Logical Methods in Computer Science
Volume 00, Number 0, Pages 000–000
S 0000-0000(XX)0000-0

ON THE COMPLEXITY OF XPATH CONTAINMENT IN THE PRESENCE OF DISJUNCTION, DTDS, AND VARIABLES

FRANK NEVEN AND THOMAS SCHWENTICK

Hasselt University and Transnational University of Limburg
E-mail address: frank.neven@uhasselt.be

University of Dortmund
E-mail address: thomas.schwentick@udo.edu

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. We restrict attention to the most common XPath expressions which navigate along the child and/or descendant axis. In addition to basic expressions using only node tests and simple predicates, we also consider disjunction and variables (ranging over nodes). Further, we investigate the containment problem relative to a given DTD. With respect to variables we study two semantics, (1) the original semantics of XPath, where the values of variables are given by an outer context, and (2) an existential semantics introduced by Deutsch and Tannen, in which the values of variables are existentially quantified. In this framework, we establish an exact classification of the complexity of the containment problem for many XPath fragments.

1. Introduction

XPath is a simple language for navigating an XML document and selecting a set of element nodes [9]. At the same time it is also the main XML selection language. Indeed, XPath expressions are used, for instance, as basic patterns in several XML query languages like XQuery [5] and XSLT [3, 10]; they are used in XML Schema to define keys [11], and in XLink [13] and XPointer [12] 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 article we focus on the complexity of the containment problem of various fragments of XPath 1.0 using only the most common axes, / and //, and extensions in which variables can refer to data values. Furthermore, the containment problem relative to a given DTD is investigated. In all cases, we only consider the Boolean containment problem. Here, given two XPath expressions p and q, the problem asks whether the fact that p selects some path from the root to a vertex

2000 ACM Subject Classification. H.2, I.7.2, F.4.
Key words and phrases. XPath, XML pattern language, containment, complexity, automata.

implies that also q selects some path from the root to a vertex. This restriction is justified as all complexity results we obtain easily transfer to unary and binary containment in the spirit of Proposition 1 in [27].

The XPath containment problem already attracted quite some attention [14, 27, 28, 36, 37, 24]. We next discuss the known results together with our own contributions.

A general result establishing a strong upper bound for a large fragment of XPath is due to Marx presented in [24]. It is shown there that the containment problem for navigational XPath, allowing navigation along all axes, even relative to a DTD, is in EXPTIME.

Other work has concentrated, like this article, on XPath expressions that can only navigate downwards in an XML tree and do not use the order between siblings, i.e., the fragment using only the / and // axis. Different fragments can be defined by allowing or disallowing the use of the wild-card $*$ in node tests, and filter predicates in location steps. In the spirit of the abbreviated syntax of XPath we use / to indicate the use of the `child` axis, // for the `descendant` axis, $*$ for the wild-card and [ ] for filter predicates. We denote XPath fragments by listing the allowed operators. For instance, XP(/,//,[ ]) denotes the XPath fragment with the `child` and `descendant` axes in which the use of filter predicates is allowed, but no wild-cards in node tests.

Among other results, Miklau and Suciu [27] obtain that containment for XP(/,//, [ ],$*$) is CONP-complete [27]. Here, inside filter predicates and between location steps, no Boolean operators are allowed.

Contributions. The first family of fragments we consider is obtained by allowing disjunction (|) in filter predicates and in location steps. We show that, in principle, adding disjunction to XP(/,//, [ ],$*$) does not make the containment problem harder. Surprisingly, when the set of allowed element names (labels) in XML documents is restricted, and given as part of the input then the containment problem becomes much harder: complete for PSPACE. The results on fragments with disjunction are shown in Table 1

/	//	[ ]	|	$*$	Complexity	Reference
+	+	+		+	CONP-complete	[27]
+	+	+	+	+	CONP-complete	(3.2,3.1)
+			+		CONP-complete	(3.2,3.1)
	+		+		CONP-complete	(3.2,3.1)
+	+	+	+	+	PSPACE-complete (given alphabet)	(3.3,3.5)
+	+		+		PSPACE-complete (given alphabet)	(3.3,3.5)

Table 1: The complexity of containment for expressions with disjunction. Square brackets refer to the references, parentheses to results of this article.

Deutsch and Tannen [14] consider XPath containment in the presence of DTDs and Simple XPath Integrity Constraints (SXICs) [14]. Here, the input to the containment problem consists of two XPath expressions p, q, a DTD and/or a set of integrity constraints and it is asked whether p selects a subset of the elements that q selects, in all documents respecting the DTD and/or the constraints. They show 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 indicate the presence of a DTD by XP(DTD,...). We give a simple proof that containment testing for XP(DTD,/,//,[ ],$*$,|) is in EXPTIME (although this result is covered by the above mentioned result of Marx [24]) and obtain that containment for XP(DTD,/,//,|) and for XP(DTD,/,//,[ ],∗) are hard for EXPTIME. We also study the complexity of more restrictive fragments in the presence of DTDs. It turns out that containment of XP(DTD,/,//) is in PTIME. On the other hand, 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 in the former case can be extended to include, for instance, the descendant operator. The results about the containment problem in the presence of DTDs are summarized in Table 2.

DTD	/	//	[ ]	|	∗	Complexity	Reference
+	+	+				in P	(4.1)
+	+		+			CONP-complete	(4.2,4.3,[35])
+		+	+			CONP-hard	(4.3,[35])
+	+	+	+	+	+	EXPTIME-complete	(4.4,4.5)
+	+	+		+		EXPTIME-complete	(4.4,4.5)

Table 2: The complexity of containment in the presence of DTDs.

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). We indicate the use of variables with XPath semantics by XP(...,xvars,...). So the semantics of an XPath expression is defined with respect to 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 expressions without variables.

In addition to the XPath semantics, Deutsch and Tannen [14] considered an existential semantics for variables: an expression matches a document if there *exists* a suitable assignment for the variables. We denote variables with existential semantics by XP(...,evars,...). In [14] it is shown that containment of XP(/,//,[ ],∗,evars) and XP(/,//,[ ],|,evars) is Π_2^P-hard, and that containment of XP(/,//, [ ],|,evars) under fixed bounded SXICs is in Π_2^P. We extend their result by showing that containment of XP(/,//,[ ],|,evars,≠), that is, inequality tests on variables and attribute values are allowed, remains in Π_2^P. Surprisingly, the further addition of ∗ to this fragment makes the containment problem undecidable. The results about XPath containment for fragments with variables are indicated in Table 3

Further related work. In [37], Wood shows that containment of XP(/,//,[ ],∗) in the presence of DTDs is decidable. 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 [2]. They also consider sound and complete axiom systems and normal forms for some of these fragments. Hidders, and Benedikt, Fan, and Geerts considered the complexity of satisfiability of XPath expressions [18, 1, 15]. The complexity of XPath evaluation has been studied by Gottlob, Koch, and Pichler in [16, 17], while its expressive power has been addressed by Marx in [26] and [25]. This article is based on [30].

Organization. This article is organized as follows. In Section 2, we define DTDs and the basic XPath fragments. We also introduce the necessary machinery w.r.t. unranked tree automata.

/	//	[]	*	$\mid$	xvars	evars	$\neq$	complexity	Reference
+		+			+		+	CONP-hard	(5.2)
+	+				+		+	CONP-hard	(5.2)
+	+	+	+	+	+		+	PSPACE-complete	(5.1,5.2)
+	+		+	+	+		+	PSPACE-complete	(5.1,5.2)
+	+	+				+		CONP-complete	[14]
+	+					+		CONP-complete	[14], (5.3)
+	+	+	+	+		+		Π_2^p-complete	[14]
+				+		+		Π_2^p-complete	[14], (5.3)
+		+				+	+	Π_2^p-complete	(5.3,5.4)
+	+	+		+		+	+	Π_2^p-complete	(5.3,5.4)
+	+	+	+	+		+	+	undecidable	(5.6)

Table 3: The complexity of containment in the presence of variables. Note that xvars and evars refer to the original XPath semantics and to existential semantics, respectively.

In Section 3, 4, and 5 we consider disjunction, DTDs, and variables, respectively. We conclude in Section 6.

2. Preliminaries

In this section, we define the tree abstraction of XML documents, DTDs and the fragments of XPath that we consider.

2.1. **XML-trees.** For the rest of this paper, we fix an infinite set Σ of labels and an infinite set $\mathbf{D}$ of data values. Only in Section 3.2, we consider XPath expressions over a finite alphabet. The set A is always a finite set of attributes. An XML document is faithfully modeled by a finite unranked tree with labels from Σ in which the attributes of the nodes have $\mathbf{D}$-values and in which the children of each node are ordered.

It is common to model the underlying tree as a *tree domain*. To this end, the edges connecting a node with its children are numbered from 1 to n, according to the ordering of its children. Each path from the root to a node then corresponds to a sequence of numbers. Finally, each node is identified with this sequence. In particular, the root, which corresponds to the document node [9], is represented by the empty string denoted by ε.

More formally, a **tree domain** D is a finite subset of $\mathbb{N}^*$ with the following closure properties:

- If $v \cdot i \in D$, where $v \in \mathbb{N}^*$ and $i \in \mathbb{N}$, then $v \in D$.
- If $i > 1$ and $v \cdot i \in D$, then also $v \cdot (i-1) \in D$.

We call the elements of D **vertices**. A vertex vi with $i \in \mathbb{N}$ is a **child** of a vertex v. Conversely, v is called the **parent** of vi. A vertex vu with $u \in \mathbb{N}^+$ is a **descendant** of v. We also say that v is an **ancestor** of vu.

Definition 2.1. An **XML-tree** (tree for short) is a triple $t = (\mathrm{dom}(t), \mathrm{lab}_t, \lambda_t)$, where $\mathrm{dom}(t)$ is a tree domain over $\mathbb{N}$, and $\mathrm{lab}_t : \mathrm{dom}(t) \to \Sigma$ and, for each $a \in A$, $\lambda_t^a : \mathrm{dom}(t) \to \mathbf{D}$ are partial functions. Intuitively, $\mathrm{lab}_t(v)$ is the label of v, while $\lambda_t^a(v)$ is the value of v's a-attribute, if it has one.

Of course, in real XML documents there can be vertices with mixed content, but these can easily be modeled by using auxiliary intermediate nodes as explained in [3]. We consider attributes only in Section 5.

For a vertex $v \in \mathrm{dom}(t)$, we denote by t_v the **sub-tree** of $\mathrm{dom}(t)$ rooted at v. As a tree domain in itself, this is the set $\{w \mid vw \in \mathrm{dom}(t)\}$.

