accepted by A.

Lemma 17. *For every* XP$(/, //, [\,], *, |)$ *pattern p there is a $f(p)$-bounded ATA A_p such that p and A_p are equivalent on all $f(p)$-bounded trees. Moreover, A_p can be constructed in* LOGSPACE.

Proof. Let p be a XP$(/, //, [\,], *, |)$ pattern. As the alphabet is finite, we can replace every $*$ with a disjunction of the alphabet symbols. Hence, we assume p does not contain $*$. Set $k := |p|$. Then define $A_p = (k, Q, \Sigma, q_0, \delta, F)$ where Q is the set of subpatterns of p and the state q_{acc}, $q_0 = p$, and $F = \{q_{\mathrm{acc}}\}$. The transition function is inductively defined as follows: for all subpatterns $p_1, \ldots, p_n$ and for all σ, σ'

– $\delta(/\sigma/p_1, \sigma, 1) = (1, p_1)$;
– $\delta(//\sigma/p_1, \sigma, 1) = (1, //\sigma/p_1) \vee (1, /p_1)$;
– $\delta(//\sigma//p_1, \sigma, 1) = (1, //\sigma//p_1) \vee (1, //p_1)$;
– $\delta(//\sigma/p_1, \sigma', 1) = (1, //\sigma/p_1)$ with $\sigma \neq \sigma'$;
– $\delta(//\sigma//p_1, \sigma', 1) = (1, //\sigma//p_1)$ $\sigma \neq \sigma'$;
– $\delta(/\sigma[p_1] \cdots [p_n], \sigma, n) = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} (i, p_j)$;
– $\delta(//\sigma[p_1] \cdots [p_n], \sigma, n) = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} (i, p_j)$;
– $\delta(//\sigma[p_1] \cdots [p_n], \sigma', 1) = (1, //\sigma[p_1] \cdots [p_n])$; note that σ need not be different from σ';
– For all $i \in \{1, \ldots, \}$, $\delta(p_1 \mid p_2, \sigma, i) = (0, p_1) \vee (0, p_2)$.
– For $\sigma \in \Sigma$, $\delta(\sigma, \sigma, 0) = q_{\mathrm{acc}}$.

The combinations $\delta(a, \sigma, i)$ that are not mentioned are empty. In the first five rules, the node at hand can only have one child. We only allow branching when a filter expression is present. This requirement keeps the automaton bounded.

The requirements in the statement of the lemma are clearly satisfied. □

Lemma 18. *Let p and q be two* XP$(/, //, [\,], *, |)$ *patterns. Testing whether $L(A_p) \subseteq L(A_q)$, where A_p, A_q are the ATA constructed in the previous lemma, can be done in space polynomial in the size of $|p| + |q|$.*

Proof. We only provide a sketch. If $L(A_p) \not\subseteq L(A_q)$ then there is an $f(p)$-bounded tree t such that $t \in L(A_p)$ and $t \notin L(A_q)$. As an $f(p)$-bounded tree consists of at most $f(p) \times f(p)$ paths, the containment problem reduces to containment of alternating string automata which is known to be in PSPACE [4]. □

The result now follows from Lemma 17 and Lemma 18.

We conclude by saying a few words about unary queries. We should be a bit careful when applying the Miklau-Suciu trick of adding a new symbol, say \$, at every position in the pattern where a selection is done: we have to make sure that in every tree only one \$ occurs, but this can easily be done by slightly modifying the construction of A_p. □

Proof of Theorem 7 (continued). The automaton A is constructed as follows. We construct in a straightforward manner a non-deterministic unranked top-down tree automaton A_d which checks whether a tree conforms to d. Next, we construct a non-deterministic unranked top-down tree automaton A_p which checks that a tree has a path conforming to p. This is easy as p is a very simple regular expression which can be transformed into a DFA in polynomial time. Further, as p does not contain filter, the only non-determinism is in the choice of the path in the tree. Finally, A_q is a deterministic unranked top-down tree automaton which verifies that no path in the tree matches q. So A is the product automaton of A_d, A_p and A_q which accepts if all three subautomata accept. □

Proof of Theorem 8 (continued). (1) We first describe algorithm `CheckPnotq`. On input $d, s, P = \{p_1, \dots, p_n\}$ it proceeds as follows.

- First, it checks whether all patterns in P have the root symbol s. If this is not the case return FALSE.
- It guesses a string u of length at most $(|d|+1)(l+1)$, where l is the total number of children of the roots of the patterns in P and verifies that u conforms to the regular expression of s in d and that all non-terminals in u are in the set $U(d)$ of useful symbols.
- For each $i \in \{1, \dots, n\}$, it guesses a mapping f_i from the children $v_1, \dots, v_m$ of the root of p_i to the positions of u.
- For each position j of u, which is in the image of at least one of the mappings f_i, it does the following
  - Let P' be the vertices that are mapped to j.
  - Call `CheckP` recursively with parameters d, u_j, P'. Here, u_j is the symbol at the j-th position in u.
- It returns TRUE iff all the recursive calls return TRUE.

Note that P might consist only of patterns with one node labelled s. In this case the algorithm returns TRUE.

