Containment and Integrity Constraints for XPath Fragments

Alin Deutsch * Val Tannen

University of Pennsylvania

Abstract

XPath is a W3C standard that plays a crucial role in several influential query, transformation, and schema standards for XML. Motivated by the larger challenge of XML query optimization, we investigate the problem of containment of XPath expressions under integrity constraints that are in turn formulated with the help of XPath expressions. Our core formalism consists of a fragment of XPath that we call *simple* and a corresponding class of of integrity constraints that we call *simple XPath integrity constraints* (`SXIC`). `SXIC`'s can express many database-style constraints, including key and foreign key constraints specified in the XML Schema standard proposal, as well as many constraints implied by DTDs. We identify a subclass of *bounded* `SXIC`'s under which containment of simple XPath expressions is *decidable*, but we show that even modest use of unbounded `SXIC`'s makes the problem *undecidable*. In particular, the addition of (unbounded) constraints implied by DTDs leads to undecidability.

We give tight Π_2^p bounds for the simple XPath containment problem and tight NP bounds for the disjunction-free subfragment, while even identifying a PTIME subcase. We also show that decidability of containment under `SXIC`'s still holds if the expressions contain certain additional features (e.g.., wildcard) although the complexity jumps to Π_2^p even for the disjunction-free subfragment.

We know that our results can be extended to some but not all of the XPath features that depend on document order. The decidability of containment of simple XPath expressions in the presence of DTDs only remains open (although we can show that the problem is PSPACE-hard) as well as the problem for full-fledged XPath expressions, even in the absence of integrity constraints.

1 Introduction

We have a general interest in the algorithmic foundations of XML query optimization. The core problem considered here is query equivalence (more generally—*query containment*) under integrity constraints. There is a large body of research on using constraints in query optimization in traditional databases. Such results do not

*Contact author, Univ. of Pennsylvania, 200 S. 33rd St., Philadelphia, PA 19104, `adeutsch@saul.cis.upenn.edu`

apply directly to XML queries because of the transitive closure (Kleene star) operator in path expressions, which is not first-order expressible. Significant work that does handle the Kleene star operator has been done on containment of semistructured queries [11, 4, 5]. But these results do not apply directly here because the XML models are trees rather than arbitrary graphs. Nonetheless, the present work benefits from ideas introduced in all these previous papers.

Integrity constraints are a fundamental mechanism for semantic specification in traditional databases. For XML, the design of specification formalisms for integrity constraints is still an ongoing effort, from DTDs [13], to recent work on keys [3] and database-style constraints [10, 9] and the current XML Schema standardization effort [15].

Several of these formalisms build on the XPath standard [14] or on closely related languages. XPath is also central to XML transformation and query language standards (XSLT [17], respectively XQuery [16]). Consequently, understanding the foundations of XPath query optimization is an important step in tackling the more general problem of XML query optimization.

Here is an example of XPath expression

$$P \stackrel{\text{def}}{=} //(a|b)/c[@m = "0" \ \texttt{and} \ .//d \ \texttt{and} \ . = //.[@o]//.]/@n$$

To describe the meaning of this notation recall that an XML document represents a rooted tree whose nodes include *elements* and *attributes* [13]. Given a *context* node x in the document tree, the meaning $[\![p]\!]x$ of an XPath expression p is the set of nodes that can be reached from x when "navigating" the tree according to p. Then one needs to explain how navigation composes for the various operators that appear in the XPath definition. For example, $[\![p_1/p_2]\!]x$ is the union of all $[\![p_2]\!]y$ for all y that are element children of the nodes in $[\![p1]\!]x$. Similarly for $[\![/p_2]\!]x$ except y ranges over the element children of the root node (x is not used!). For $p_1//p_2$ and $//p_2$ replace "children" with "descendants". Moreover, we have $[\![.]\!]x = \{x\}$ while $[\![n]\!]x$ and $[\![@n]\!]x$ consist of the element children, respectively attributes, of node x that have tag, respectively name, n. Finally, $[\![p[q]]\!]x$ selects those nodes y in $[\![p]\!]x$ such that $[\![q]\!]y$ holds true.

Accordingly, P above returns the set of attributes named n of all elements tagged c which are children of an element tagged a or b and have an attribute named m of value "0", a descendant element tagged d and some ancestor with an attribute named o. There is much more about XPath and its semantics that can be found in [14, 18]. The space limitations prevent more detailed explanations here. Additional operators are described below. [8] contains precise semantic definitions for the XPath fragments we analyze.