2.2. **DTDs.** We formalize Document Type Definitions (DTDs) as context-free grammars with regular expressions on the right-hand side of rules. As usual, we denote by $L(r)$ the language defined by the regular expression r.

Definition 2.2. 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 match d.

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

2.3. **XPath.** We next define the core fragment of XPath that we will consider in Sections 3 and 4. In our definition we follow Marx [23]. In Section 5, we consider a larger fragment which allows the use of attribute values.

Definition 2.3. An **XPath expression** is generated by the following grammar:

$$\begin{array}{rcl} \text{lpath} & ::= & \text{lstep} \mid \text{lpath '/' lpath} \mid \text{lpath '|' lpath} \\ \text{lstep} & ::= & \text{axis '::' node-test ('['fexpr']')}^* \\ \text{axis} & ::= & \texttt{self} \mid \texttt{child} \mid \texttt{descendant} \\ \text{fexpr} & ::= & \text{lpath} \mid \text{lpath 'or' lpath} \end{array}$$

Here, lpath is the start symbol which is short for location path; node-test is either a label or the wild-card '$*$'.

We write $|p|$ for the **size** of an XPath expression, which is the total number of occurrences in p of axes `child` and `descendant` (including those in filter expressions).

We use $\bigcup$ to denote a big disjunction of expressions.

Note that, as we only consider expressions which navigate downwards in the tree, we do not allow *absolute location paths*, i.e., paths requiring to be evaluated from the root. For convenience, we further assume that location steps with the `self`-axis only occur at the top-level. This is no loss of generality, as one can always translate a location path of the form

$$\text{axis} :: \sigma[e_1] \cdots [e_k]/\texttt{self} :: \sigma'[f_1] \cdots [f_\ell]$$

into

$$\text{axis} :: \gamma[e_1] \cdots [e_k][f_1] \cdots [f_\ell]$$

where

$$\gamma = \begin{cases} \sigma & \text{if } \sigma' = *\text{; and,} \\ \sigma' & \text{if } \sigma = \sigma' \text{ or } \sigma = *. \end{cases}$$

Of course, when $\sigma \neq \sigma'$ are two labels in the above expression, then it is unsatisfiable.

For notational brevity, we often use abbreviated syntax for XPath expressions [9]. Thus, instead of

$$\texttt{child}{::}\ a\ [\texttt{child}{::}\ b][\texttt{descendant}{::}\ \mathrm{e}]/\texttt{descendant}{::}\ c,$$

we simply write $a[b][.//e]//c$. Note that the sub-expression $.//e$ is the abbreviated notation for `descendant-or-self`:: $*$ / `child`:: e and thus accounts for `descendant`:: e.

For the definition of the semantics of XPath expressions we again basically follow [23].

Definition 2.4. For each tree t, each location path p induces a binary relation $\llbracket p \rrbracket_t$ which is inductively defined as follows:

- $\llbracket a :: n[e_1] \cdots [e_k] \rrbracket_t$ is the set of all pairs (u, v) for which all the following conditions hold:
  - if a is `child` then v is a child of u;
  - if a is `descendant` then v is a descendant of u;
  - if a is `self` then $v = u$;
  - if n is a label then $\mathrm{lab}_t(v) = n$, otherwise n is $*$ and the label of v can be arbitrary; and,
  - $E_t(v, e_i)$ is true, for each $i \leq k$. Here, E_t is defined as follows:
    - $E_t(v, p)$ is true for a vertex v and a location path p if and only if there is a vertex w such that $(v, w) \in \llbracket p \rrbracket_t$.
    - $E_t(v, e_1 \text{ or } e_2)$ is true for a vertex v if and only if $E_t(v, e_1)$ or $E_t(v, e_2)$ is true.
- $\llbracket p/q \rrbracket_t = \llbracket p \rrbracket_t \circ \llbracket q \rrbracket_t$ (where $\circ$ denotes the composition of binary relations); and,
- $\llbracket p \mid q \rrbracket_t = \llbracket p \rrbracket_t \cup \llbracket q \rrbracket_t$.

So, the semantics definition associates with every tree t and every expression p a binary relation. When the *context vertex*, i.e., the first vertex in pairs, is fixed to be the root then every expression defines the set $\{v \mid (\varepsilon, v) \in \llbracket p \rrbracket_t\}$. Recall that ε denotes the root of a tree. We say a tree t **matches** an expression p (written: $t \models p$) if there is some vertex v in t such that $(\varepsilon, v) \in \llbracket p \rrbracket_t$. In the latter case we interpret an expression p as a **Boolean query**.

Symbol	Meaning
/	child axis is allowed
//	descendant axis is allowed
[ ]	filter expressions are allowed
|	disjunction ('or' and |) is allowed
$*$	wild-cards are allowed

Table 4: Symbols used in the notation for XPath fragments.

We denote sub-fragments of the above defined XPath fragment using the abbreviated syntax. We use the notations explained in Table 4. If filter expressions are not allowed, location steps are only of the form axis :: node-test. Disjunction is allowed in location paths and in filter expressions. If wild-cards are not allowed, every node-test has to be a label.

We denote XPath fragments by XP(...) where inside the brackets the allowed features are listed. For instance, we write XP(/,[ ],|) for the fragment, where the wild-card is not allowed and the descendant axis can not be used.

In some proofs, we view expressions p from XP(/,//,[ ],$*$) as **tree patterns** as described by Miklau and Suciu [27]. For example, the expression $a/b//c[d][*/e]$ corresponds to the tree pattern in Figure 1. Single edge and double edge correspond to the child and descendant axis, respectively. We denote the tree pattern associated with an expression p by $\tau(p)$.

From this point of view, $t \models p$ if and only if there is a homomorphism h from $\tau(p)$ to t, i.e., h maps the nodes of $h(p)$ to the nodes of t such that (1) $h(v)$ has the same label as v unless v carries a wild-card, (2) $h(v)$ is a child (descendant) of $h(u)$ if and only if v is a child (descendant) of u. So, h respects labels, child and descendant, and does not care about $*$.

Every tree pattern has one selecting node: all pairs (u, v) of nodes of the input tree are selected, for which the root of the tree pattern can be mapped to u and the selecting node to v. In Figure 1, the selecting node is labeled by x.

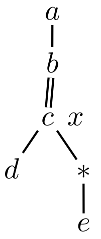

Figure 1: The tree pattern corresponding to $a/b//c[d][*/e]$

We will frequently make use of the fact that every expression p from XP(/,//,[ ],*,|) can be written in **disjunctive normal form**, i.e., in the form $p_1 \mid \cdots \mid p_n$, where each p_i is an expression from XP(/,//,[ ],*). It should be noted that n can be exponential in $|p|$.

2.4. **Containment.** Corresponding to the three different interpretations of an XPath expression as a binary, unary or Boolean query, there are three different notions of query containment.

Definition 2.5. For two XPath expressions p and q,

- p is **contained as a binary query** in q, denoted by $p \subseteq_2 q$, if $[\![p]\!]_t \subseteq [\![q]\!]_t$, for every tree t;
- p is **contained as a unary query** in q, denoted by $p \subseteq_1 q$ if, for each tree t and each vertex v of t, $(\varepsilon, v) \in [\![p]\!]_t$ implies $(\varepsilon, v) \in [\![q]\!]_t$; and
- p is **contained as a Boolean query** in q, denoted $p \subseteq q$, if, for every tree t, $t \models p$ implies $t \models q$.

Definition 2.6. XContainment is the algorithmic problem to decide for two XPath expressions p and q, whether $p \subseteq q$.

For a DTD d and two XPath expressions p and q, by $p \subseteq_d q$, we denote that $t \models p$ implies $t \models q$, for all trees t that match d.

Definition 2.7. XDContainment is the algorithmic problem to decide for two XPath expressions p and q and a DTD, whether $p \subseteq_d q$.

The restriction to Boolean containment is justified: it is shown in [27] that for XP(/,//,[ ],*) the complexity of deciding binary (or k-ary, for any k) containment is the same as the complexity of deciding Boolean containment. For binary queries, this result can be generalized to all fragments we consider in this article. More precisely: all complexity results stated in this article hold also for unary and binary containment. First of all, binary and unary containment are computationally equivalent in our framework as from a node v expressions can only navigate inside the subtree induced by v. Furthermore, to get from unary to Boolean containment, it is easy to see that the result in [27] only requires the child axes, i.e., only one node is added as a child of the selecting node of the tree pattern. It is straightforward to get the same statement also in the presence of a DTD. For the fragment XP(//,|) a similar approach works: for the label a of the selecting node a new label a' is introduced and every reference of a not-selecting node to a is replaced by $a|a'$. The lower bound we prove for XP(//,[ ]) also goes through for the binary and unary case.

2.5. **Unranked tree automata.** We recall the definition of non-deterministic tree automata over unranked trees from [4]. These are used in the proofs of Theorem 4.1 and Theorem 4.4. We refer the unfamiliar reader to [29] for a gentle introduction. The alternating automata of Section 3.2 operate over ranked trees.

Definition 2.8. A **nondeterministic tree automaton (NTA)** is a tuple $A = (Q, \Delta, \delta, F)$, where Q is a finite set of states, Δ is a finite alphabet, $F \subseteq Q$ is the set of final states, and δ is a function $\delta : Q \times \Delta \to 2^{(Q^*)}$ such that $\delta(q, a)$ is a regular string language over Q for every $a \in \Delta$ and $q \in Q$.

A *run* of A on a tree t is a labeling $\lambda : \mathrm{dom}(t) \to Q$ such that for every $v \in \mathrm{dom}(t)$ with n children we have that $\lambda(v1) \cdots \lambda(vn) \in \delta(\lambda(v), \mathrm{lab}^t(v))$. Note that when v has no children, the criterion reduces to $\varepsilon \in \delta(\lambda(v), \mathrm{lab}^t(v))$. A run is *accepting* iff the root is labeled with an accepting state, that is, $\lambda(\varepsilon) \in F$. A tree is *accepted* if there is an accepting run. The set of all accepted trees is denoted by $L(A)$.

The regular languages encoding the transition function of an NTA are represented by NFAs. The size of an NTA is then $|Q| + |\Delta|$ plus the sizes of the NFAs.

A **deterministic tree automaton (DTA)** is an NTA where $\delta(q, a) \cap \delta(q', a) = \emptyset$ for all $a \in \Delta$ and $q \neq q' \in Q$. The transition function of a DTA is represented by DFAs.