It is relatively straightforward to check that `CheckP` is correct. Of course, if it returns TRUE then there is a tree t with the stated properties. For the converse direction, assume that t is a tree with root s conforming to d which contains all the patterns from P. We show that there t can be transformed into a tree where each node has at most $(|d|+1)(l+1)$ children. We can argue in an inductive fashion. First of all, the sequence of children of the root conforms to the DTD. For each $i \leq n$ there is a homomorphism from p_i to t. Let $v_1, \dots, v_m$ be those children of the root of t that are in the image of at least one of these mappings. Clearly $m \leq l$. Hence, if the root of t has more than $(|d|+1)(l+1)$ children then there is a subsequence of length at least $|d|$ that is not in the image of any h_i. By a standard pumping lemma argument it follows that we can get rid of some of these vertices together with their subtrees. Note that $|d|$ is an upper bound for the number of states of a non-deterministic automaton which describes the regular language of children of s-nodes. Hence, we can assume that t has at most $(|d|+1)(l+1)$ children and we proceed by induction.

Algorithm `Checknotq` works as follows on input d, q.

- If q consists of only one node then it returns FALSE.
- It guesses a string u of length at most $|d|$ and verifies that u conforms to the regular expression of s in d and that all non-terminals in u are in the set $U(d)$ of useful symbols.

– It guesses a child v of the root of q. Let q' be the pattern rooted at this child. If the label s' of v does not occur in u it returns TRUE. Otherwise it calls `Checknotq` with parameters d, q'.

(2–3) We use a reduction from 3SAT to the complement of the containment problem. Let $\varphi = \bigwedge_{i=1}^{k} C_i$ be a CNF formula with variables $x_1, \ldots, x_n$. We construct a DTD d and expressions e, e_1, …, e_k such that

$$\varphi \text{ is satisfiable iff } e \not\subseteq e_1 \mid \cdots \mid e_k. \qquad (*)$$

The DTD is defined as follows: $d(r) = x_1 \cdots x_n$ and $d(x_i) = \text{true} \mid \text{false}$ for every i. Define e as $/r$ and, for every i, e_i as $/r[x_j/\text{true}][x_\ell/\text{false}][x_m/\text{true}]$ when C_i is of the form $(\neg x_j \vee x_\ell \vee \neg x_m)$. Note that $(*)$ holds. Further, the reduction goes through even when every / is replaced by //. This concludes the proof. $\square$

Proof of Theorem 9 (continued)

In the next two proofs we make use of a reduction from TWO-PLAYER CORRIDOR TILING. This is the extension of CORRIDOR TILING, used in the proof of Theorem 6 (1), to two players. Let $T = (D, H, V, \bar{b}, \bar{t}, n)$ be a tiling system. Here, D is a finite set of tiles; $H, V \subseteq D \times D$ are horizontal and vertical constraints, respectively; $\bar{b}, \bar{t}$ are n-tuples of tiles; n is a natural number. There are two players (`I` and `II`) that place in turn tiles on an $n \times \mathbb{N}$ board. On this board the bottom row is tiled with $\bar{b}$. Player `I` starts on the first square of the second row. Each player in turn places a tile on the next free square going from left to right and bottom to top. A player that puts down a tile not consistent with the already placed tiles immediately looses. If player `I` can achieve a tiling of which the top row is consistent with $\bar{t}$ no matter how player `II` plays, then we say that player `I` has a winning strategy. It is well-known that it is EXPTIME-complete to determine whether `I` has a winning strategy [5].

(2) We use the following DTD which defines all possible strategy trees. There is only one terminal symbol: #. The set of non-terminals consists of the start symbol S; two delimiters $\$^1$ and $\$^2$ separating different rows where the index indicates which player should place a tile; and, the symbols (d, k) where $d \in D$ and $k = 1, 2$ indicating which tile has been placed by which player. Let $D = \{d_1, \ldots, d_m\}$. Then we define the productions of the DTD as follows: for every $d \in D$,

$$\begin{aligned}
&S \to (d_1, 1) + \cdots + (d_m, 1)\\
&(d, 1) \to (d_1, 2) \cdots (d_m, 2) + \# + \$^2\\
&(d, 2) \to (d_1, 1) + \cdots + (d_m, 1) + \# + \$^1\\
&\$^2 \to (d_1, 2) \cdots (d_m, 2)\\
&\$^1 \to (d_1, 1) + \cdots + (d_m, 1)
\end{aligned}$$

Note that a derivation tree encodes a possible strategy tree (or game tree) for `I`. Indeed, tiles labeled with $(d, 1)$ have all other tiles as children (indicating that

I needs an answer for every choice of II, while tiles carrying a $(d, 2)$ only have one child (indicating that I should have only one answer for every tile placed by II). As the bottom and the top row are fixed we do not represent them in the strategy trees. I.e., these trees only represent intermediate rows. We assume that the tiling consisting only of the top and bottom row is not valid. Therefore any strategy tree has to represent at least one row.

We have to check whether there is a tree encoding a valid strategy tree for I. If no such tree exists then I cannot win. To check whether a tree is a valid strategy tree, we have to verify whether the horizontal and vertical constraints are satisfied and whether every row has exactly n tiles. Actually, we will test for the converse. That is, we will construct a union of XPath expressions that select a tree when it does not encode a strategy for I.