We also consider a class of XML integrity constraints that combine the logical shape of the classical relational dependencies [1] with a controlled amount of disjunction and with atoms defined by the XPath expressions themselves. Here are a few examples of constraints

$$\begin{array}{rl}
(oneAddress) & \forall x \forall s_1 \forall s_2 \ [//person \ x \ \wedge x \ ./address \ s_1 \wedge x \ ./address \ s_2 \ \rightarrow s_1 = s_2] \\
(someAddress) & \forall x \ [//person \ x \ \rightarrow \ \exists y \ x \ ./address \ y] \\
(idref) & \forall x \ [//person/@spouse \ x \ \rightarrow \exists y \ //. \ y \wedge y \ ./@* \ x] \\
(key_{s,p}) & \forall x, y, s \ [//person \ x \ \wedge //person \ y \wedge x \ ./@ssn \ s \wedge y \ ./@ssn \ s \rightarrow x = y]
\end{array}$$

$$(grandpa) \quad \forall x,y,z,u\ [x\ ./(son|daughter)\ y \wedge y\ ./(son|daughter)\ z \wedge x\ ./@ssn\ u \rightarrow z\ ./@grandparent\ u]$$

DTDs imply some of these constraints. Consider the DTD entry `<!ELEMENT person (address?,...)>` stating (among other things) that *person*-elements have at most one *address*-element nested within them. We express this as (*oneAddress*) above. With `address+` instead of `address?` we assert the existence of at least one *address*-subelement, thus implying (*someAddress*). (*idref*) above holds whenever a DTD describes the *spouse*-attribute of *person*-elements to have type IDREF (the wildcard $*$ in $@*$ is used to say that there is some attribute of *unspecified* name, in an element y of unspecified tag, agreeing on its value with the value of the *spouse*-attribute). The constraint ($key_{s,p}$) captures the fact that the *ssn*-attribute is a key for *person*-elements (this is not DTD expressible, but can be stated in XML Schema and in [10, 3, 9]). (*grandpa*) expresses another useful kind of constraint which is reminiscent of relational inclusion dependencies [1] but goes beyond the formalisms of [10, 3, 9] or XML Schema.

As we can see, inspired loosely by path specification in UNIX-like file directory systems, XPath was designed to be a (1) compact and (2) expressive notation. Its full-fledged definition has many features inspired by practical considerations. The techniques that we bring to bear in this paper can tackle many of the features of full-fledged XPath, but not all. Our approach (explained in section 4) limits the XPath expressions we can analyze formally to a subclass we call *simple* and some significant extensions thereof, handled separately because they feature distinct complexities of the containment problem.

2 Simple XPath

Simple XPath expressions are generated by the following grammar (n is any tag or attribute name, v any variable name, and s any string constant):

$$\begin{aligned}(simple\ xpath)\ p\ &::=\ p_1|p_2 \mid /p \mid //p \mid p_1/p_2 \mid p_1//p_2 \mid p[q] \mid . \mid n \mid @n \mid @* \mid \$v := @n \mid \\ &\quad\ \ \texttt{text}() \mid \texttt{id}(p) \mid \texttt{id}(s) \\ (qualifier)\ q\ &::=\ q_1\ \texttt{and}\ q_2 \mid q_1\ \texttt{or}\ q_2 \mid p \mid p = s \mid @n = \$v \mid \$v_1 = \$v_2\end{aligned}$$

Simple XPath expressions feature both an extension and restrictions from the XPath standard. The extension is their ability to *bind variables*. Variables are introduced by the \$ sign, and only for attribute values. The meaning of $\$v := @n$ is that of binding variable v to the string value of the current node's n-attribute. The test $@n = \$v$ in a qualifier is satisfied if the value of the current node's n-attribute equals the value v was previously bound to. $\$v_1 = \$v_2$ is satisfied if v_1, v_2 were bound to the same string value. Our ability to bind variables goes beyond the XPath specification, which intends variables to be bound in the outside context (usually represented by XPointer, XSLT or XQuery expressions), and only allows for testing their values inside XPath expressions. However, for practical purposes this does not result in an effective increase in expressive power, since XPath expressions are not meant to be used standing alone, but rather embedded in expressions of the three standards mentioned above. In the case of XQuery for instance (which is what we ultimately want to optimize), an XPath expression that binds variables is just syntactic sugar for a query with several XPath expressions that don't: the XQuery body `For` $/a[@n = \$v]//b[@m = \$v]$ $\$x$ is equivalent to `For` $/a/@n$ $\$v$, $\$v//b$ $\$x$, $\$x/@m$ $\$w$, `Where` $\$v = \$w$.

