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 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

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

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 [17], to recent work on keys [3] and database-style constraints [10, 9] and the current XML Schema standardization effort [19].

Several of these formalisms build on the XPath standard [18] or on closely related languages. XPath is also central to XML transformation and query language standards (XSLT [21], respectively XQuery [20]). 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

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

Meaning of XPath expressions. To describe the meaning of this notation recall that an XML document represents a rooted tree whose nodes include *elements* and *attributes* [17]. 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 [18, 22]. Additional operators are described below.

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

$$\begin{aligned}
(oneAddress) \quad & \forall x \forall s_1 \forall s_2 \, [//person \ x \ \wedge x \ ./address \ s_1 \wedge x \ ./address \ s_2 \ \rightarrow s_1 = s_2] \\
(someAddress) \quad & \forall x \, [//person \ x \ \rightarrow \ \exists y \ x \ ./address \ y] \\
(idref) \quad & \forall x \, [//person/@spouse \ x \ \rightarrow \exists y \ //. \ y \wedge y \ ./@* \ x] \\
(key_{s,p}) \quad & \forall x, y, s \, [//person \ x \ \wedge //person \ y \wedge x \ ./@ssn \ s \wedge y \ ./@ssn \ s \rightarrow x = y] \\
(grandpa) \quad & \forall x, y, z, u \, [x \ ./(son|daughter) \ y \wedge y \ ./(son|daughter) \ z \wedge x \ ./@ssn \ u \rightarrow z \ ./@grandparent \ u]
\end{aligned}$$

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):

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

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 $\texttt{For}\ /a[@n = \$v]//b[@m = \$v]\ \ \$x$ is equivalent to $\texttt{For}\ \ /a/@n\ \ \$v,\ \$v//b\ \$x,\ \$x/@m\ \$w,\ \ \texttt{Where}\ \$v = \$w$.

The most notable restrictions to the full-fledged standard [18] 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 7, 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.

[1] Note that this view is actually consistent with the XPath 1.0 specification [18], 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.

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 SXICs. We have seen that some of them are not expressible by DTDs, while others are implied by them. But in general DTDs and SXICs are incomparable. DTD features that cannot be expressed by SXICs 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 SXICs, 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 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* SXICs *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* SXICs *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* SXICs*, and (2) in the presence of bounded* SXICs *and DTDs.*

Bounded SXICs 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 $p ::= \texttt{parent}$ to the grammar in section 2.

Ancestor axes. We allow navigation along the ancestor and ancestor-or-self axis. The corresponding productions are $p ::= \texttt{ancestor} \mid \texttt{ancestor-or-self}$.

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 [18].

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 4). (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 ($\text{key}_{s,p}$) does not. The intuition behind all three restrictions is that

no combination of tree SXICs 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 SXICs *is in* Π_2^p *in the expression size.*

4 Decision Technique: First-Order Translation

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 SXICs 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 simple XPath expressions under C to containment of unions of relational conjunctive queries under $\Sigma_{\text{XML}} \cup \Sigma_C$.

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 (root, `el`, `child`, `desc`, `tag`, `attr`, `id`, `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 `el` $\times$ `el` and they say that their second component is a child, respectively a descendant of the first component. `tag` $\subseteq$ `el` $\times$ *string* associates the tag in the second component to the element in the first. `attr` $\subseteq$ `el` $\times$ *string* $\times$ *string* gives the element, attribute name and attribute value in its first, second, respectively third component. `id` $\subseteq$ *string* $\times$ `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). `text` $\subseteq$ `el` $\times$ *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)	$\forall x, y\ [\ \texttt{child}(x, y) \rightarrow \texttt{desc}(x, y)\ ]$	(oneTag)	$\forall x, t_1, t_2\ [\ \texttt{tag}(x, t_1) \wedge \texttt{tag}(x, t_2) \rightarrow t_1 = t_2\ ]$
(trans)	$\forall x, y, z\ [\ \texttt{desc}(x, y) \wedge \texttt{desc}(y, z) \rightarrow \texttt{desc}(x, z)\ ]$	(id)	$\forall s, e_1, e_2\ [\ \texttt{id}(s, e_1) \wedge \texttt{id}(s, e_2) \rightarrow e_1 = e_2\ ]$
		(noLoop)	$\forall x, y\ [\ \texttt{desc}(x, y) \wedge \texttt{desc}(y, x) \rightarrow x = y\ ]$
(refl)	$\forall x\ [\ \texttt{el}(x) \rightarrow \texttt{desc}(x, x)\ ]$	(oneParent)	$\forall x, y, z\ [\ \texttt{child}(x, z) \wedge \texttt{child}(y, z) \rightarrow x = y\ ]$
(el_c)	$\forall x, y\ [\ \texttt{child}(x, y) \rightarrow \texttt{el}(x) \wedge \texttt{el}(y)\ ]$	(noShare)	$\forall x, y, u, v\ [\ \texttt{child}(x, u) \wedge \texttt{child}(x, v) \wedge \texttt{desc}(u, y) \wedge \texttt{desc}(v, y) \rightarrow u = v\ ]$
(el_d)	$\forall x, y\ [\ \texttt{desc}(x, y) \rightarrow \texttt{el}(x) \wedge \texttt{el}(y)\ ]$		
(el_{id})	$\forall s, x\ [\ \texttt{id}(s, x) \rightarrow \texttt{el}(x)\ ]$	(oneRoot)	$\forall x\ [\ \texttt{desc}(x, \texttt{root}) \rightarrow x = \texttt{root}]$
(el_r)	$\texttt{el}(\texttt{root})$		
(line)	$\forall x, y, u\ [\ \texttt{desc}(x, u) \wedge \texttt{desc}(y, u) \rightarrow x = y \vee \texttt{desc}(x, y) \vee \texttt{desc}(y, x)]$		