The following is well known.

Lemma 2.9. (1) Deciding whether, for a given NTA A, $L(A) = \emptyset$ is in PTIME. [21, Theorem 19]
(2) Testing whether, for given DTAs A and B, $L(A) \subseteq L(B)$ is in PTIME. [22, Theorem 3]
(3) Given a DTD d, a DTA A_d such that $L(d) = L(A_d)$ can be constructed in exponential time. (essentially, [4])

3. Containment in the presence of disjunction

Miklau and Suciu showed that XContainment for XP(/,//,[],*) is coNP-complete [27]. In this section, we consider the addition of disjunction to this fragment and show that XContainment remains in coNP. The problem remains hard for coNP even if only the child or only the descendant axis is allowed together with disjunction. Miklau and Suciu already mention these results but do not provide full proofs [27] (with exception of the lower bound for XP(/,|)). Therefore, we decided to include the proofs in this paper.

We were surprised by the fact that the coNP upper bound strongly depends on the fact that the alphabet Σ is infinite. Let XFContainment be the containment problem, where additionally a finite set of labels is given as input and the containment only has to hold for documents with labels from this set. In Theorem 3.3, we show that even for XP(/,//,|) the problem XFContainment is hard for PSPACE.

3.1. Unrestricted alphabet.

Theorem 3.1. XContainment for XP$(/, //, [\,], *, |)$ expressions is in coNP.

Proof. We develop a criterion which allows to check in NP whether, for given expressions p and q, $p \not\subseteq q$. Let p and q be fixed and let $p_1 | \ldots | p_l$ and $q_1 | \ldots | q_{l'}$ be the disjunctive normal forms (DNFs) of p and q, respectively. Hence, each p_i and q_j is an expression from XP$(/, //, [\,], *)$. Let n and m denote the maximum number of nodes in an expression p_i and q_j, respectively. Let $T(n, m)$ be the set of trees with at most $2n(m + 2)$ nodes that are labeled with symbols that occur in p and with the new symbol # not occurring in p nor in q. We prove the following claim:

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

Clearly "$\Leftarrow$" holds. For the other direction we assume that there is a tree s matching p but not q. Then s has to match one of the p_i. Hence, there is a homomorphism f 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 f. We delete all nodes in s that are neither in V nor an ancestor of a node in V. The resulting tree, t_1, has at most as many leaves as p_i. We replace the labels of those nodes of t_1 which are not in V by # and obtain t_2. Let V' be the set of branching nodes of t_2, i.e., those nodes that have more than one child. The set V' contains at most n vertices. Let a *pure path* of t_2 be a path without nodes from $V \cup V'$. In particular, the nodes of a pure path are all labeled with #. We get t by replacing in t_2 each maximal pure path with $> m + 1$ inner nodes by a path with $m + 1$ #-labeled inner nodes. We refer to the nodes of t which are inserted in this last step as *special nodes*. Clearly, there is a one-to-one correspondence between non-special nodes in t_2 and t. For a non-special node u in t, the corresponding node in t_2 is denoted by $\tilde{u}$.

It is easy to see that $t \in T(n, m)$, 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 that h can be modified to obtain a homomorphism from q_j to s which leads to the desired contradiction. We first define a homomorphism h_2 from q_j to t_2 as follows. Whenever $h(v)$ is a non-special node then $h_2(v) = \widetilde{h(v)}$.

Let $v_1, \ldots, v_k$ be nodes of q_j such that $h(v_1), \ldots, h(v_k)$ are special nodes which lie on some path of t_2 which consists entirely of special nodes, 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 $h(v_{i+1})$ is not a child of $h(v_i)$ or $h(v_1)$ is not the first node of the path or $h(v_k)$ is not the last node of the path.

In either case, we can define h_2 for nodes from $\{v_1, \ldots, v_k\}$ such that the child and descendant relations are respected. In this way, we get a homomorphism h_2 from q_j to t_2.

Clearly, all nodes of q_j which are not mapped to nodes in V must be labeled with a $*$. Thus, h_2 also defines a homomorphism from q_j to t_1 and to s, the desired contradiction. This completes the proof of the claim.

It remains to show how the criterion of the above claim can be used for an NP-algorithm that checks whether $p \not\subseteq q$. The algorithm simply guesses an expression p_i from the DNF of p (by non-deterministically 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 [17]. □

Theorem 3.2. (a) XContainment for XP(/, |) is coNP-hard.
(b) XContainment for XP(//, |) is coNP-hard.

Proof. (a) The hardness proof is the same proof that shows that containment of regular expressions is CONP-hard [20]. 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 [20]. 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 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.

(b) 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 $p \subseteq \bar{q} \Leftrightarrow \varphi$ is valid. Suppose $p \subseteq \bar{q}$, then in particular $\bar{q}$ matches every 0-1-string of length n, hence, φ is valid. To prove the converse direction, suppose φ is valid. If p matches a branch in a tree then there are in particular n positions with 0 or 1. The i-th 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.

□

3.2. **Finite alphabet.** As mentioned above, when the alphabet is finite, and given as part of the input, containment becomes much harder. In the rest of this section, Σ is therefore a *finite* alphabet.

Theorem 3.3. XFCONTAINMENT for XP$(/, //, |)$ is PSPACE-hard.

Proof. We make use of a reduction from CORRIDOR TILING which is known to be hard for PSPACE [8]. 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; $\bar{b} = (b_1, \ldots, b_n)$ and $\bar{t} = (t_1, \ldots, t_n)$ are n-tuples of tiles; and, n is a natural number in unary notation. The question is whether there exists a number m and a valid tiling of a board with n columns and m rows. Here, a tiling is valid if the following conditions are fulfilled:

- The bottom row is tiled with $\bar{b}$.
- The top row is tiled with $\bar{t}$.
- For each horizontal pair (x, y) of tiles, $(x, y) \in H$.
- For each vertical pair (x, y) of tiles (y above x), $(x, y) \in V$.

We use a string representation of the board where every row is delimited by # and the last symbol is \$. The expression q selects all strings that do not encode a tiling. As Σ we take $D \cup \{\#, \$\}$. For $S = \{c_1, \ldots, c_n\} \subseteq \Sigma$, we abbreviate the expression $(c_1 \mid \cdots \mid c_n)$ by S. For an expression r, r^i denotes $r/\cdots/r$ with i occurrences of r. The expression p is $.//\$$ assuring that the string contains the symbol \$. The expression q is the disjunction of the following expressions.

- some row has the wrong format:
  - some inner row has too few tiles: $\bigcup_{i=0}^{n-1} .//\#D^i\#$
  - the first row has too few tiles: $\bigcup_{i=0}^{n-1} D^i/\#$
  - the last row has not enough tiles: $\bigcup_{i=0}^{n-1} .//\#D^i/\$$

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

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. □

Actually, in the proof of Theorem 3.3, the restriction to a finite alphabet is only used to express that a certain element name in the XML document does not occur in a certain set. Therefore, if we extended the formalism with an operator $*_{\notin S}$ for a finite set S, expressing that any symbol but one from S is allowed, then containment would also be hard for PSPACE.

For the upper bound we need the notion of alternating tree automata [33] which is defined next. These automata operate on trees where every node has rank at most k (for some fixed k). That is, every node has at most k children.

Definition 3.4. An **alternating tree automaton (ATA)** is a tuple $A = (k, Q, \Sigma, q_0, \delta)$ where $k > 0$, Q is a finite set of states, Σ is the finite alphabet, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Sigma \times \{0, 1, \ldots, k\} \to \mathbf{B}^+(\{0, 1, \ldots, k\} \times Q)$ is the transition function. Here, $\mathbf{B}^+(\{1, \ldots, k\} \times Q)$ 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 $[\varepsilon, q_0]$, where ε is the root of t and, for every node u of s (including leaf nodes), the following local consistency condition holds. Let u be labeled with $[v, q]$ with n children labeled $[v_1, q_1], \ldots, [v_n, q_n]$. Then it must hold that

- each v_i is a child of v or v itself; and,
- $\delta(q, \mathrm{lab}_t(v), m)$ is satisfied by the truth assignment ρ, where $m \leq k$ is the number of children of v in t, and $\rho((\ell, q'))$ is true if for some i, $q_i = q'$ and v_i is the ℓ-th child of v (where we view v itself as the 0-th child).

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

Note that ATAs as we defined them do not have final states. These are encoded by transitions of the form $\delta(q, \sigma, 0) = \texttt{true}$.

Theorem 3.5. XFCONTAINMENT for XP$(/, //, [\,], *, |)$ is in PSPACE.

Proof. We show first that $p \not\subseteq q$ implies that there is a counter example tree with small degree and only a few branching nodes. More precisely, we call a tree **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. For an XP$(/, //, [\,], *, |)$ -expression p let $f(p)$ be the maximum number of filter expressions in any disjunct of the DNF of p. We claim that $p \subseteq q$ if and only if $t \models p$ implies $t \models q$ 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 $q_1|\cdots|q_m$, respectively. Thus, for some i, $t \models p_i$, but $t \not\models q_j$, for all j. Let h be a homomorphism from p_i to t and let s be the tree obtained from t by deleting all nodes that are neither in the image of h nor ancestors of such nodes. Clearly, s is $f(p)$-bounded, $s \models p_i$ and $s \not\models q_j$ for all j (otherwise, $t \models q_j$).

Next, we show that for every XP$(/, //, [\,], *, |)$ expression p there is an ATA A_p such that for every $f(p)$-bounded tree t, $t \models p$ iff A_p accepts t. Moreover, A_p can be constructed in LOGSPACE.

To this end, let p be an XP$(/, //, [\,], *, |)$ expression. As the alphabet is finite and fixed, we can replace every $*$ with a disjunction of the alphabet symbols. Hence, we assume p does not contain $*$. Let $k = f(p)$. We define $A_p = (k, Q, \Sigma, q_0, \delta)$ where $q_0 = p$ and Q is the set of sub-expressions of p, all filter expressions of p and all node tests of p. Intuitively, a pair $[v, q]$ in an accepting run of A_p on a tree t means that q holds in the sub-tree of t rooted at v. For all $m \leq k$, the transition function is inductively defined as follows:

- $\delta(\texttt{self} :: \sigma[e_1] \cdots [e_\ell]/p', \sigma, m) = (0, p') \wedge \bigwedge_{i=1}^{\ell}(0, e_i)$;
- $\delta(\texttt{child} :: \sigma[e_1] \cdots [e_\ell]/p', \tau, m) = \bigvee_{j=1}^{m}[(j, \sigma) \wedge (j, p') \wedge \bigwedge_{i=1}^{\ell}(j, e_i)]$;
- $\delta(\texttt{descendant} :: \sigma[e_1] \cdots [e_\ell]/p', \tau, m) =$
$$\bigvee_{j=1}^{m}(j, [\![\texttt{descendant} :: \sigma[e_1] \cdots [e_\ell]/p']\!]_t \vee ((j, \sigma) \wedge (j, p') \wedge \bigwedge_{i=1}^{\ell}(j, e_i));$$
- $\delta(\sigma, \sigma, m) = \texttt{true}$;
- $\delta(p \mid q, \sigma, m) = (0, p) \vee (0, q)$; and,
- $\delta(e_1 \mid e_2, \sigma, m) = (0, e_1) \vee (0, e_2)$.