The most notable restrictions to the full-fledged standard [14] are the absence of the navigation axes `following`, `following-sibling`, `preceding`, `preceding-sibling`. This is because for the time being we disregard the document order, seeing the XML document as an unordered tree, in which these axes have no meaning [1]. Some of these restrictions are lifted in section 5, where we handle `following-sibling` and `preceding-sibling`.

Moreover, we disallow for the moment navigation steps via the child axis from or to elements of unspecified tag. This can be done either by using the wildcard $*$ for going to a child of unspecified tag name, or inversely, by using the `parent` axis to get to a parent of unspecified tag name, and finally, by using the `ancestor` axis, which performs an implicit `parent` navigation step, followed by an `ancestor-or-self` step. This is why `*,parent,ancestor,ancestor-or-self` are missing from our grammar. We treat these separately in section 3 because it turns out that the corresponding containment problem has higher complexity.

We also rule out negation from qualifiers, for the same reason for which negation causes problems in the classical relational dependency theory [1].

Simple XPath integrity constraints (SXICs). We consider dependencies of the general form

$$\forall x_1 \ldots \forall x_n \; [B(x_1, \ldots, x_n) \rightarrow \bigvee_{i=1}^{l} \exists z_{i,1} \ldots \exists z_{i,k_i} \; C_i(x_1, \ldots, x_n, z_{i,1}, \ldots, z_{i,k_i})] \tag{1}$$

where B, C_i are conjunctions of atoms of form $v \; p \; w$ where p is a simple XPath expression or equality atoms of the form $v = w$, where v, w are variables or constants. We demand of course that v, w be of compatible type. v may be missing from a path atom if p's context node is the root of the document (i.e. if p begins with $/$ or $//$).

All constraints shown in section 1 are `SXIC`s. We have seen that some of them are not expressible by DTDs, while others are implied by them. But in general DTDs and `SXIC`s are incomparable. DTD features that cannot be expressed by `SXIC`s are the order of sibling elements, and the fact that an element admits subelements of given tags *only*.

Satisfaction of SXICs. We say that the binding of v to a node a, and of w to a node b satisfies a path atom $v \; p \; w$ if b is equal to some node in the set returned by p when starting from context node a. We define equality as equality of the string values for text and attribute nodes, while an element node is equal only to itself. Equality atoms are satisfied according to this definition. An `SXIC` of general form (1) is satisfied if for any type-consistent binding of the variables $x_1, \ldots, x_n$ that satisfies all atoms in B, there is some $1 \leq i \leq l$ and some extension of this binding to the variables $z_{i,1}, \ldots, z_{i,k_i}$ such that all atoms of C_i are satisfied by the extended binding.

Containment under SXICs. Given a set C of `SXIC`s, and simple XPaths P_1, P_2, we say that P_1 is contained in P_2 under C (denoted $P_1 \subseteq_C P_2$) if every node in the set returned by P_1 is equal (in our sense) to some node in the set returned by P_2 whenever

[1] Note that this view is actually consistent with the XPath 1.0 specification [14], which defines the semantics of XPath expressions as being a *set* of nodes. The upcoming XPath 2.0 is expected to introduce list semantics, which we do not consider here.

both are applied to any XML document which satisfies all SXICs in C. (This definition is more flexible than just asking for containment of the node sets returned by P_1, P_2, because it does not distinguish between attribute and text nodes of distinct identity but equal string value.)

Bounded SXICs. This subclass of SXICs allows the same generality as all SXICs in the left-hand-side B of the implication, but it restricts the form of the right-hand-side of the implication. Namely, the XPath atoms occuring in the conjunctions C_i must have one of the following forms:

$$v = w \qquad v \ ./@n \ w \qquad v \ ./@* \ w \qquad v \ ./n \ w \qquad v \ .//. \ w \qquad //. \ w \qquad /n \ w$$

Moreover, while all occurrences of v, w can be universally quantified, there are restrictions on the cases when they may be existentially quantified. In order to state these restrictions, we introduce the notion of *bounded-depth variable*: we say that variable w is bounded-depth if it appears on either side of the implication in an atom $/n \ w$, or in atoms $v \ ./n \ w$ or $w \ ./n \ v$, with v bounded-depth. The restrictions on existential quantification are given below:

- w must be universally quantified in $v \ ./@n \ w$, $v \ ./@* \ w$ and $v \ .//. \ w$
- v or w may be existentially quantified in $v \ ./n \ w$ only if they are bounded-depth.

All SXIC examples in section 1 are bounded except for (*someAddress*), which contains the existentially quantified, non-bounded-depth variable y.