More precisely, we construct an expression p such that $/S//\# \subseteq p$ iff player I has no winning strategy. We define

$$p := q_{n \text{ tiles}} \mid \bigcup_{(d,d')\notin H} q^H_{d,d'} \mid \bigcup_{(d,d')\notin V} q^V_{d,d'} \mid q_{\bar{b}} \mid q_{\bar{t}},$$

where the expressions on the right hand side will be defined shortly. We use $\bigcup$ to denote a big disjunction of expressions. Each of the above expressions identifies an error in the strategy tree. Hence, if every tree matches one of these expressions, every tree contains an error and no tree can be a valid strategy tree.

Note that, although the expressions under consideration do not have the wildcard available, the disjunction of all alphabet symbols defined by the grammar is a kind of wildcard as the DTD assures that no other symbols occur in the tree. In the rest of this proof, $*$ is an abbrevation for the expression that denotes the disjunction of all alphabet symbols defined by the grammar, and $*^i$ is an abbreviation for $/*/*/\cdots/*$ (i times).

Vertical Constraints are violated. For every $d, d' \in D$,

$$q^V_{d,d'} := \bigcup_{k,\ell=1,2} //(d,k)/*^{n+1}/(d',\ell)$$

Further, define

$$q_{\bar{b}} := \bigcup_{i=1}^{n} \bigcup_{k=1,2} \bigcup_{(b_i,d)\notin V} /S/*^{i-1}/(d,k),$$

checking the vertical constraints w.r.t. $\bar{b}$. Define

$$q_{\bar{t}} := \bigcup_{k=1,2} \bigcup_{i=1}^{n} \bigcup_{(d,t_i)\notin V} //(d,k)/*^{n-i}/\#,$$

checking the vertical constraints w.r.t. $\bar{t}$.

Horizontal Constraints are violated. For every $d, d' \in D$,

$$p^H_{d,d'} := \bigcup_{k,\ell=1,2} //(d,k)/(d',\ell)$$

A row does not contain exactly n tiles.

$$q_{n \text{ tiles}} := D^{n+1} \mid \bigcup_{i=0}^{n-1} (\$^1 \mid \$^2 \mid S)/D^i/(\$^1 \mid \$^2 \mid \#)$$

Here, D stands for the expression $((d_1,1) \mid \cdots \mid (d_m,1) \mid (d_1,2) \mid \cdots \mid (d_m,2))$ and D^i stands for the expression $/D/\cdots/D/$ (i times).

(3) The construction in this case is similar to the construction in (2). First of all, we define the expression $q_{n \text{ tiles}}$ slightly different in order to get rid of the inner disjunctions.

$$q_{n \text{ tiles}} := \bigcup_{\sigma \in \{\$^1,\$^2,S\}} \bigcup_{i=1}^{m} \bigcup_{j=1}^{2} \sigma/*^n/(d_i,j) \mid$$
$$\bigcup_{i=0}^{n-1} \bigcup_{\substack{\sigma \in \{\$^1,\$^2,S\} \\ \sigma' \in \{\$^1,\$^2,S\}}} \sigma/*^i/\sigma'$$

The expression of the first line matches if the symbol at distance $n+1$ from a delimiter $\$^1, \$^2, S$ is not a delimiter $\$^1, \$^2, \#$. The expression in the second line matches if delimiters occur in distance less than $n+1$.

The outermost union can be handled by Lemma 1 of [16]. Of course, the DTD has to be adapted accordingly. □

Proof of Theorem 10. We use a reduction from Post's Correspondence Problem (PCP) which is well-known to be undecidable [14]. An *instance* of PCP is a sequence of pairs $(x_1,y_1),\ldots,(x_n,y_n)$, where $x_i, y_i \in \{a,b\}^*$ for $i = 1,\ldots,n$. This instance has a *solution* if there exist $m \in \mathbb{N}$ and $\alpha_1,\ldots,\alpha_m \in \{1,\ldots,n\}$ such that $x_{\alpha_1}\cdots x_{\alpha_m} = y_{\alpha_1}\cdots y_{\alpha_m}$.

We construct a DTD d, and two XPath expressions p_1 and p_2 such that $p_1 \subseteq_d p_2$ iff the PCP instance has a solution.

We consider XML trees that are almost unary trees or, equivalently, simply strings. They are of the form $u\$v$, where \$ is a delimiter and u, v are strings representing a candidate solution $(x_{\alpha_1},\ldots,x_{\alpha_m}; y_{\beta_1},\ldots,y_{\beta_m})$ for the PCP instance in a suitable way. To check whether such a candidate is indeed a solution, we roughly have to check whether

1. $\alpha_i = \beta_i$ for each i, that is, corresponding pairs are taken; and
2. both strings are the same, that is, corresponding positions in $x_{\alpha_1}\cdots x_{\alpha_m}$ and $y_{\alpha_1}\cdots y_{\alpha_m}$ carry the same symbol.

To check the correspondences mentioned in (1) and (2), we make use of a double indexing system based on string-values.

We explain the intuition behind our reduction by means of a small concrete example.