The combinations $\delta(p, \sigma, m)$ that are not mentioned are $\texttt{false}$. For location paths of length 1, i.e., if in one of the first three transitions there is no p', the atoms $(0, p')$ or (j, p') are removed, respectively. It is straightforward to prove by a nested induction on the structure of the tree and the expression that the pairs $[v, q]$ have the intended meaning. Therefore, a tree t has an accepting run of A_p if and only if $t \models p$.

Thus, to decide $p \subseteq q$ it is sufficient to test whether every $f(p)$-bounded tree accepted by A_p is also accepted by A_q. Note that $f(p)$-bounded trees can be easily encoded by strings. We say that a node is a fork if it has more than one child. As there are at most $f(p)$ forks, every tree consists of at most $k := f(p) \times f(p)$ unary paths that are joined at the at most $f(p)$ forks. To every path we associate its lower fork (or none if there is no such fork). Next, we assign a unique number to each path such that higher paths get lower numbers. Let s_i be the concatenation of the labels on path i and let $i_1, \ldots, i_\ell$ be the paths rooted at the fork below branch i. Let $a_i := s_i\, i_1 \cdots i_\ell \#$. Every $f(p)$-bounded tree t can then be encoded by the string $a_1 \cdots a_m$. Let A'_p and A'_q be the alternating string automata that simulate A_p and A_q on the string representations of bounded trees. Basically, whenever the automaton reaches the end of (the encoding of) a path, the numbers $i_1, \ldots, i_\ell$ indicate the positions of the children paths and it can reach a path i_j by skipping everything before its occurrence. This is possible as all paths are ordered and higher numbered paths occur to the right. Finally, let A be the automaton that checks whether the input string is a valid encoding of a bounded tree. The problem then reduces to testing whether $A \cap A'_p \cap \neg A'_q$ is empty. The latter can be done in PSPACE [7]. We arrive at the desired result. □

4. Containment in the presence of DTDs

In this section we study the XDContainment problem, i.e., the containment problem relative to a DTD. Deutsch and Tannen [14] show a PSPACE lower bound for XDContainment for XP(/,//,[ ],∗,|). Marx [24] gives an EXPTIME upper bound for a much larger fragment, including all axes and negation in filter expressions.

We show here that XDContainment for XP(/,//,[ ],∗,|) problem is actually EXPTIME-complete. We also exhibit a simple fragment with tractable XDContainment and 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 ∗. The results of this section are summarized in Table 2.

We illustrate by a simple example that the presence of a DTD can complicate matters. Consider the DTD

$$\begin{aligned} a &\rightarrow ab \mid \varepsilon \\ b &\rightarrow c \\ c &\rightarrow \varepsilon \end{aligned}$$

and the expressions $p = a/a$ and $q = .//b/c$. Although p and q are seemingly unrelated and, in particular, it is not the case that every path matching p also matches q, it holds that each tree which respects the DTD and matches p also matches q.

We remark that it can be tested in polynomial time whether, for a DTD d and a symbol a, there exists a tree $t \in L(d)$ with a vertex labeled a. Therefore, we assume in the following that each input DTD contains only *useful* symbols. In particular, for each d and each symbol $a \in \Sigma_d$, there is a tree valid with respect to (d, a, Σ_d).

4.1. **A tractable fragment.** We start with a fragment in P.

Theorem 4.1. XDContainment of XP$(DTD, /, //)$-expressions is in P.

Proof. Let d be a DTD and p, q be expressions of XP$(DTD, /, //)$.

We first show how to construct a non-deterministic top-down automaton A_p which checks that, for a tree t, $t \models p$ holds. To this end, let $p = p_1//p_2// \cdots //p_k$, where in each p_i only the child axis is used. For each i, let p_i contain i_ℓ child-axis location steps.

Intuitively, A_p guesses a path in t which matches p and, for each node v on this path it maintains the lexicographically maximal (i, j) such that the path from the root to v matches the expression $p_1// \cdots //p_{i-1}//p_i^j$, where p_i^j consists of the first j location steps of p_i (along the child-axis).

It should be stressed here, that A_p needs non-determinism only to guess the path. The computation of the pairs (i, j) is completely deterministic and similar to the case of the standard string pattern matching automata [32]. The automaton, A_p enters an accepting state on the leaf u of the distinguished (guessed) path if and only if the computed pair for u is (k, i_k). On all other leaves it takes an accepting state in any case.

An automaton A_q which accepts all trees that do *not* match q can be constructed along the same lines. It computes a pair (i, j) with the same intended meaning as above, for every node v of the tree and takes an accepting state at all leaves that have *not* reached (k, i_k). This automaton is actually deterministic.

By combining A_p with A_q and the canonical non-deterministic automaton A_d which tests $t \models d$, we obtain an automaton which accepts all counterexamples to $p \subseteq_d q$. As this automaton is of polynomial size in p, q, d and testing emptiness of non-deterministic tree automata is in PTIME (Lemma 2.9(1)), we obtain the stated upper bound. □

It should be mentioned that in [30] we claimed that XDCONTAINMENT of XP(DTD, /, //, $*$)-expressions is in P. Unfortunately, we were not able to extend the proof sketch given there into a complete proof. In fact, we conjecture that this problem is CONP-hard.

4.2. Fragments in CONP.

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 4.2. XDCONTAINMENT for XP(DTD, /, [ ]) is in CONP.

Proof. The obvious idea is to guess a tree t which matches P but not q. A complication arises from the fact that the smallest such tree t might be of exponential size due to the constraints from d. Thus, we give a non-deterministic algorithm `CheckPnotq` (d, a, P, q) which checks, given a DTD d, a non-terminal a of d, a set $P = \{p_1, \dots, p_n\}$ of XP(/, [ ])-expressions, and an XP(/, [ ])-expression q, whether there is a tree t with root symbol a which conforms to d, matches all expressions in P but does not match q. But it does not explicitly construct such a tree. Clearly, invoking this algorithm with d, q, $P = \{p\}$ and a as the start symbol of d checks whether $p \not\subseteq_d q$. We note that we allow a set P of expressions as input for `CheckPnotq` because the algorithm uses such sets for recursive calls.

Algorithm `CheckPnotq` makes 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 expressions 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.

We assume in the following that all labels of P and q occur in d and that d only contains symbols from which a tree can be derived. By $s(p)$ we denote the root symbol of an expression p, i.e., the label of the root of $\tau(p)$. A level 1 sub-expression of an expression p is an expression corresponding to a child of the root in $\tau(p)$. Let l denote the overall number of depth-1-nodes in expressions of P.

The algorithms are given in Figure 2. The algorithms follow a top-down approach and work recursively. The correctness can be shown by induction on the number of recursive calls. `Checknotq` and `CheckP` are quite straightforward.

An important point is that in `CheckP` it is sufficient to consider strings u of length at most $(|d| + 1)(l + 2)$. It can be shown by a simple pumping argument that if a tree matching P has a level with more children then there is a sub-sequence of these children which is not in the image of any mapping and can be removed without leaving $d(a)$.

`CheckPnotq` is basically a combination of `Checknotq` and `CheckP`. It has to check that there is a path in q which does not match any path in the counter-example tree. Step 6 is needed to verify that this also holds in those parts of the tree which are not needed to fulfil P.

It remains to show that this (non-deterministic) algorithm works in polynomial time. This follows directly from the fact that for each node in $\tau(p)$ and each node v in $\tau(q)$ there is at most one recursive call of `CheckPnotq` in which v is the root of some expression in P (or q') .

□

```
Checknotq$(d, a, q)$
(Returns TRUE if there exists a tree with root $a$ which does not match $q$)
1. If $s(q) \neq a$ return TRUE.
2. If $\tau(q)$ has only one node return FALSE.
3. Guess a string $u \in d(a)$ of length $\leq |d|$ and a level 1 sub-expression $q'$ of $q$.
4. If $b := s(q')$ does not occur in $u$ return TRUE.
5. Return Checknotq$(d, b, q')$.
```

```
CheckP$(d, a, P = \{p_1, \ldots, p_n\})$
(Returns TRUE if there exists a tree with root $a$ matching all $p_i$)
1. If some expression in $P$ does not have the root symbol $a$ return FALSE.
2. Guess a string $u \in d(a)$ of length $\leq (|d| + 1)(l + 2)$.
3. For each $i \in \{1, \ldots, n\}$, guess a mapping $f_i$ from the level 1 sub-expressions of
   $p_i$ to the positions of $u$.
4. For each position $j$ of $u$, which is in the image of at least one of the mappings
   $f_i$
   (a) Let $P'$ be the set of level 1 sub-expressions $p$ with $f_i(p) = j$.
   (b) Call CheckP$(d, u_j, P')$.
5. Return TRUE iff all the recursive calls return TRUE.
```

```
CheckPnotq$(d, a, P = \{p_1, \ldots, p_n\}, q)$
(Returns TRUE if there exists a tree with root $a$ matching all $p_i$ but not $q$)
1. If some expression in $P$ does not have the root symbol $a$ THEN return FALSE.
2. If $s(q) \neq a$ return CheckP$(d, a, P)$.
3. If $\tau(q)$ has only one node return FALSE.
4. Guess a level 1 sub-expression $q'$ of $q$.
5. Guess a string $u \in d(a)$ of length $\leq (|d| + 1)(l + 2)$.
6. If $b := s(q')$ occurs in $u$ call CheckPnotq$(d, b, \emptyset, q')$.
7. For each $i \in \{1, \ldots, n\}$, guess a mapping $f_i$ from the level 1 sub-expressions of
   $p_i$ to the positions of $u$.
8. For each position $j$ of $u$, which is in the image of at least one of the mappings
   $f_i$
   (a) Let $P'$ be the set of level 1 sub-expressions $p$ with $f_i(p) = j$.
   (b) Call CheckPnotq$(d, u_j, P', q')$.
9. Return TRUE iff all the recursive calls return TRUE.
```

Figure 2: Algorithms `Checknotq`, `CheckP` and `CheckPnotq` used in the proof of Theorem 4.2.