Theorem 2.1 *Containment of simple XPath expressions under bounded* SXIC*s is decidable. If we fix the constraints, the problem is in* Π_2^p *in the size of the expressions (if we don't, the problem is in EXPTIME in the size of the constraints). If in addition we consider disjunction-free simple expressions and constraints, the complexity drops to NP. Moreover, if we also disallow attribute variables in the expressions, the complexity drops to PTIME.*

In practice, we often know that XML documents satisfy SXICs that are not necessarily bounded, the most salient examples being SXICs implied by DTDs, such as (*someAddress*) from the introduction. Unfortunately, we have the following result:

Theorem 2.2 *Containment of simple XPath expressions under unbounded* SXIC*s is undecidable.*

Complexity lower bounds. It turns out that for fixed SXICs, the upper bounds in theorem 2.1 are tight:

Theorem 2.3 *Just containment of simple XPath expressions (no constraints) is* Π_2^p*-hard. Containment of disjunction-free simple XPath expressions (again no constraints) is NP-hard.*

A missing piece in the puzzle is the lower bound for containment under simple bounded SXICs when the constraints are not fixed. We conjecture EXPTIME-hardness however, expecting that the proof of EXPTIME-hardness for the relational chase [6] can be adapted.

DTDs and SXICs. In XML practice, constraints on the form of documents are often specified using DTDs. A natural question pertains to the status of our decidability results in the presence of DTDs, with or without SXICs. A careful analysis of the proof of theorem 2.2 shows the following. Let C_1 be a set of bounded SXICs without disjunction and existentials. Let D be a DTD and let C_2 be a set of unbounded SXICs implied by D. Let also X_1 and X_2 be two simple XPath expressions. What we prove, in fact, is that the problem of whether $X_1 \subseteq_{C_1 \cup C_2} X_2$ is undecidable.

Corollary 2.4 *Containment of simple XPath expressions is undecidable in both following scenarios (1) under unbounded* SXIC*s, and (2) in the presence of bounded* SXIC*s and DTDs.*

Bounded XICs cover many common cases: given a DTD, it is usually possible to rewrite constraints such as (someAddress) in bounded syntax, unless in the rest of the DTD (which we do not specify) the `address` element is nested (immediately or not) in some “X”-element that may contain a descendant “X”-subelement. Such “cyclic” element declarations are not very common!

The problem of deciding containment of simple XPath expressions under DTDs only (no SXICs) remains open, and the following lower bound which is in fact in the size of the expressions, combined with the upper bound in theorem 2.1, suggests that the techniques that we use in this paper are unlikely to help:

Theorem 2.5 *Containment of simple XPath expressions in the presence of DTDs is PSPACE-hard.*

3 Beyond Simple XPaths

In this section we enrich simple XPath expressions with several navigation primitives from the XPath standard.

Parent axis. We allow navigation to the parent of the current node. Concretely, this amounts to adding the production $p ::=$ `parent` to the grammar in section 2.

Ancestor axes. We allow navigation along the ancestor and ancestor-or-self axis. The corresponding productions are $p ::=$ `ancestor` | `ancestor-or-self`.

Wildcard Child. We further allow navigation along the child axis to elements of unspecified tag, adding $p ::= *$ to our grammar ($*$ is called the *wildcard*). Here is an XPath expression using wildcard child navigation: $P' \stackrel{\text{def}}{=} //c/*[@m = "0]$. It returns the set of elements of unspecified tag (indicated by the $*$), that have an m-attribute of value $''0''$ and a parent tagged c who is a non-immediate descendant of the document root. Note the use of the wildcard $*$ (disallowed in simple XPath expressions).

Path equality. We extend the grammar of qualifiers with the production $q ::= p_1 = p_2$, corresponding to path equality tests. Such tests are satisfied if some node returned by path p_1 is equal to some node returned by path p_2. Equality tests must of course typecheck, and they are satisfied for text and attribute nodes if and only if the *string values* are equal. In contrast, an element node is only equal to itself. This definition of element node equality follows XML-QL [7] as opposed to the ad-hoc treatment in [14].

Although none of the above extensions seems to have anything to do with disjunction, each one of them (except `parent`, for which we do not know what happens) when added to the disjunction-free fragment raises the complexity (recall from theorem 2.1 that it is NP when the constraints are fixed):

Theorem 3.1 *Adding any one of the following to disjunction-free simple XPath expressions makes their containment problem (no constraints)* Π_2^p*-hard: 1. path equality 2. ancestor axis 3. ancestor-or-self axis 4. wildcard child*

However, this is pretty much as far as the complexity raises:

Theorem 3.2 *Consider simple XPath expressions enriched with path equality and ancestor-or-self axis. The containment of such expressions under fixed bounded* `SXIC`*s is in* Π_2^p *in the expression size.*

In dealing with wildcard in this paper we have further restricted the constraints. We believe however that this restriction can be lifted.