Example 19. Consider the following PCP instance:

$$\begin{aligned} x_1 &:= ab \qquad & y_1 &:= a \\ x_2 &:= b \qquad & y_2 &:= bb. \end{aligned}$$

We want to encode possible solutions $x_1x_2; y_1y_2$ by means of an almost linear tree. Each $x_1 = ab$, for instance, will be represented by a block of the form

$$\begin{tikzcd}
 & X \arrow[dl, dash] \arrow[dr, dash] & & \\
\text{block} & & 1(x) \arrow[d, dash] & \\
 & & a(x,1,1) \arrow[dl, dash] \arrow[dr, dash] & \\
 & \text{position} & & b(x,1,2) \arrow[d, dash] \\
 & & & \text{position}
\end{tikzcd}$$

The structure of the block is determined by the labels of the right-descendants. Here, X indicates the beginning of a block; $1(x)$ means that x_1 is picked; and, $a(x,1,1)$ $b(x,1,2)$ encode that x_1 is the string ab. More precisely, $\sigma(x,i,j)$ encodes that the j-th position in the string x_i is σ. We need this involved encoding as we will define a DTD that can only produce valid sequences of blocks. The elements "block" and "position" make up the double index system as will become clear further on. Similarly, y_2 is encoded by the block

$$\begin{tikzcd}
 & Y \arrow[dl, dash] \arrow[dr, dash] & & \\
\text{block} & & 2(y) \arrow[d, dash] & \\
 & & b(y,2,1) \arrow[dl, dash] \arrow[dr, dash] & \\
 & \text{position} & & b(y,2,2) \arrow[d, dash] \\
 & & & \text{position}
\end{tikzcd}$$

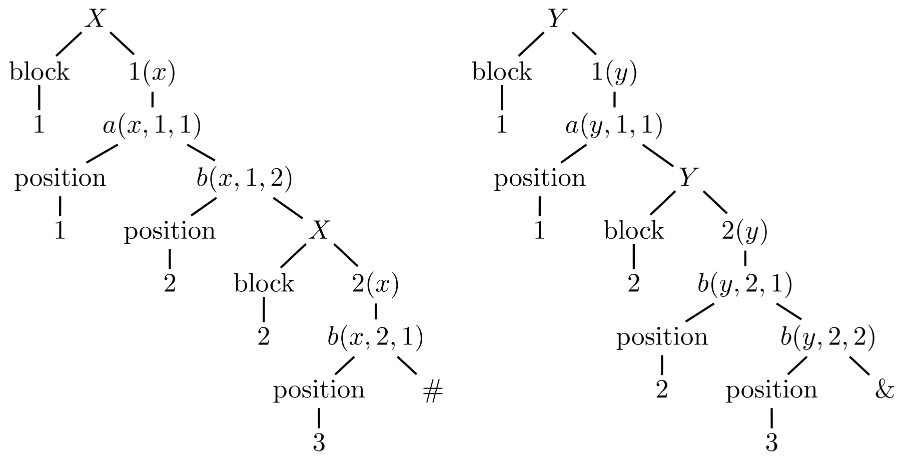
We refer to blocks corresponding to encodings of an x_i (y_i) as X-blocks (Y-blocks). If a block corresponds to x_i or y_i the we say that its number is i.

The DTD defines trees of the form

$$S \; u\#v\&,$$

where u is a sequence of X-blocks and v is a sequence of Y-blocks. To check that such a tree is indeed a solution we need to check that the block numbers of corresponding blocks are the same and that the values (a or b) of corresponding positions in the output string are the same. To this end, we use the element types *block* and *position*. The block element is associated to X and Y labeled nodes, the position element is associated to nodes labeled with $\sigma(z,i,j)$ elements.

A correct encoding of the candidate solution $x_1x_2; y_1y_2$ would be the following tree:



Here, the values of the *block* and *position* elements determines corresponding blocks and corresponding positions within the strings.

The DTD d which derives trees of such form is given by the following set of productions:

$$\begin{array}{rcl}
S & \rightarrow & X \\
X & \rightarrow & \text{block}\,1(x) \mid \text{block}\,2(x) \\
1(x) & \rightarrow & a(x,1,1) \\
a(x,1,1) & \rightarrow & \text{position}\,b(x,1,2) \\
b(x,1,2) & \rightarrow & \text{position}\,X \mid \text{position}\,\# \\
2(x) & \rightarrow & b(x,2,1) \\
b(x,2,1) & \rightarrow & \text{position}\,X \mid \text{position}\,\# \\
\# & \rightarrow & Y \\
Y & \rightarrow & \text{block}\,1(y) \mid \text{block}\,2(y) \\
1(y) & \rightarrow & a(y,1,1) \\
a(y,1,1) & \rightarrow & \text{position}\,Y \mid \text{position}\,\& \\
2(y) & \rightarrow & b(y,2,1) \\
b(y,2,1) & \rightarrow & \text{position}\,b(y,2,2) \\
b(y,2,2) & \rightarrow & \text{position}\,Y \mid \text{position}\,\& \\
\& & \rightarrow & \varepsilon \\
\text{block} & \rightarrow & \text{PCDATA} \\
\text{position} & \rightarrow & \text{PCDATA}
\end{array}$$

Of course, the schema does not ensure that the string-values of the nodes are defined as explained above. We will define a pattern that will select the root of each string iff this string is not a valid encoding or, if it is a valid encoding but does not represent a solution.