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 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.

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

It is not hard to see that this translation captures exactly the formal semantics in [22] over models in which `desc` has the intended interpretation.

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

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

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 `SXIC`s into disjunctive embedded dependencies (DEDs) over the schema of Σ_{XML} (see appendix A).

Section 5 shows in detail how we use this reduction to containment of relational queries under first-order dependencies to decide containment of simple XPath expressions, and section 6 shows how we extend this kind of reasoning to handle containment for the extensions of simple XPath mentioned in section 3.

5 Detailed Treatment for Simple XPath

5.1 Upper Bounds

We first point out a major difference between containment for simple XPaths and `SXIC`s versus containment for conjunctive queries and DEDs. In the absence of constants in the language, given arbitrary conjunctive query p and set of DEDs D, there is always a relational database satisfying D, on which p returns a non-empty answer. We say that p is *D-satisfiable*. This is not always the case if p is a simple XPath expression and D a set of `SXIC`s. As an example, let p be $//person[@ssn = ./loves/person/@ssn]$ returning persons who love persons of same social security number (in particular, p may return narcissistically inclined individuals). Let D contain the key constraint $(key_{s,p})$ on the ssn-attribute of $person$-elements (shown in section 1). p is D-unsatisfiable (it returns the empty answer on all documents satisfying D) because any XML document satisfying the key constraint cannot nest a $person$-element e

in a *person*-element that agrees with e on the *ssn*-attribute: this would amount to nesting e within itself. We need to detect D-unsatisfiable XPaths, because they are vacuously contained in any other XPath under D. If constants are present in the language, unsatisfiability can occur even in the relational case, from equality tests between distinct constants. Our decision procedure is given in theorem 5.1 below. Conditions (1) and (2) are used precisely to handle D-unsatisfiability.

Theorem 5.1 *Given simple XPaths p_i ($i = 1, 2$) and the set C of* bounded `SXICs`*, let p_i be translated to the union of conjunctive queries $Q_1^i, \ldots, Q_{n_i}^i$, and let Σ_C denote the result of C's translation to a set of DEDs according to $\mathcal{T}()$. Then the following are true*

- *The chase of Q_j^1 with $\Sigma_{XML} \cup \Sigma_C$ terminates for every $1 \leq j \leq n_1$ and, the depth of the chase tree is polynomial in the size of Q_j^1 and exponential in that of the constraints in Σ_C. Denote the leaves with $\{L_1, \ldots, L_m\} = \bigcup_{j=1}^{n_1} chase_{\Sigma_{XML} \cup \Sigma_C}(Q_j^1)$.*
- *p_1 is contained in p_2 under C if and only if for every $1 \leq i \leq m$ either*
  (1) there is a homomorphism from the formula $\texttt{child}(x', y') \wedge \texttt{desc}(y', x')$ into L_i, or
  (2) the equality of distinct string constants s_1, s_2 is implied by the equalities in L_i, or
  (3) there is a $1 \leq j \leq n_2$ and a containment mapping from Q_j^2 into L_i.

We omit the proof (it uses theorem A.1), but we give the intuition behind the conditions (1),(2),(3) above. (1) detects queries L_i which test the existence of a nontrivial cycle in the XML document, thus being unsatisfiable. (1) is obviously PTIME-checkable. (2) detects queries that contain unsatisfiable tests (they could result from testing for elements with two distinct tags, or with two non-IDREFS attributes of same name but distinct values). It can be checked in PTIME by checking the membership of (s_1, s_2) in the symmetric, reflexive, transitive closure of the equality conditions of L_i. There are conceivably other reasons for L_i's unsatsifiability (e.g. a test for two distinct paths leading to the same node). It turns out however that, no matter what the reasons are, one of the conditions (1) or (2) must apply, as a result of chasing with the DEDs $(noLoop), (oneParent), (noShare), (line)$ from Σ_{XML}. Therefore, if none of (1),(2) applies, L_i is satisfiable and (3) turns out to be equivalent to containment in Q_j^2.