Tree `SXIC`s. These are bounded `SXIC`s that satisfy the following additional restrictions: (i) v must be universally quantified in $v\ ./n\ w$ (recall page 5). (ii) We disallow atoms of form $v\ .//\ w$ from the right-hand side of the implication and (iii) For any constraint c and any of c's equality atoms of form $v = w$ (where v, w are variables) in the right-hand side of the implication, if v, w are bound to element nodes, c must contain the atoms $u\ /n\ v$ and $u\ /m\ w$ for some variable u and tag names n, m.

Restriction (iii) ensures that in all models satisfying c, the only expressible key constraints are keys among sibling nodes. Recalling the examples in section 1, (oneAddress) corresponds to this restriction, while $(\mathrm{key}_{s,p})$ does not. The intuition behind all three restrictions is that no combination of tree `SXIC`s can compromise the tree property of a given document, whence their name.

Theorem 3.3 *Consider simple XPath expressions enriched with* `parent`, `ancestor`, `ancestor-or-self` *and wildcard child navigation. The containment of such expressions under fixed* tree `SXIC`*s is in* Π_2^p *in the expression size.*

4 Upper Bound Proof Techniques

For the decision procedure, we set out to leverage techniques from classical relational theory by *reducing containment under constraints to an equivalent first-order question.* We define shortly Σ_{XML} which consists of a relational schema *and* some first-order integrity constraints on this schema. Then, we translate XPath expressions into unions of relational conjunctive queries over the schema of Σ_{XML}. Moreover, we translate `SXIC`s into first-order sentences over the same schema and of the same form as the integrity constraints in Σ_{XML}. Denoting the translation of a set C of SXICs with Σ_C, we will reduce containment of regular XPath expressions under C to containment of unions of relational conjunctive queries under $\Sigma_{\mathrm{XML}} \cup \Sigma_C$.

Any XML document D corresponds to a (finite) Σ_{XML}-instance I_D such that if ϕ is a containment or an SXIC then ϕ holds in D if and only if its Σ_{XML} translation holds in I_D. This makes our reduction by translation to a first-order problem sound. However, transitive closure and "treeness" cannot be captured by the integrity constraints we are willing to allow in Σ_{XML} hence there are Σ_{XML}-instances that do not correspond to any XML document. Any decision procedure used for the first-order problem must therefore be strengthened to make the entire reduction also complete.

The restriction to bounded SXICs allows us to use the classical *chase*-based decision procedure [2] for the first-order problem that results from the translation to Σ_{XML}. Even so, the presence of disjunctions in the constraints requires an extension of the classical technique (shown in [8]. See also the result given for a restricted kind of disjunctive dependencies in [12]).

Σ_{XML} consists of the relational schema ($\mathtt{root}, \mathtt{el}, \mathtt{child}, \mathtt{desc}, \mathtt{tag}, \mathtt{attr}, \mathtt{id}, \mathtt{text}$) and of a set of first-order constraints outlined below. The "intended" meaning of the relational symbols in Σ_{XML} is the following. The constant root denotes the root of the XML document, and the unary relation el is the set of its elements. child and desc are subsets of $\mathtt{el} \times \mathtt{el}$ and they say that their second component is a child, respectively a descendant of the first component. $\mathtt{tag} \subseteq \mathtt{el} \times \mathit{string}$ associates the tag in the second component to the element in the first. $\mathtt{attr} \subseteq \mathtt{el} \times \mathit{string} \times \mathit{string}$ gives the element, attribute name and attribute value in its first, second, respectively third component. $\mathtt{id} \subseteq \mathit{string} \times \mathtt{el}$ associates the element in the second component to a string attribute in the first that uniquely identifies it (if DTD-specified ID-type attributes exist, their values can be used for this). $\mathtt{text} \subseteq \mathtt{el} \times \mathit{string}$ associates to the element in its first component the string in its second component. Some (but not all!) of this intended meaning is captured by the following set Σ_{XML} of first-order constraints