To define the DTD for the general case we introduce some notation, for $i := 1,\ldots,n$, let $x_i := \sigma_1^i \cdots \sigma_{k_i}^i$ and $y_i := \delta_1^i \cdots \delta_{\ell_i}^i$. Further, define $P(x_i)$ as the set of productions $i(x) \to \sigma_1^i(x,i,1)$, and for $j := 1,\dots,k_i - 1$, $\sigma_j^i(x,i,j) \to$ position $\sigma_{j+1}^i(x,i,j{+}1)$, and $\sigma_{k_i}^i(x,i,k_i) \to$ position $X \mid$ position $\#$. Analogously, let $P(y_i)$ be the set of productions $i(y) \to \delta_1^i(y,i,1)$, and for $j := 1,\dots,\ell_i - 1$, $\delta_j^i(y,i,j) \to$ position $\delta_{j+1}^i(y,i,j+1)$, and $\delta_{\ell_i}^i(y,i,\ell_i) \to$ position $Y \mid$ position $\&$.

The DTD d consists of the productions

$$\begin{array}{ll} S & \to X \\ X & \to \text{block}\,1(x)|\dots|\text{block}\,n(x) \\ \# & \to Y \\ Y & \to \text{block}\,1(y)|\dots|\text{block}\,n(y) \\ \& & \to \varepsilon \\ \text{block} & \to \text{PCDATA} \\ \text{position} & \to \text{PCDATA} \end{array}$$

together with $P(x_i)$ and $P(y_i)$ for $i = 1,\dots,n$. The start symbol is S.

Formally, a tree $u\#v\&$ is *syntactically correct* if u and v contain the same number of blocks and it fulfills the following two conditions. For $z \in \{u,v\}$, let block(z) be the list consisting of the string-values of the block nodes in z and let position(z) be the list consisting of the string-values of the position nodes in z. Then it should be the case that block(u) = block(v) and position(u) = position(v).

A syntactically correct string $u\$v\&$ *represents a solution of the PCP instance*, iff the block numbers of corresponding blocks are the same and the values (a or b) of corresponding positions in the output string are the same.

Let p_1 be the XPath expression $/S$ and let d be as above. We next construct p_2 in such a way that it selects the root of an XML document if and only if it is *not* syntactically correct or does *not* represent a solution. Hence, p_2 accepts *all* inputs and therefore $p_1 \subseteq_d p_2$ if and only if the PCP instance has *no* solution.

The XPath expressions is a union of the following expressions. Each of them represents an error. In the following if z is the string $abab$ then $/z$ is a shorthand for $/a/b/a/b$. Further, denote the string generated from x_i (y_i) by $\tilde{x}_i$ ($\tilde{y}_i$).

1. The block index is wrong.
   (a) the block value of the first X in u differs from the block value of the first Y in v:
   $$/S[\text{not}(X/\text{block} = //\#/Y/\text{block})].$$
   (b) the block value of the last X in u differs from the block value of the last Y in v: for each $i,j \in \{1,\dots,n\}$ we have the XPath expression
   $$/S[\text{not}(//X[i(x)/\tilde{x}_i/\#]/\text{block} = //Y[j(y)/\tilde{y}_i/\&]/\text{block})].$$
   (c) two X-block values are the same:
   $$/S//X[\text{block} = //X/\text{block}]$$
   (d) two Y-block values are the same;
   $$/S//Y[\text{block} = //Y/\text{block}]$$

(e) there is an X-block that does not occur as a Y-block:

$$/S[//X[not(\text{block} = //Y/\text{block})]]$$

(f) there is an Y-block that does not occur as an X-block:

$$/S[//Y[not(\text{block} = /S//X/\text{block})]]$$

(g) the indices in X are not successive: for all $i \in \{1, \dots, n\}$

$$/S[//X[\text{block} >= i(x)/\tilde{x}_i/X/\text{block}]]$$

(h) the indices in Y are not successive: for all $j \in \{1, \dots, n\}$

$$/S[//Y[\text{block} >= j(y)/\tilde{y}_j/Y/\text{block}]]$$

2. The position index is wrong. This is checked in an analogous fashion. In the next expressions we make use of the star. However, we can get rid of the star by having an expression for each possible match. Indeed, each star can only be matched by a symbol of the form $i(z)$, $\sigma(z, i, j)$ where i, j are numbers depending on the PCP instance, $\sigma = a, b$, and $z = x, y$.

(a) the first position in u differs from the first position in v:

$$/S[\text{not}(/X/*/*/\text{position} = //\#/Y/*/*/\text{position})].$$

(b) the last position in u differs from the last in v:

$$/S[\text{not}(//*[\#]/\text{position} = //*[\&]/\text{position})]$$

(c) two X-position values are the same:

$$/S[//*[\text{position} = //*/\text{position}[//\#]]]$$

(d) two Y-position values are the same;

$$/S[//\&///*[\text{position} = //*/\text{position}]]]$$

(e) there is an X-position that does not occur as an Y-position: for every $\sigma_1, \sigma_2 \in \{a, b\}$, $i, j \in \{1, \dots, n\}$, $i_1, i_2 \in \{1, \dots, |x_i|\}$, and $j_1, j_2 \in \{1, \dots, |y_j|\}$ we have the XPath expression:

$$/S[//\sigma_1(x, i_1, i_2)[not(\text{position} = //\sigma_2(y, j_1, j_2)/\text{position})]]$$