The next theorem follows directly from [35].

Theorem 4.3. (a) XDContainment for XP$(/, [\,])$ is coNP-hard.
(b) XDContainment for XP$(//, [\,])$ is coNP-hard.

Proof. In [35], the following problem is shown to be coNP-hard.

SIBLING CONSTRAINT IMPLICATION (SC IMP):
Given: Regular expression r over alphabet Σ, a set $S \subseteq \Sigma$, and $a \in \Sigma$.

Question: Does every string $w \in L(r)$ that contains all the symbols in S also contain the symbol a (denoted $r \models S \to a$)?

We reduce SC IMP to $\mathrm{XP}(/, [\,])$ and $\mathrm{XP}(//, [\,])$. Thereto, assume given r, $S = \{s_1, \dots, s_k\}$, and a. Construct the DTD d consisting of the sole rule start $\to r$, where start $\notin \Sigma$, then $\mathrm{start}[s_1] \cdots [s_k] \subseteq_d \mathrm{start}[a]$ iff $\mathrm{start}[.//s_1] \cdots [.//s_k] \subseteq_d \mathrm{start}[.//a]$ iff $r \models S \to a$. □

4.3. **Fragments in EXPTIME.** When both the child and descendant axes are allowed, then adding filter expressions and wild-card or disjunction raises the complexity of XDCONTAINMENT to EXPTIME.

Although in [24] it is shown that XDCONTAINMENT is in EXPTIME even for full navigational XPath by a reduction to propositional dynamic logic, we give here a simpler proof of the result for our downwards navigating fragment.

First, we introduce two concepts. Let p be an XPath-expression. An expression is a **sub-expression** of an expression p if it is generated by an lpath node in the derivation tree of p according to the grammar in Definition 2.3.

The **`self`-closure** of p, denoted by $\mathtt{self}(p)$, is inductively defined as follows: for a location step $p_1 = \mathrm{axis} :: \sigma[e_1] \cdots [e_k]$ its `self`-closure, is $\mathtt{self} :: \sigma[e_1] \cdots [e_k]$. For expressions p_1/p_2 and $p_1 \mid p_2$, their `self`-closure is $\mathtt{self}(p_1)/p_2$ and $\mathtt{self}(p_1) \mid \mathtt{self}(p_2)$, respectively.

Theorem 4.4. XDCONTAINMENT for $\mathrm{XP}(/, //, [\,], *, |)$ is in EXPTIME.

Proof. We provide a translation to containment of unranked deterministic tree automata whose size is exponential in the input. By Lemma 2.9(2), the latter is in EXPTIME.

We first show that for each $\mathrm{XP}(/, //, [\,], *, |)$-expression p, one can construct in exponential time an exponential size deterministic tree automaton $A_p = (Q, \Sigma_d, \delta, F)$ such that A_p accepts a tree t if and only if $t \models p$. Here, Σ_d is the finite alphabet associated to the given DTD d. The states of A_p are pairs (S, D) where S and D are sets of sub-expressions of p or the `self`-closure of sub-expressions of p.

The intended meaning of the states is as follows. If $p_1 \in S$ at some vertex v of a tree t then $t_v \models p_1$. If $p_1 \in D$ then there is some node w below v in t such that $t_w \models p_1$. So, S describes all expressions that hold at the current node, while D describes all expressions that hold at descendants of the current node.

Set $F = \{(S, D) \mid p \in S\}$. It remains to define the transition function for each $\delta((S, D), a)$. Recall that the corresponding DFA operates on strings of the form: $(S_1, D_1) \cdots (S_\ell, D_\ell)$. Then S should contain exactly the XPath-expressions generated by the following rules.

- $\mathtt{child} :: \sigma[e_1] \cdots [e_k]/p' \in S$ iff there is an $i \le \ell$ such that $\mathtt{self} :: \sigma[e_1] \cdots [e_k]/p' \in S_i$;
- $\mathtt{descendant} :: \sigma[e_1] \cdots [e_k]/p' \in S$ iff there is an $i \le \ell$ such that $\mathtt{self} :: \sigma[e_1] \cdots [e_k]/p' \in S_i \cup D_i$;
- $\mathtt{self} :: \sigma[e_1] \cdots [e_k]/p' \in S$ iff $\sigma = a$ or $\sigma = *$, $p' \in S$ and $e_i \in S$ for $i = 1, \dots, k$;
- $p_1 \mid p_2 \in S$ iff $p_1 \in S$ or $p_2 \in S$; and,
- p_1 or $p_2 \in S$ iff $p_1 \in S$ or $p_2 \in S$.

For the case without a sub-expression p' the first three rules are adapted in the obvious way.

For each expression p, $p \in D$ if $p \in S_i \cup D_i$, for some i.

It remains to describe how a DFA B of exponential size can execute the above rules. There is a linear number of sub-expressions of p and self-closures of those. The DFA B keeps for each of them one bit in memory indicating whether the corresponding expression is in S or D. Initially none of them are. An expression is put in a set if one of the above rules fire.

Every rule should be checked at every transition step. So, the size of each B is exponential in p. As A_p contains an exponential number of such DFAs, its size is also exponential.

Let A_d be the exponential size deterministic automaton accepting d (cf. Lemma 2.9(3)). Then deciding whether $p \subseteq_d q$ reduces to testing whether $L(A_d) \cap L(A_p) \subseteq L(A_q)$. By Lemma 2.9(2), the latter can be done in EXPTIME.

□

Theorem 4.5. XDCONTAINMENT for XP($/, //, |$) is hard for EXPTIME.

Proof. The proof makes use of a reduction from TWO-PLAYER CORRIDOR TILING. This is the extension of CORRIDOR TILING, used in the proof of Theorem 3.3, to two players. Let $T = (D, H, V, \bar{b}, \bar{t}, n)$ be a tiling system. Again, D is a finite set of tiles; $H, V \subseteq D \times D$ are horizontal and vertical constraints, respectively; $\bar{b}$ and $\bar{t}$ are n-tuples of tiles; and n is a natural number. There are two players (I and II) that place tiles in turn 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 from the bottom. Each player in turn places a tile on the next free square going from left to right and from bottom to top. While player I tries to construct a corridor tiling from $\bar{b}$ to $\bar{t}$, player II tries to prevent it. If player II places a tile which is not consistent with respect to the horizontal and vertical constraints then player I can answer with a special tile '!'. Player I wins if a tiling is constructed satisfying the horizontal and vertical constraints with the top row tiled with $\bar{t}$, or if she answers an inconsistent tile placed by II with '!'. We say that player I has a winning strategy if she can always win no matter how II plays. It is well-known that it is EXPTIME-complete to determine whether I has a winning strategy [8]. This result even holds if the number of tiles in a row is forced to be even. Thus we assume in the following that n is even.

We encode strategies for player I as trees. For each position in the game in which II moves, the tree contains all possible moves of player II and, for each I-position, it contains only one move. Thus, such a tree encodes a winning strategy if and only if each path corresponds to a correct tiling or to a wrong move of II.

To this end, we use symbols of the form (a, i), where $a \in D$ and $i \in \{1, 2\}$ plus some additional auxiliary symbols: \$ to indicate the borders between rows, # to mark the end of a tiling and the "protest symbol" '!'. Inner nodes of the tree correspond either to moves of I or II. Nodes corresponding to I are labeled $(a, 1)$ where a is the tile chosen by I in this move. They have one child for every tile corresponding to the possible next moves of II. Nodes corresponding to II are labeled $(a, 2)$ and have only one child which is the unique answer move of I according to her strategy or one of \$, # or '!'. Nodes with label \$ have one child corresponding to a move of I. Nodes with label # or '!' are leaves. The root of the tree is labeled S and represents an empty (dummy) move which has to be answered by I. It has one child. On each path from the root, I-nodes and II-nodes alternate (when we ignore the intermediate \$ nodes).

Now we describe the reduction in more detail. Let $D = \{d_1, \dots, d_m\}$. We use the following DTD f which defines all possible strategy trees for player I:

$S \to (d_1, 1) + \cdots + (d_m, 1)$
$\$ \to (d_1, 1) + \cdots + (d_m, 1)$
and for every $d \in D$, we have the rules
$(d, 1) \to (d_1, 2) \cdots (d_m, 2)$
$(d, 2) \to (d_1, 1) + \cdots + (d_m, 1) + \# + \$ + !$
$! \to \varepsilon$
$\# \to \varepsilon$

Note that # and '!' are the only terminal symbols. Thus, each path in the tree either ends with # or '!'.

A derivation tree encodes a strategy tree (or game tree) for I. As the bottom and the top row are fixed we do not represent them in these trees, i.e., only intermediate rows are represented. 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. We will construct an expression q which selects a tree if and only if it does *not* encode a winning strategy for I. Thus, $S \subseteq_f q$ if and only if player I has no winning strategy.

We define

$$q := q_{n \text{ tiles}} \mid q_V \mid q_H \mid q_!,$$

where the sub-expressions on the right hand side will be defined shortly. Intuitively, $q_{n \text{ tiles}}$ expresses that some row has a wrong length (which can only be due to player I), q_V and q_H express that some vertical or horizontal constraint, respectively, is violated by I, and $q_!$ expresses that I used the protest symbol wrongly. Each of these sub-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 wild-card available, the disjunction of all alphabet symbols defined by the grammar is a kind of wild-card as the DTD assures that no other symbols occur in the tree. We denote the set of all pairs (σ, i), where σ is a tile and $i \in \{1, 2\}$ by Σ. Further, we denote by $\Sigma_\$$ the set $\Sigma \cup \{\$\}$.

Three types of errors can occur in a strategy tree: (1) the tree is of the wrong shape; (2) player I places an inconsistent tile, or (3) player I uses the symbol '!' although II placed a correct tile.

A row does not contain exactly n tiles.:

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

Recall that we use D as a shortcut for $d_1 \mid \cdots \mid d_m$.