(base)	$\forall x, y\ [\ \mathtt{child}(x,y) \rightarrow \mathtt{desc}(x,y)\ ]$	(oneTag)	$\forall x, t_1, t_2\ [\ \mathtt{tag}(x,t_1) \wedge \mathtt{tag}(x,t_2) \rightarrow t_1 = t_2\ ]$
(trans)	$\forall x, y, z\ [\ \mathtt{desc}(x,y) \wedge \mathtt{desc}(y,z) \rightarrow \mathtt{desc}(x,z)\ ]$	(id)	$\forall s, e_1, e_2\ [\ \mathtt{id}(s,e_1) \wedge \mathtt{id}(s,e_2) \rightarrow e_1 = e_2\ ]$
		(noLoop)	$\forall x, y\ [\ \mathtt{desc}(x,y) \wedge \mathtt{desc}(y,x) \rightarrow x = y\ ]$
(refl)	$\forall x\ [\ \mathtt{el}(x) \rightarrow \mathtt{desc}(x,x)\ ]$	(oneParent)	$\forall x, y, z\ [\ \mathtt{child}(x,z) \wedge \mathtt{child}(y,z) \rightarrow x = y\ ]$
(el_c)	$\forall x, y\ [\ \mathtt{child}(x,y) \rightarrow \mathtt{el}(x) \wedge \mathtt{el}(y)\ ]$	(noShare)	$\forall x, y, u, v\ [\ \mathtt{child}(x,u) \wedge \mathtt{child}(x,v) \wedge \mathtt{desc}(u,y) \wedge \mathtt{desc}(v,y) \rightarrow u = v\ ]$
(el_d)	$\forall x, y\ [\ \mathtt{desc}(x,y) \rightarrow \mathtt{el}(x) \wedge \mathtt{el}(y)\ ]$		
(el_{id})	$\forall s, x\ [\ \mathtt{id}(s,x) \rightarrow \mathtt{el}(x)\ ]$	(oneRoot)	$\forall x\ [\ \mathtt{desc}(x, \mathsf{root}) \rightarrow x = \mathsf{root}]$
(el_r)	$\mathtt{el}(\mathsf{root})$		
(line)	$\forall x, y, u\ [\ \mathtt{desc}(x,u) \wedge \mathtt{desc}(y,u) \rightarrow x = y \vee \mathtt{desc}(x,y) \vee \mathtt{desc}(y,x)]$		

Observe that (base), (trans), (refl) above only guarantee that desc *contains* its intended interpretation, namely the reflexive, transitive closure of the child relation. There are many models satisfying these constraints, in which desc is interpreted as a proper superset of its intended interpretation, and it is well-known that we have no way of ruling them out using first-order constraints. The fact that we can nevertheless use the constraints in Σ_{XML} and classical relational (therefore first-order) techniques for deciding containment under constraints comes therefore as a pleasant surprise.

DEDs. Note that except for *(line)*, all constraints in Σ_{XML} are *embedded dependencies* (as [1] calls them, but also known as tuple- and equality-generating dependencies [2]) for which a deep and rich theory has been developed. *(line)* contains disjunction but so do the XPath expressions, implicitly, via the | operator. Extending the theory to what we will call *disjunctive embedded dependencies* (DEDs) is fairly straightforward as suggested already in [2]. We show this extension in [8] where DEDs are defined exactly like the `SXIC`s in formula (1) but with relational atoms instead of XPath atoms. The main difference to the classical chase is that, instead of a chase *sequence*, the rewrite yields a chase *tree*, whose leaves are conjunctive queries to which no chase step with DED from the set D applies.