(f) there is an Y-position that does not occur as an X-position: for every $\sigma_1, \sigma_2 \in \{a, b\}$, $i, j \in \{1, \dots, n\}$, $i_1, i_2 \in \{1, \dots, |x_i|\}$, and $j_1, j_2 \in \{1, \dots, |y_j|\}$ we have the XPath expression:

$$/S[//\sigma_2(y, j_1, j_2)[not(\text{position} = /S//\sigma_1(x, i_1, i_2)/\text{position})]]$$

(g) the position indices in X are not successive: we have to deal with several cases as the successive positions can occur in the same block or in successive blocks.

   i. the X-positions occur in the same block for all $i \in \{1, \dots, n\}$ and $k \in \{1, \dots, |x_i| - 1\}$ we have the XPath expression

$$/S//X/i(x)/*^{k-1}\,[\text{position} >= /*/\text{position}].$$

   ii. the X-positions occur in successive blocks for all $i \in \{1, \dots, n\}$ we have the XPath expression

$$/S//X/i(x)/*^{|x_i|-1}\,[\text{position} >= X/*/*/\text{position}]$$

(h) the position indices in Y are not successive: similar.

3. w does not represent a solution:

(a) The block number for some block in u is different from the corresponding block in v: for all $i, j \in \{1, \dots, n\}$ with $i \neq j$ we have the XPath expression

$$/S[//X[i(x)]/\text{block} = //Y[j(y)]/\text{block}]$$

(b) The a/b-value for some δ in u is different from the corresponding a/b-value in v. Thereto, we have the following expressions: for all $i, j \in \{1, \dots, n\}$, $k \in \{1, \dots, |x_i|\}$, $\ell \in \{1, \dots, |y_j|\}$

$$/S[//a(x, i, k)/\text{position} = //\#//b(y, j, \ell)/\text{position}]$$

and

$$/S[//b(x, i, k)/\text{position} = //\#//a(y, j, \ell)/\text{position}]$$

Clearly, w is not a solution iff one of the above conditions hold.

□

Lemma 20. – *Containment testing for XP*$(/, [\,], vars)$ *is* CONP*-hard.*
– *Containment testing for XP*$(/, [\,], vars, \neq)$ *is* Π_2^P*-hard.*

Proof. Let Q be a Boolean CQ of the form $L_1, \dots, L_n, x_1 \neq x_2, \dots, x_{m-1} \neq x_m$. Before we define corresponding XP-expressions we first describe how we represent a database as a tree. Let **DB** be a database. The root is labeled with S and for every relation R in **DB** and every tuple $(d_1, \dots, d_n)$ in R it has a child labeled R with n attributes @1, …, @n, where, for each i, @i has the value d_i.

The XP pattern p_Q has the form $/S[p_1]\cdots[p_n]$ where every p_i is obtained from L_i in the following way: if L_i is of the form $R(y_1, \dots, y_k)$ and the inequalities $y_{i_1} \neq z_{j_1}, \dots y_{i_r} \neq z_{j_r}$ occur in $x_1 \neq x_2, \dots, x_{m-1} \neq x_m$ then p_i is the pattern

$$R[\$y_1 = @1]\cdots[\$y_k = @k][\$z_{j_1} \neq @i_1]\cdots[\$z_{j_r} \neq @i_r].$$

We illustrate the construction with an example. For instance, if Q equals

$$E(x, y), E(y, z), x \neq z$$

then p_Q is

$$/S[E[\$x = @1][\$y = @2][\$z \neq @1][E[\$y = @1][\$z = @2][\$x \neq @2]].$$

As containment of CQs and CQs with inequality is hard for CONP and Π_2^P, respectively, it suffices to show that for all Q_1, Q_2, $Q_1 \subseteq Q_2$ iff $p_{Q_1} \subseteq p_{Q_2}$.

Clearly, if $Q_1 \not\subseteq Q_2$ then there is a database $\mathbf{DB}$ such that $\mathbf{DB} \models \mathbf{Q_1}$ and $\mathbf{DB} \not\models \mathbf{Q_2}$. Clearly, $t_{\mathbf{DB}}$, as described above, matches p_{Q_1} but not p_{Q_2}.

Suppose that $p_{Q_1} \not\subseteq p_{Q_2}$ and let t be the tree that matches p_{Q_1} but not p_{Q_2}. We then take the restriction of t that suffices to match p_{Q_1} and transform it into a database. □

Proof of Theorem 14 (continued) We prove (a).
Let p and q be patterns and let $p_1|...|p_m$ and $q_1|\dots|q_n$ be the DNFs of p and q, respectively. Note that the disjuncts can again be represented as tree patterns, this time with additional constraints reflecting the equalities and inequalities between variables and attributes.

Clearly, $p \not\subseteq q$ if and only if for some i, $p_i \not\subseteq q$. Hence, in proving (a) we can restrict to the case where p does not contain $|$. Let A be the set of attributes that occur in p or q.

We call a tree t an A-canonical tree for (p, q) if the following conditions hold.