Vertical Constraints are violated.:

$$q_V := q_{\bar{b}} \mid q_{\bar{t}} \mid \bigcup_{(d',d) \notin V} .//(d', 1)/\Sigma_\$^n/(d, 1),$$

where

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

checks the vertical constraints w.r.t. $\bar{b}$, and

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

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

Horizontal Constraints are violated.:

$$q_H := \bigcup_{(d',d)\notin H} .//(d',2)/(d,1).$$

Wrong use of '!'.:

$$q_! := (\bigcup_{\substack{(d_1,d)\in V\\(d_2,d)\in H}} .//(d_1,2)/\Sigma_{\$}^{n}/(d_2,1)/(d,2)/!) \cup (\bigcup_{i=2}^{n} \bigcup_{\substack{(b_i,d)\in V\\(d',d)\in H}} S/\Sigma^{i-2}/(d',1)/(d,2)/!)$$

This expression takes care of the case that player I uses the protest symbol '!' although the last tile placed by player II was consistent with the vertical and horizontal constraints. The second disjunct in this expression takes care of the first row of the game (i.e., the second row in the tiling).

□

By similar techniques, making use of the techniques of Lemma 3 in [27], it can be shown that XDContainment for XP(/, //, [ ], ∗) is hard for exptime.

The outermost union can be handled as in Lemma 3 of [27]. Of course, the DTD has to be adapted accordingly.

5. Containment in the presence of variables

In this section, we study Boolean containment of XPath expressions which allow the comparison of attribute values. More precisely, in this section we consider XPath expression generated by the following grammar.

lpath	::=	lstep | lpath '/' lpath | lpath '|' lpath
lstep	::=	axis '::' nodetest ('['expr']')*
axis	::=	`self` | `child` | `descendant`
expr	::=	fexpr | vexpr | expr 'or' expr
fexpr	::=	lpath
vexpr	::=	variable = attribute | variable ≠ attribute

A variable x is denoted as $\$x$, an attribute a as $@a$. In this section, fragments without filter expressions still allow sub-expressions of the type vexpr. Furthermore, fragments without filter but with disjunction allow disjunctions of sub-expressions of type vexpr (cf. Theorem 5.3(c)).

We consider two different semantics. The original XPath semantics and the existential semantics, introduced in [14].

In the **XPath semantics**, variable bindings are defined in an outer context. In particular, the value of an expression is defined with respect to a variable assignment $\rho : X \to \mathbf{D}$ where X is the set of all variables. We denote the truth value of a filter expression e relative to a variable assignment ρ and a vertex v by $E_t^{\rho}(v, e)$ and the semantics of an XPath expression p by $[\![p]\!]_t^{\rho}$.

Formally, we extend Definition 2.4 as follows. Given a tree t, a node v, an attribute a and an assignment ρ, $E_t^\rho(v, \$x = @a)$ is true if $\rho(x) = \lambda_t^a(v)$. Likewise, $E_t^\rho(v, \$x \neq @a)$ is true if $\rho(x) \neq \lambda_t^a(v)$. The semantics of $[\![p]\!]_t^\rho$ is then defined accordingly and we write $t \models^\rho p$ if there is a vertex v of t such that $(\varepsilon, v) \in [\![p]\!]_t^\rho$.

Deutsch and Tannen [14] consider 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, $t \models p$ under **existential semantics**, if there is a variable assignment ρ and a vertex v of t such that $t \models^\rho p$. We will write this as $t \models_\exists p$.

We denote the allowance of variables under XPath semantics by xvars and under the existential semantics by evars; the presence of inequalities with variables is denoted by $\neq$. For instance, XP$(/, //, \text{xvars}, \neq)$ denotes the XPath fragment with / and //, together with variable equality and inequality under the XPath semantics.

We define Boolean containment of XP-expressions with attribute values correspondingly. More precisely, under the XPath semantics, $p \subseteq q$ holds, if for every tree t and every variable assignment ρ, $t \models^\rho p$ implies $t \models^\rho q$. Under the existential semantics, $p \subseteq q$ if for every tree t, if there is ρ such that $t \models^\rho p$ then there is also a variable assignment π such that $t \models^\pi q$.

In the first subsection, we show that adding variables under XPath semantics to the basic XPath fragment with disjunction results in a XCONTAINMENT problem with PSPACE complexity.

In [14], it is shown that XCONTAINMENT for XP$(/, //, [\,], *, |, \text{evars})$ is Π_2^P-complete [14] (Theorems 2.3 and 3.3). Further, they show that XCONTAINMENT for XP$(/, //, [\,], \text{evars})$ is CONP-complete. We show in the second subsection that in the presence of inequalities these hardness results hold for even smaller fragments.

Adding variables with XPath semantics to the basic fragment with disjunction but *without wild-card* gives a Π_2^P-complete XCONTAINMENT problem. Surprisingly, adding wild-card to this fragment ends up in an undecidable XCONTAINMENT problem.

The results of this section are summarized in Table 3.

For a set X of variables an **equality type** e is an equivalence relation on X. Intuitively, e describes which variables have the same value.

5.1. **XPath semantics.** We start with the PSPACE-upper bound.

Theorem 5.1. XCONTAINMENT for XP$(/, //, [\,], |, *, \text{xvars}, \neq)$ is in PSPACE.

Proof. We basically show that the problem can be reduced to the case without variables. Let p and q be two expressions with variables $\{x_1, \ldots, x_k\}$ and let Σ' be the set of element and attribute names which appear in p or q (and which we assume to be disjoint). Let $\Sigma = \Sigma' \uplus \{\#\}$. Clearly, $p \subseteq q$ holds in general if and only if it holds for trees with element names from Σ. For each equality type e of the variables $x_1, \ldots, x_k$, we construct two expressions p_e and q_e in XP$(/, //, [\,], |, *)$. By construction it then follows that $p \subseteq q$ if and only if, for every equality type e, $p_e \subseteq q_e$. As we can cycle through all equality types e and construct each p_e and q_e in PSPACE, and each single test will be doable in PSPACE, the complexity of the overall algorithm is PSPACE.

Let us first consider the equality type e where all variables are pairwise different. With every tree t and every variable assignment ρ of type e, we associate a tree t_ρ over the alphabet $\Gamma = \Sigma \cup \{x_1, \ldots, x_k, \texttt{none}\}$ as follows. We add, for each attribute a of a node v, a new child of v labeled by a which itself has a child which is labeled by one of $x_1, \ldots, x_k$ or with `none` depending on $@a$. More precisely, if $@a$ equals some $\rho(x_i)$ then the grandchild of v is labeled by x_i otherwise by `none`.

Correspondingly, we construct p_e by replacing in p each subexpression $\$x_i = @a$ 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 construct q_e from q in the same way. It is easy to see that, for each t, and each variable assignment ρ of type e, $t \models^\rho p$ if and only if $t_\rho \models p_e$.

Thus, it remains to test whether, for all trees of the form t_ρ, $t_\rho \models p_e$ implies $t_\rho \models q_e$. This can be done along the lines of the proof of Theorem 3.5. In particular, it is sufficient to consider only $f(p_e)$-bounded trees.

This completes the description of the algorithm for equality type e. If, for some other equality type, two variables get the same value then we can replace one of them by the other in p and in q. Hence, we get possibly fewer variables which are again pairwise different. □

We show that the PSPACE upper bound is tight in the general case and exhibit some lower bounds for restrictions of the formalism.

Theorem 5.2. (a) XContainment for XP$(/, [\,], \text{xvars}, \neq)$ is coNP-hard.
(b) XContainment for XP$(/, //, \text{xvars}, \neq)$ is coNP-hard.
(c) XContainment for XP$(/, //, |, *, \text{xvars}, \neq)$ is PSPACE-hard.

Proof. The proofs of (a) and (b) are by reduction from the set of unsatisfiable 3SAT-formulas. To this end let $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_k$ be a 3CNF formula where each φ_i is a disjunction $l_{i1} \vee l_{i2} \vee l_{i3}$ of 3 literals. Let $x_1, \ldots, x_m$ be the variables occurring in φ. In both cases, intuitively we construct from φ an expression p that, describes all (tree representations of) possible truth assignments to the literals of φ. The expression q selects a truth assignment if it leaves at least one clause unsatisfied. Thus, φ is unsatisfiable iff $p \subseteq q$.

More precisely, both reductions map φ to expressions with variables y and $y_1, \ldots, y_m$. The value $\rho(y)$ represents `true`, the other variables represent a truth assignment to $x_1, \ldots, x_m$ in the obvious way: x_i is `true` iff $\rho(y_i) = \rho(y)$.

In the trees we are interested in, the root a-attribute carries the data value corresponding to `true` and each literal l_{ij} is represented by one node v_{ij} with an attribute a and label b. Expression p tests that the attribute values of these nodes are consistent with the truth assignment induced by the y_l: this is checked by variable expressions $v(l_{ij})$ which are $\$y_l = @a$ if $l_{ij} = x_l$ and $\$y_l \neq @a$ if $l_{ij} = \neg x_l$.

(a) In case of XP$(/, [\,], \text{xvars}, \neq)$, the trees selected by p have one path of length 3, for each clause. To this end, let, for each i, p_i be the sub-expression

$$p_i = b[v(l_{i1})]/b[v(l_{i2})]/b[v(l_{i3})],$$

and let p be

$$p = b[\$y = @a][p_1] \cdots [p_k].$$

Thus, $t \models^\rho p$ guarantees that t has, for each clause φ_i, a path in which the j-th node $(j = 1, 2, 3)$ has an attribute value consistent with the truth value of l_{ij}, as induced by ρ.

Thus, formula φ is unsatisfiable, if every tree which "passed" p has a path of length 3 in which all induced truth values are `false`. To this end, we define

$$q = b[\$y = @a]/b[\$y \neq @a]/b[\$y \neq @a]/b[\$y \neq @a].$$

(b) Recall that in XP$(/, //, \text{vars}, \neq)$ sub-expressions of the forms $[\$y = @a]$ and $[\$y \neq @a]$ are allowed. As XP$(/, //, \text{vars}, \neq)$ can not refer to branches of a tree, in this case, the vertices

v_{ij} have to be organized in a linear fashion. Basically, in the construction of (a), we replace the general [ ] subexpressions in p by using // in q. To this end, the trees matched by p need to have a path of length $4k$ with label pattern $(cbbb)^*$. The i-th subpath labeled $cbbb$ corresponds to φ_i.

Thus,

$$\begin{aligned} p_i &= c/b[v(l_{i1})]/b[v(l_{i2})]/b[v(l_{i3})] \\ p &= b[\$y = @a]/p_1/\cdots/p_k \\ q &= b[\$y = @a]//c/b[\$y \neq @a]/b[\$y \neq @a]/b[\$y \neq @a] \end{aligned}$$

By a similar reasoning as in (a) it follows that φ is satisfiable if and only if $p \not\subseteq q$.