We translate regular XPath expressions into unions of relational conjunctive queries over the schema of Σ_{XML}. This translation is performed by first translating away the disjunction (| in paths, `or` in qualifiers), thus obtaining a union of simple, disjunction-free XPath expressions: $/(son|daughter)$ translates to $/son \cup /daughter$. Next, we translate these XPath expressions according to the operator $\mathcal{T}(c, p, s)$ defined below. It takes a variable denoting the context node c, a disjunction-free XPath (sub)expression p and a variable s denoting a node in the node set yielded by p, and returns the body of a relational conjunctive query. z, u below denote fresh variables.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{T}(x, /p, y) &=& \mathcal{T}(\mathsf{root}, p, y) & \mathcal{T}(x, \texttt{ancestor-or-self}, y) &=& \{\texttt{desc}(y, x)\} \\
\mathcal{T}(x, //p, y) &=& \{\texttt{desc}(\mathsf{root}, z)\} \cup \mathcal{T}(z, p, y) & \mathcal{T}(x, \texttt{ancestor}, y) &=& \{\texttt{child}(x, z), \texttt{desc}(y, z)\} \\
\mathcal{T}(x, p_1/p_2, y) &=& \mathcal{T}(x, p_1, z) \cup \mathcal{T}(z, p_2, y) & \mathcal{T}(x, *, y) &=& \{\texttt{child}(x, y)\} \\
\mathcal{T}(x, p_1//p_2, y) &=& \mathcal{T}(x, p_1, z) \cup \{\texttt{desc}(z, u)\} \cup \mathcal{T}(u, p_2, y) & \mathcal{T}(x, .., y) &=& \{\texttt{child}(y, x)\} \\
\mathcal{T}(x, p[q], y) &=& \mathcal{T}(x, p, y) \cup \mathcal{Q}(y, q) & \mathcal{T}(x, \$v := @n, y) &=& \{\texttt{attr}(x, ''n'', v), y = v\} \\
\mathcal{T}(x, ., y) &=& \{x = y\} & \mathcal{Q}(x, q_1 \texttt{ and } q_2) &=& \mathcal{Q}(x, q_1) \cup \mathcal{Q}(x, q_2) \\
\mathcal{T}(x, n, y) &=& \{\texttt{child}(x, y), \texttt{tag}(y, n)\} & \mathcal{Q}(x, p) &=& \mathcal{T}(x, p, z) \\
\mathcal{T}(x, @n, y) &=& \{\texttt{attr}(x, ''n'', y)\} & \mathcal{Q}(x, p = s) &=& \mathcal{T}(x, p, s) \\
\mathcal{T}(x, @*, y) &=& \{\texttt{attr}(x, z, y)\} & \mathcal{Q}(x, @n = \$v) &=& \{\texttt{attr}(x, ''n'', v)\} \\
\mathcal{T}(x, \texttt{text}(), y) &=& \{\texttt{text}(x, y)\} & \mathcal{Q}(x, \$v_1 = \$v_2) &=& \{v_1 = v_2\} \\
\mathcal{T}(x, \texttt{id}(p), y) &=& \mathcal{T}(x, p, z) \cup \{\texttt{id}(z, y)\} & \mathcal{Q}(x, p_1 = p_2) &=& \mathcal{T}(x, p_1, z) \cup \mathcal{T}(x, p_2, z) \\
\mathcal{T}(x, \texttt{id}(s), y) &=& \{\texttt{id}(s, y)\} & & &
\end{array}
$$

It is not hard to see that this translation captures exactly the formal semantics in [18] over models in which `desc` has the intended interpretation.

Example translation. Recalling our regular XPath example P from section 1, we first translate away the disjunction obtaining $P_1 \cup P_2$, where $P_1 = //a/c[@m = "0"$ `and` $.//d$ `and` $. = //.[@o]//.]/@n$ and $P_2 = //b/c[@m = "0"$ `and` $.//d$ `and` $. = //.[@o]//.]/@n$. Next, we translate P_1, P_2 according to $\mathcal{T}()$. For example, P_1 translates to

$$
\begin{array}{rcl}
P_1'(x) &\leftarrow& \texttt{desc}(\mathsf{root}, u_1)\ ,\ \texttt{tag}(u_1, "a"), \texttt{child}(u_1, u_2)\ ,\ \texttt{tag}(u_2, "c"),\ \texttt{attr}(u_2, "m", "0"), \\
&& \texttt{desc}(u_2, u_3),\ \texttt{tag}(u_3, "d"),\ \texttt{desc}(\mathsf{root}, u_4),\ \texttt{attr}(u_4, "o", u_5),\ \texttt{desc}(u_4, u_2),\ \texttt{attr}(u_2, "n", x)
\end{array}
$$

where the equalities of variables (of the form $w = v$) obtained during the translation were eliminated (by substituting w for v everywhere).

`SXIC` **Translation.** Combining the $\mathcal{T}()$-translation of the XPath atoms shown above with a straightforward translation of logical connectives and quantifiers, we translate SXICs into disjunctive embedded dependencies (DEDs) over the schema of Σ_{XML} (see [8]).

Example for deciding containment of simple XPaths. We highlight here how we deal with the // operator. Given $q_1 = /A/B$ and $q_2 = //B//.$, it is easy to see that q_1 is contained in q_2 over all XML documents (i.e. even in the absence of SXICs). We show how we infer this using the classical result on the chase: given conjunctive queries Q_1, Q_2 and relational dependencies D, Q_1 is contained in Q_2 under all D-instances if and only if there is a containment mapping from Q_2 into the result of chasing Q_1 with D [1]. The translation yields $q_1'(x) \leftarrow \texttt{child}(\mathsf{root}, x_1), \texttt{tag}(x_1, A), \texttt{child}(x_1, x), \texttt{tag}(x, B)$ and $q_2'(y) \leftarrow \texttt{desc}(\mathsf{root}, y_1), \texttt{child}(y_1, y_2)$, $\texttt{tag}(y_2, B)$, $\texttt{desc}(y_2, y)$. Note that there is no containment mapping from q_2' to q_1' as the latter contains no desc-atoms to serve as image for the former's desc-atoms. But by chasing q_1' with (base),($\texttt{el}_c$),(refl) we add $\texttt{desc}(\mathsf{root}, x_1), \texttt{el}(x_1), \texttt{el}(x), \texttt{desc}(x, x)$ to q_1', thus creating an image for the containment mapping $\{y \mapsto x, y_1 \mapsto x_1, y_2 \mapsto x\}$. There are further applicable chase steps, omitted here as they only add new atoms and hence do not affect the existence of the containment mapping. •

[8] shows how we extend this kind of reasoning to deciding containment for the extensions of simple XPath mentioned in section 3.