– The tree structure of t is obtained from the tree patterns of p by replacing each //-edge by two child edges with a new intermediate #-labeled node where # is a label not occuring in p or q. Note that the number of vertices of t is at most twice the number of vertices of the tree pattern of p.
– The attribute values in t are from the set $\{0, \dots, mk\}$, where m is the number of vertices in t and k is the number of attributes in A.

Let $S(p, q)$ denote the set of all A-canonical trees for (p, q). Note that, as the data values are bounded by $km = (|p| + |q|)m$ these trees can be encoded by strings of polynomial size.

We show next, that whenever $p \not\subseteq q$ for a pattern p from $\mathrm{XP}(/, //, [\,], \mathrm{vars}, \neq)$ and a pattern q from $\mathrm{XP}(/, //, [\,], |, \mathrm{vars}, \neq)$ then there is a tree $t \in S(p, q)$ that matches p but not q.

Let therefore $p \not\subseteq q$ be witnessed by a tree t' not necessarily from $S(p, q)$. Hence, $t' \models p$ but $t' \not\models q$. Let e be a homomorphism from p to t'. Let $a_1, \dots, a_m$ be the pairwise different attribute values of the vertices in $e(p)$.

We construct the tree t as follows. Its structure is obtained from p as above by replacing //-edges with new nodes labeled #. We call a vertex v of t that is already in p an *original vertex* and write $p(v)$ for its corresponding vertex in p. An original vertex v of t inherits its attribute values from $e(p(v))$ as follows. If attribute b of $e(p(v))$ has value a_i then v gets the attribute value i. The attributes of the other nodes get the value 0.

Let u and u' be (not necessarily distinct) original vertices in t and let b, b' be two attributes. Then the b-attribute of u is different from the b'-attribute of u' if and only if the b-attribute of $e(p(u))$ is different from the b'-attribute of $e(p(u'))$.

Clearly, $t \in S(p,q)$ and $t \models p$ via the obvious homomorphism. It remains to show that $t \not\models q$. Assume otherwise. Hence, for some j, $t \models q_j$. Let e' be a homomorphism from q_j to t. As q does not contain the symbol $\#$ and there are no wildcards, the image of q_j under e' only contains original vertices of t. As these vertices have the same relationships within each other as their corresponding vertices in t' we can conclude that t' also matches q_j via the homomorphism $e \circ p \circ e'$, the desired contradiction. This concludes the proof of (a).

Proof of Theorem 15. Again the reduction is from PCP. The structure of the proof is similar to the proof of Theorem 10. We only mention the differences and make use of the notation introduced in the latter proof. The crucial difference is that the double index system is no longer encoded by the string-values of *elements* but by the values of attributes. Indeed, the *elements* block and position in the proof of Theorem 10 are now attributes. We first describe a construction which still needs the presence of a DTD. Eventually, we will explain how we can get of the DTD. Our DTD will define *strings* of the form $S\ u\#v\&$. For instance, the candidate solution $x_1x_2; y_1y_2$ of Example 19 will be represented as follows:

	S	X	$1(x)$	$a(x,1,1)$	$b(x,1,2)$	X	$2(x)$	$b(x,2,1)$	$\#$
block		1				2			
position				1	2			3	

	Y	$1(y)$	$a(y,1,1)$	Y	$2(y)$	$b(y,2,1)$	$b(y,2,2)$	$\&$
block	1			2				
position			1			2	3	

So we have the DTD D which consists of the productions

$$\begin{aligned} S &\to X \\ X &\to 1(x) | \dots | n(x) \\ \# &\to Y \\ Y &\to 1(y) | \dots | n(y) \\ \& &\to \varepsilon \end{aligned}$$

together with $P(x_i)$ and $P(y_i)$ for $i = 1, \dots, n$. Here, $P(x_i)$ consists of the productions $i(x) \to \sigma_1^i(x, i, 1)$, and for $j := 1, \dots, k_i - 1$, $\sigma_j^i(x,i,j) \to \sigma_{j+1}^i(x,i,j+1)$, and $\sigma_{k_i}^i(x,i,k_i) \to X \mid \#$. $P(y_i)$ is defined analogously. The start symbol is S. Every X and Y has an attribute *block*; every a and b has an attribute *position*. The XPath expression p_1 is again $/S$ while p_2 is a union of expressions each of which identifies an error.

1. The block index is wrong.
   (a) the block value of the first X in u differs from the block value of the first Y in v:
   $$/S/X[\$d = @block]//\#/Y[\$d \neq @block].$$
   (b) the block value of the last X in u differs from the block value of the last Y in v: for each $i, j \in \{1, \dots, n\}$ we have the XPath expression
   $$/S//X[\$d = @block]/i(x)/\tilde{x}_i/\#//Y[\$d \neq @block]/j(y)/\tilde{y}_i\&$$

(c) two X-block values are the same:

$$/S//X[\$d = @block]//X[\$d = @block]//\#$$

(d) two Y-block values are the same;

$$/S//\#//Y[\$d = @block]//Y[\$d = @block]$$

(e) two successive X-block values are not successive in v: for all $i, j \in \{1, \dots, n\}$ we have the XPath expression
$/S//X[\$d = @block]/i(x)/\tilde{x}_i/X[\$e = @block]//$
$\#//Y[\$d = @block]/j(y)/\tilde{y}_j/Y[\$e \neq @block]$.