(c) We use basically the same construction as in Theorem 3.3. Let $D = \{\sigma_1, \ldots, \sigma_k\}$ be the alphabet used in that construction and assume without loss of generality 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. More precisely, $a_1, \ldots, a_l$ and x represent σ_b, if, the i-th digit of b is 1 exactly if $x = a_i$. The remaining symbols \$ and # are still represented by labels. In the expressions p and q the element tests are replaced by the wild-card symbol together with the respective attribute comparisons.

□

5.2. **Existential semantics.** Now we turn to XContainment in the context of existential semantics for variables. In some cases the complexity does not change as to compared with the XPath semantics, but in others it raises considerably, even leading to undecidability for XP(/,//, [ ],∗,|,evars,≠).

Theorem 5.3. (a) XContainment for XP(/, [ ], evars) is coNP-hard.
(b) XContainment for XP(/, [ ], evars, ≠) is Π_2^P-hard.
(c) XContainment for XP(/, |, evars) is Π_2^P-hard.

Proof. The proofs for (a) and (b) are similar. Both are reductions from containment for Boolean Conjunctive Queries (BCQs). In (b) inequalities are allowed, in (a) not. As containment of BCQs and BCQs with inequality is hard for coNP [6] and Π_2^P [34], respectively, (a) and (b) then follow.

We first describe how we represent a relational database D by a tree t_D. The root is labeled with S and for every relation R in D and every tuple $(d_1, \ldots, d_n)$ in R it has a child labeled R with n attributes $@1, \ldots, @n$, where, for each i, $@i$ has the value d_i.

A BCQ is a conjunction of relational atoms $R(x_1, \ldots, x_k)$, and, in case of (b), inequalities $x \neq y$. Note that atoms of the form $x = y$ can be eliminated as each variable must occur in a relational atom. An atom $R(x_1, \ldots, x_k)$ can be represented by the subexpression $p_i = R[\$x_1 = @1]\cdots[\$x_k = @k]$.

We can not represent an inequality $x \neq y$ by an expression $[\$x \neq \$y]$, as variables can only be compared with attributes. Nevertheless, as x must occur as the i-th entry in a relational atom, we add $[y \neq @i]$ to the expression for L.

We illustrate the construction with an example. For instance, if Q is $E(x, y), E(y, z), x \neq z$ then p_Q is

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

Clearly, for a database D, $D \models Q$ if and only if $t_D \models_\exists p_Q$. On the other hand, from each t (even if it is not of the intended form) we can define, in a straightforward manner, using all correctly encoded tuples, a database $D(t)$ such that $D(t) \models Q$ if and only if $t \models_\exists p_Q$.

Thus, for BCQs Q_1 and Q_2 it holds $Q_1 \subseteq Q_2$ if and only if $p_{Q_1} \subseteq p_{Q_2}$, as required.

The proof of (c) is by a reduction from $\forall\exists$-3SAT [31]. Note that the fragment XP$(/, |, \text{evars})$ allows the use of disjunction of variable-attribute comparisons.

Let $\varphi = \forall x_1 \cdots x_m \exists y_1 \cdots y_m \theta$, where $\theta = \theta_1 \wedge \cdots \wedge \theta_n$ is in 3-CNF with variables from $\{x_1, \ldots, x_m, y_1, \ldots, y_m\}$. Intuitively, the $\forall\exists$-structure of a formula will be mimicked by a *for all trees there is a variable assignment* statement. Hence, the values for $x_1, \ldots, x_m$ will be encoded in the trees, the values for $y_1, \ldots, y_m$ will be given by the variable assignment.

Each tree satisfying p will have two root attributes @T and @F which represent the truth values `true` and `false`, respectively. Note that, as there are no inequalities available, there will be no guarantee that the values of T and F are different. We only make use of the node label b. With every path of length m which starts immediately below the root we associate a (partial) truth assignment as follows: if the c-attribute of the i-th node equals the value of the attribute T, we set $\rho(x_i) = T$, if it is F then $\rho(x_i) = F$.

The expression p is designed to guarantee the existence of such a path. To this end, p is $b[\$z_1 = @T][\$z_0 = @F]/p_1/\cdots/p_m$, where, for each i, $p_i = b[\$z_1 = @c \text{ or } \$z_0 = @c]$. Intuitively, the variables z_0 and z_1 are used to transport the values `true` and `false` in the tree.

So far, each tree t with $t \models_\exists p$ induces at least one truth assignment for $x_1, \ldots, x_m$.

Expression q checks that

- z_1 and z_0 take the values @T and @F at the root, respectively,
- each (XPath) variable x_i has a truth value corresponding to the value of the i-th node of some path of the tree,
- each (XPath) variable y_i is either @T or @F, and
- the induced truth assignment makes θ true.

To this end, $q = b[e_1]\cdots[e_n][\$z_1 = @T][\$z_0 = @F][\$y_1 = @T \text{ or } \$y_1 = @F]\cdots[\$y_m = @T \text{ or } \$y_m = @F]/b[\$x_1 = @c]/\cdots/b[\$x_m = @c]$, where each e_i is a variable expression corresponding to θ_i. As an example, if θ_1 is $x_2 \vee y_3 \vee \neg x_1$ then e_1 is $[\$x_2 = @T \text{ or } \$y_3 = @T \text{ or } \$x_1 = @F]$.

Now it is easy to see that φ holds iff $p \subseteq q$ with respect to the existential semantics. Note in particular that trees for which $@T = @F$ at the root fulfill q whenever they fulfill p. □

Theorem 5.4. XContainment for XP$(/, //, [\,], |, \text{evars}, \neq)$ under existential semantics is in Π_2^P.

Proof. We show that for XP$(/, //, [\,], |, \text{evars}, \neq)$-expressions the following holds.

(a) $p \not\subseteq q$ if and only if there is a tree t of size polynomial in $|p| + |q|$ such that $t \models_\exists p$ but $t \not\models_\exists q$, and

(b) $t \models_\exists p$ 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 XContainment.

To prove (a), let p and q be expressions and let $p_1|\cdots|p_m$ and $q_1|\cdots|q_n$ be the DNF of p and q, respectively. As the disjuncts do not contains disjunction themselves they can again be represented as tree patterns 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, as i can be guessed, in proving (a) we can restrict to the case where p does not contain $|$.

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

- The tree structure of t is obtained from the tree pattern $\tau(p)$ by replacing each //-edge by a path of length two with two child edges and a new intermediate #-labeled node where # is a label neither occurring in p nor q. Note that the number of vertices of t is at most twice the number of vertices of $\tau(p)$.
- 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 occurring in p or q.

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

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

Therefore let $p \not\subseteq q$ be witnessed by a tree t' not necessarily from $S(p, q)$. Hence, $t' \models_\exists p$ but $t' \not\models_\exists q$. Let e be a homomorphism from $\tau(p)$ to t'. Let $a_1, \dots, a_l$, $l \leq k$, be the pairwise different attribute values of the vertices in $e(\tau(p))$.

We construct t as follows. Its structure is obtained from $\tau(p)$ as above by replacing //-edges with new nodes labeled #. We call a vertex v of t that is already in $\tau(p)$ an *original vertex* and write $p(v)$ for its corresponding vertex in $\tau(p)$. 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 $\tau(q_j)$ to t. As q does not contain the symbol # and there are no wild-cards, the image of $\tau(q_j)$ 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. This concludes the proof of (a).

To show (b), we remark that whether $t \models_\exists p$ for an expression p in $\mathrm{XP}(/, //, [\,], |, \mathrm{evars}, \neq)$ can be tested as follows. First, a disjunct p_i of the disjunctive normal form of p is guessed. Next, a homomorphism from $\tau(p_i)$ 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. □

We note in passing that for variables with existential semantics even query evaluation is hard.

Proposition 5.5. (a) Evaluation of Boolean $\mathrm{XP}(|, \mathrm{evars})$-expressions is NP-hard.
(b) Evaluation of Boolean $\mathrm{XP}(/, [\,], \mathrm{evars})$-expressions is NP-hard.

Proof. (a) The proof is by reduction from 3SAT. Let $\varphi = \bigwedge_{i=1}^{n} \varphi_i$ be a 3CNF formula with variables from $\{x_1, \dots, x_m\}$. Let t be the tree consisting of a single vertex with attributes $@T = 1$ and $@F = 0$. Let p be the expression $[\$x_1 = @T \text{ or } \$x_1 = @F] \cdots [\$x_m = @T \text{ or } \$x_m = @F]p_\varphi$, where $p_\varphi = p_{\varphi_1} \cdots p_{\varphi_n}$ and each p_{φ_i} represents clause φ_i. E.g., if $\varphi_i = (x_3 \vee \neg x_5 \vee x_1)$ then p_{φ_i} is $[\$x_3 = @T \text{ or } \$x_5 = @F \text{ or } \$x_1 = @T]$. Clearly, φ is satisfiable iff $t \models_\exists p$.

(b) This follows easily from the correspondence between XP and BCQ as explained in the proof of Theorem 5.3 and the fact that BCQ-evaluation is hard for NP.

□

The following theorem shows that in the setting of variables with existential semantics the wild-card has a strong impact.

Theorem 5.6. XContainment for XP(/,//,$*$,|,evars,$\neq$) is undecidable.

Proof. We use a reduction from Post's Correspondence Problem (PCP) which is well-known to be undecidable [19]. An *instance* of PCP is a sequence of pairs $(x_1, y_1), \dots, (x_n, y_n)$, where $x_i, y_i \in \{a,b\}^*$ for $i = 1, \dots, n$. This instance has a *solution* if there exist $m \in \mathbb{N}$ and $\alpha_1, \dots, \alpha_m \in \{1, \dots, n\}$ 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. At the end of the proof, we will explain how we can get rid of the DTD.

We consider unary XML-trees, that is, strings. They are roughly of the form $u\$v$, where \$ is a delimiter and u, v are strings representing a candidate solution $(x_{\alpha_1}, \dots, x_{\alpha_m}; y_{\beta_1}, \dots, y_{\beta_m})$ 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) for each i, $\alpha_i = \beta_i$, 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 (1) and (2), we make use of a double indexing system based on the values of the attributes block and position of the nodes in u and v. We explain the intuition behind our reduction by means of a small concrete example.