5 Extensions and further work

Order. Our decision procedure for containment extends straightforwardly if we add the `preceding-sibling` and `following-sibling` navigation steps to the fragments of XPath we show in section 3, and the complexity results carry over to this extension. All we need to do is capture the `preceding-sibling` relation with first-order statements such as (see [8] for details):

$$(\text{trans}_{ps}) \quad \forall x, y, z \; [\, \texttt{preceding-sibling}(x, y) \wedge \texttt{preceding-sibling}(y, z) \rightarrow \texttt{preceding-sibling}(x, z)]$$
$$(\text{min}_{ps}) \quad \forall x, y \; [\, \texttt{preceding-sibling}(x, y) \rightarrow \exists z \; \texttt{child}(z, x) \wedge \texttt{child}(z, y)]$$

If the XPath expressions contain `following` and `preceding` as well, our algorithm remains *sound*, but we do not know if it is complete for deciding containment.

What we do not capture. The order-related features we do not capture in this way are *index* and *range qualifiers*. The expression $/a[2]$ uses the index qualifier 2 to return the second a-child of the root. $/a[\texttt{range 2 to 4}]$ returns the second, third and fourth a-child.

Other open problems. In addition to what we pointed out above, we have the containment of full-fledged XPath expressions, both under the set semantics given in XPath 1.0, and the list semantics coming up in XPath 2.0.

Another, maybe more important problem is that of extending optimization of XPath expressions to optimization of XQueries [16]. The latter lets variables range over node sets defined by XPath expressions. Two extensions are needed here: the output of XQueries is not a node set, but rather full XML. Also, XQueries have list semantics.

References

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

[2] C. Beeri and M. Y. Vardi. A proof procedure for data dependencies. *Journal of the ACM*, 31(4):718–741, 1984.

[3] P. Buneman, S. Davidson, W. Fan, C. Hara, and W.-C. Tan. Keys for xml. In *WWW10*, 2001.

[4] D. Calvanese, G. De Giacomo, and M. Lenzerini. Queries and constraints on semi-structured data. In *CAiSE*, pages 434–438, 1999.

[5] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Vardi. Containment of conjunctive regular path queries with inverse. In *KR*, 2000.

[6] A. K. Chandra, H. R. Lewis, and J. A. Makowsky. Embedded implicational dependencies and their inference problem. In *Proceedings of STOC*, pages 342–354, 1981.

[7] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A Query Language for XML. In *WWW8*, 1999.

[8] A. Deutsch and V. Tannen. Containment of Classes of XPath Expressions Under Integrity Constraints. Technical Report MS-CIS-01-21, University of Pennsylvania, 2001. Available from `http://db.cis.upenn.edu/cgi-bin/Person.perl?adeutsch`

[9] W. Fan and L. Libkin. On XML Constraints in the Presence of DTDs. In *PODS, 2001.*

[10] W. Fan and J. Siméon. Integrity Constraints for XML. In *SIGMOD, 2000.*

[11] D. Florescu, A. Levy, and D. Suciu. Query containment for conjunctive queries with regular expressions. In *PODS, 1998.*

[12] G. Grahne and A. Mendelzon. Tableau techniques for querying information sources through global schemas. In *ICDT, 1999.*

[13] W3C. Extensible Markup Language (XML) 1.0. W3C Recommendation 10-February-1998. Available from `http://www.w3.org/TR/1998/REC-xml-19980210`.

[14] W3C. XML Path Language (XPath) 1.0. W3C Recommendation 16 November 1999. Available from `http://www.w3.org/TR/xpath`.

[15] W3C. XML Schema Part 0: Primer. Working Draft 25 February 2000. Available from `http://www.w3.org/TR/xmlschema-0`.

[16] W3C. XQuery: A query Language for XML. W3C Working Draft 15 February 2001. Available from `http://www.w3.org/TR/xquery`.

[17] W3C. XSL Transformations (XSLT) Version 1.0. W3C Recommendation 16 November 1999. Available from `http://www.w3.org/TR/xslt`.

[18] Phil Wadler. A Formal Semantics of Patterns in XSLT. In *Proceeding of the Conference for Markup Technologies*, 1999.