2. The position index is wrong. This is done in an analogous fashion.

(a) the first position in u differs from the first position in v:

$$/S/X/*/*[\$d = @position]//Y/*/*[\$d \neq @position].$$

(b) the last position in u differs from the last in v:

$$/S//*[\$d = @position]/\#//*[\$d \neq @position]/\&$$

(c) two X-position values are the same:

$$/S//*[\$d = @position]//*[\$d = @position]//\#$$

(d) two Y-position values are the same;

$$/S//\#//*[\$d = @position]//*[\$d = @position]$$

(e) two successive X-position values are not successive in v: we have to deal with several cases as the successive positions can occur in the same block or in successive blocks.

i. the X-positions occur in the same block, the Y-positions occur in the same block: for all $i, j \in \{1, \dots, n\}$, $k \in \{1, \dots, |x_i| - 1\}$, $\ell \in \{1, \dots, |y_j| - 1\}$ we have the XPath expression

$$/S//X/i(x)/*^{k-1}/*[\$d = @position]/*[\$e = @position]$$
$$//Y/j(y)//*^{\ell-1}/*[\$d = @position]/*[\$e \neq @position]$$

ii. the X-positions occur in successive blocks, the Y-positions occur in the same block: for all $i, j \in \{1, \dots, n\}$, $\ell \in \{1, \dots, |y_j| - 1\}$ we have the XPath expression

$$/S//X/i(x)/*^{|x_i|-1}/*[\$d = @position]/X/*/*[\$e = @position]$$
$$//Y/j(y)//*^{\ell-1}/*[\$d = @position]/*[\$e \neq @position]$$

iii. the X-positions occur in the same block, the Y-positions occur in successive blocks: for all $i, j \in \{1, \ldots, n\}$, $k \in \{1, \ldots, |x_i| - 1\}$ we have the XPath expression

$$/S//X/i(x)/*^{k-1}/*[\$d = @\mathit{position}]/*[\$e = @\mathit{position}]$$
$$//Y/j(y)//*^{|y_j|-1}/*[\$d = @\mathit{position}]/Y/*/*[\$e \neq @\mathit{position}]$$

iv. the X-positions occur in successive blocks, the Y-positions occur in successive blocks: for all $i, j \in \{1, \ldots, n\}$ we have the XPath expression

$$/S//X/i(x)/*^{|x_i|-1}/*[\$d = @\mathit{position}]/X/*/*[\$e = @\mathit{position}]$$
$$//Y/j(y)//*^{|y_j|-1}/*[\$d = @\mathit{position}]/Y/*/*[\$e \neq @\mathit{position}]$$

3. w does not represent a solution:
   (a) The block number for some block in u is different from the corresponding block in v: for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$ we have the XPath expression

$$/S//X[\$d = @\mathit{block}]/i(x)//Y[\$d = @\mathit{block}]/j(y)$$

   (b) The a/b-value for some δ in u is different from the corresponding a/b-value in v. Thereto, we have the following expressions: for all $i, j \in \{1, \ldots, n\}$, $k \in \{1, \ldots, |x_i|\}$, $\ell \in \{1, \ldots, |y_j|\}$

$$/S//a(x, i, k)[\$d = @\mathit{position}]//\#//b(y, j, \ell)[\$d = @\mathit{position}]$$

and

$$/S//b(x, i, k)[\$d = @\mathit{position}]//\#//a(y, j, \ell)[\$d = @\mathit{position}]$$

Clearly, w is not a solution iff one of these conditions holds.

It remains to show how to get rid of the DTD. In the proof of Theorem 10 we needed negation to express that a certain node can *not* have a certain label. Of course, when labels come from a finite alphabet we do not need explicit negation (c.f., the proof of Theorem 6(2)). For this reason, we encode labels of nodes by equality types of attribute values. So, let $L := \ell_1, \ldots, \ell_m$ be an enumeration of all the labels we need. Every node has m attributes $a_1, \ldots, a_m$. If for a node, $j > 1$ is the smallest number such that the value of a_1 equals the value of a_j then the node is considered as labeled with ℓ_j (when all attributes are different, then the node is considered as labeled with a_1). One can match a node labeled with ℓ_j by checking the corresponding equality type: for instance, by the expression

$$*[\$x_1 = @a_1][\$x_2 \neq @a_1] \cdots [\$x_{j-1} \neq @a_1][\$x_j = @a_1][\$x_2 = @a_2] \cdots [\$x_{j-1} = @a_{j-1}].$$

Clearly, when using this approach we can express that a certain node is not labeled by a certain label. For every rule $a \to b_1 \mid \cdots \mid b_k$ in the DTD we add the disjunct $//a/c$ to p_2 where $c \in L \setminus \{b_1, \ldots, b_k\}$. When we write $//a/c$, we of course mean XPath expressions taking labels into account as specified in the manner above. Let p_3 be obtained from p_2 by adding all the disjuncts from the DTD and replacing all references to labeling by references to encoding with attributes.

It remains to argue that $p_1 \not\subseteq p_3$ iff the PCP instance has a solution. When there is a solution to the PCP then clearly the encoding of this string will match p_1 but not p_3. Suppose that there is a tree that matches p_1 but not p_3. This means that no error occurs on any path in the tree. Therefore, every path is an encoding of a solution to the PCP instance. $\square$