The DTD will define *strings* of the form $S\ u\#v\&$. For instance, the candidate solution $x_1x_2; y_1y_2$ where $x_1 = ab$, $x_2 = b$, $y_1 = a$, and $y_2 = bb$ will be represented as a concatenation of X- and Y-blocks 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	

Here, the block and position rows indicate the values of the respective attributes. The symbols X and Y indicate the beginning of an x- and y-block, respectively. The symbol $1(x)$ 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 attributes "block" and "position" make up the double index system as will become clear further on. We refer to blocks corresponding to encodings of an x_i and y_i as X-blocks and Y-blocks, respectively. If a block corresponds to x_i or y_i we say that its number is i.

Let, for each $i \leq n$, x_i be $\sigma_1^i \cdots \sigma_{k_i}^i$ and y_i be $\delta_1^i \cdots \delta_{\ell_i}^i$. Then the DTD d consists of the productions

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

and further sets $P(x_i)$ and $P(y_i)$ of productions, for each $i \leq n$. Here, $P(x_i)$ consists of the productions $i(x) \to \sigma^i_1(x,i,1)$, and for $j := 1,\ldots,k_i - 1$, $\sigma^i_j(x,i,j) \to \sigma^i_{j+1}(x,i,j+1)$, and $\sigma^i_{k_i}(x,i,k_i) \to X \mid \#$. Analogously, $P(y_i)$ is the set of productions $i(y) \to \delta^i_1(y,i,1)$, and for $j := 1,\ldots,\ell_i - 1$, $\delta^i_j(y,i,j) \to \delta^i_{j+1}(y,i,j+1)$, and $\delta^i_{\ell_i}(y,i,\ell_i) \to Y \mid \&$. The start symbol is S. Every X and Y has an attribute *block*; every a and b has an attribute *position*.

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

Let p be the XP-expression S and let d be as above. We next construct q' in such a way that it selects the root of an XML-tree if and only if it is *not* syntactically correct or does *not* represent a solution. As p defines *all* inputs, $p \subseteq_d q'$ if and only if the PCP instance has *no* solution.

In the following, if z is the string $abab$ then we use z as a shorthand for $a/b/a/b$. Further, we denote the string generated by the grammar from x_i and y_i by $\tilde{x}_i$ and $\tilde{y}_i$, respectively. That is, for $x_1 = ab$, $\tilde{x_1} = a(x,1,1)b(x,1,2)$.

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:
   $$S/X[\$d = @\mathit{block}]//\#/Y[\$d \neq @\mathit{block}].$$
   - (b) the block value of the last X in u differs from the block value of the last Y in v: for each $i,j \in \{1,\ldots,n\}$ we have
   $$S//X[\$d = @\mathit{block}]/i(x)/\tilde{x}_i/\#//Y[\$d \neq @\mathit{block}]/j(y)/\tilde{y}_i\&$$
   - (c) two X-block values are the same:
   $$S//X[\$d = @\mathit{block}]//X[\$d = @\mathit{block}]//\#$$
   - (d) two Y-block values are the same;
   $$S//\#//Y[\$d = @\mathit{block}]//Y[\$d = @\mathit{block}]$$
   - (e) two successive X-block values are not successive in v: for all $i,j \in \{1,\ldots,n\}$ we have
   $$S//X[\$d = @\mathit{block}]/i(x)/\tilde{x}_i/X[\$e = @\mathit{block}]// \\ \#//Y[\$d = @\mathit{block}]/j(y)/\tilde{y}_j/Y[\$e \neq @\mathit{block}].$$
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:
   $$S/X/*/*[\$d = @\mathit{position}]//Y/*/*[\$d \neq @\mathit{position}].$$
   - (b) the last position in u differs from the last in v:
   $$S//*[\$d = @\mathit{position}]/\#//*[\$d \neq @\mathit{position}]/\&$$
   - (c) two X-position values are the same:
   $$S//*[\$d = @\mathit{position}]//*[\$d = @\mathit{position}]//\#$$

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

$$S/\!/\#/\!/*[\$d = @\mathit{position}]/\!/*[\$d = @\mathit{position}]$$

  (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 $i, j \in \{1, \dots, n\}$, $k \in \{1, \dots, |x_i| - 1\}$, $\ell \in \{1, \dots, |y_j| - 1\}$:

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

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

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

    (iii) the X-positions occur in the same block, the Y-positions occur in successive blocks: for all $i, j \in \{1, \dots, n\}$, $k \in \{1, \dots, |x_i| - 1\}$:

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

    (iv) the X-positions occur in successive blocks, the Y-positions occur in successive blocks: for all $i, j \in \{1, \dots, n\}$:

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

(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 $i \neq j$:

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

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

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

  and

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

Clearly, w is not syntactically correct or does not represent a solution if and only if one of the above conditions holds.

To get rid of the DTD, we add disjuncts to q' that capture violations of the DTD. However, to this end we need to express that children of a node cannot have a certain label. As we can not express this kind of negation directly, we encode labels of nodes by equality types of attribute values. So, let $L := \{\ell_1, \ldots, \ell_m\}$ be the set of all the labels we need. Every node now has m attributes $a_1, \ldots, a_m$. If for a node, $j \geq 1$ is the largest number such that the value of a_1 equals the value of a_j then the node is considered as labeled with ℓ_j. One can match a node labeled with ℓ_j by checking the corresponding equality type: for instance, by an expression of the form

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

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 q' 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 q be obtained from q' by adding all the disjuncts from the DTD and replacing all references to labeling by references to encoding with attributes. Note that now we allow arbitrary trees as well.

It remains to argue that $p \not\subseteq q$ 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 but not q. Suppose that there is a tree that matches p but not q. 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. □

6. Discussion

The article studied the complexity of the containment problem for a large class of XPath expressions. 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 modest negation ($\neq$) and variables with the existential semantics. In [30] a corresponding result is shown in the presence of node-set equality. However, the reduction employs XPath expressions with absolute filter expressions which fall outside the scope of the present paper. It would be interesting to have a precise classification of which combination of features makes the problem undecidable.

Although the complexity has been settled for a lot of fragments of XPath with child and descendant axes, the picture is by no means complete. A particular case that remains open is the case of XP(DTD, /, //, $*$).

Corresponding results in the presence of other axes remain to be investigated. Besides the general upper bound of Marx [24] for navigational XPath with all axes, few precise results are known. Work on satisfiability of XPath with the sibling axis has been done by Fan and Geerts [15].

Acknowledgment

We thank Stijn Vansummeren for comments on a previous version of this paper. We thank the anonymous referees of our ICDT 2003 paper and the referees of this article for many valuable suggestions.

References

[1] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *Proceedings 24th Symposium on Principles of Database Systems (PODS 2005)*, pages 25–36. ACM Press, 2005.

[2] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. *Theoretical Computer Science*, 336(1):3–31, 2005.

[3] G. J. Bex, S. Maneth, and F. Neven. A formal model for an expressive fragment of XSLT. *Information Systems*, 27(1):21–39, 2002.

[4] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets: Version 1, april 3, 2001. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[5] 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.

[6] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings 9th ACM Symposium on Theory of Computing*, pages 77–90. ACM, 1977.

[7] A. K. Chandra, D. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[8] B. S. Chlebus. Domino-tiling games. *Journal of Computer and System Sciences*, 32(3):374–392, 1986.

[9] J. Clark. XML Path Language (XPath). `http://www.w3.org/TR/xpath`.

[10] James Clark. XSL transformations version 1.0. http://www.w3.org/TR/WD-xslt, august 1999.

[11] World Wide Web Consortium. XML schema. `http://www.w3.org/XML/Schema`.

[12] S. DeRose, E. Maler, and R. Daniel. XML pointer language (XPointer) version 1.0. `http://www.w3.org/TR/xptr/`, 2001.

[13] S. DeRose, E. Maler, and D. Orchard. XML linking language (XLink) version 1.0. `http://www.w3.org/TR/xlink/`, 2001.

[14] 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.

[15] F. Geerts and W. Fan. Satisfiability of XPath queries with sibling axes. In G. M. Bierman and C. Koch, editors, *Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 122–137. Springer, 2005.

[16] G. Gottlob and C. Koch. Monadic queries over tree-structured data. In *Proceedings 17th IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 189–202, 2002.

[17] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of 28th Conference on Very Large Databases (VLDB)*, pages 95–106, 2002.

[18] J. Hidders. Satisfiability of XPath expressions. In G. Lausen and D. Suciu, editors, *Database Programming Languages (DBPL 2003)*, volume 2921 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2004.

[19] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[20] H. Lewis and C. Papadimitriou. *Elements of the theory of computation.* Prentice-Hall, 2 edition, 1997.

[21] W. Martens and F. Neven. On the complexity of typechecking top-down XML transformations. *Theoretical Computer Science*, 336(1):153–180, 2005.

[22] W. Martens and J. Niehren. Minimizing tree automata for unranked trees. In G. M. Bierman and C. Koch, editors, *Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 2005.

[23] M. Marx. Conditional XPath, the first order complete XPath dialect. In *Proceedings 23th Symposium on Principles of Database Systems (PODS 2004)*, pages 13–22. ACM Press, 2004.

[24] M. Marx. XPath with conditional axis relations. In E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Böhm, and E. Ferrari, editors, *Advances in Database Technology (EDBT 2004)*, volume 2992 of *Lecture Notes in Computer Science*, pages 477–494. Springer, 2004.

[25] M. Marx. First order paths in ordered trees. In *Proceedings of the 10th International Conference on Database Theory (ICDT)*, pages 114–128, Berlin, 2005. Springer.

[26] M. Marx and M. de Rijke. Semantic characterizations of navigational XPath. *SIGMOD Record*, 34(2):41–46, 2005.

[27] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.

[28] G. Moerkotte. Incorporating XSL processing into database engines. In *Proceedings of 28th Conf. on VLDB*, pages 107–118, 2002.

[29] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3), 2002.

[30] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, pages 315–329, 2003.

[31] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[32] R. Sedgewick. *Algorithms.* Addison-Wesley, 1983.

[33] G. Slutzki. Alternating tree automata. *Theoretical Computer Science*, 41(2–3):305–318, 1985.

[34] R. Van der Meyden. The complexity of querying indefinite data about linearly ordered domains. *Journal of Computer and System Sciences*, 54(1):113–135, 1997.

[35] P. T. Wood. Minimising simple XPath expressions. WebDB informal proceedings, 2001.

[36] 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.

[37] P. T. Wood. Containment for XPath fragments under DTD constraints. In D. Calvanese, M. Lenzerini, and R. Motwani, editors, *Database Theory - ICDT 2003*, volume 2572 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2002.

Hasselt University and Transnational University of Limburg
E-mail address: frank.neven@uhasselt.be

University of Dortmund
E-mail address: thomas.schwentick@udo.edu