Example: Containment by condition (1). The simple XPath expression p above is shown to be $\{(key_{s,p})\}$-unsatisfiable as follows. Let $\mathcal{T}(p) = p'$ where $p'(y) \leftarrow \texttt{desc}(\texttt{root}, x), \texttt{child}(x, y), \texttt{tag}(y, person), \texttt{attr}(y, ssn, z), \texttt{child}(y, u), \texttt{tag}(u, loves), \texttt{child}(u, v), \texttt{tag}(v, person), \texttt{attr}(v, ssn, z)$.

By chasing p' in order with `(key),(oneParent),(base)`, we obtain a query p'' that extends p' with the atoms $y = v, x = u, \texttt{desc}(u, v)$ respectively. Note that condition (1) applies now, as witnessed by the homomorphism $h = \{x' \mapsto y, y' \mapsto u\}$. The chase continues since more steps are applicable, but they cannot affect the existence of h, as they only add atoms to p''. •

Example: Containment by condition (3). 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 if $C = \emptyset$). We show how we infer this using condition (3). The translation yields $q_1'(x) \leftarrow \texttt{child}(\texttt{root}, x_1), \texttt{tag}(x_1, A), \texttt{child}(x_1, x), \texttt{tag}(x, B)$ and $q_2'(y) \leftarrow \texttt{desc}(\texttt{root}, y_1), \texttt{child}(y_1, y_2), \texttt{tag}(y_2, B), \texttt{desc}(y_2, y)$. 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 $\texttt{desc}(\texttt{root}, x_1), \texttt{el}(x_1), \texttt{el}(x), \texttt{desc}(x, x)$ 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. •

The upper bounds for containment given in theorem 2.1 follow as a corollary of theorem 5.1.

Proof of Theorem 2.1: (1) We prove equivalently that non-containment is in Σ_2^p, that is it is decidable by an NP machine with an NP oracle. In the notation of theorem 5.1, the machine guesses Q_i, then the root-leaf path in the chase tree of Q_i leading to some L_j as follows. The necessary space is polynomial in the size of p_1 and the maximum size of a ded in Σ_C: for every step of the root-leaf path in the chase tree, the machine guesses the ded d that applies, the homomorphism h from d's left-hand side of the implication, and the disjunct (in d's right-hand side of the implication) which is used to chase on this particular path. This information is sufficient to check in PTIME (in the size of d) whether the guessed step corresponds to a chase step. Then the machine uses the oracle to check that this chase step is indeed applicable (it must ask whether there is an extension of h to any of d's disjuncts). At every step, the machine asks the oracle if further chase steps apply and goes on to guessing the next step if the answer is "yes". The oracle is guaranteed to answer "no" after polynomially many invocations (in the size of Q_i!), due to the first item in theorem 5.1.

Once the leaf L_j is guessed, the machine checks conditions (1) and (2) in PTIME (in the size of L_j which is polynomial in that of Q_i, hence also in that of p_1) and answers "yes" if any of them is true. Otherwise, it checks condition (3) by asking the oracle (this can be checked in NP in the maximum size of a ded, as finding containment mappings is in NP). The machine answers "yes" if and only if the oracle answers "no".

(2) Note that in the absence of disjunction of any kind, p_1 is translated to a single conjunctive query, Q_1^1. The chase tree degenerates into a single root-leaf path, because there is no disjunction in Σ_C and because the absence of the element equality tests and `ancestor` and `ancestor-or-self` navigation steps guarantees that (line) in Σ_{XML} never applies. This single root-leaf path corresponds to a standard chase *sequence*, whose result is a conjunctive query we denote with L_1. By the first item of theorem 5.1, the number of steps in this chase sequence is polynomial in the size of Q_1^1. For each step in the sequence, the machine must guess a homomorphism from some dependency $d \in \Sigma_C \cup \Sigma_{\text{XML}}$, which is polynomial in the size of d. Once the chase sequence has been guessed, the machine checks conditions (1) and (2) from theorem 5.1 in PTIME, and if none is satisfied, it guesses a containment mapping from Q_1^2 into the chase result (polynomial in the size of Q_1^2).

(3) As in (2), the absence of disjunction ensures that the paths are translated to the single conjunctive queries Q_1^1, Q_1^2, and, together with the absence of equality tests, this ensures that the chase of Q_1^1 degenerates to a sequence. The chase result L_1 is polynomial in the size of Q_1^1.

Let $I(L_1)$ be a Σ_{XML}-instance obtained from L_1 such that (i) `el` consists of all variables and constants in L_1, (ii) the entries in `child`, `attr`, `tag`, `text`, `id` are the corresponding atoms in L_1's body, and (iii) `desc` is the *minimal* relation closed under $(base), (trans), (refl)$. It is easy to see that $I(L_1)$ can be computed in PTIME in the size of L_1. It is also easy to show that there is a containment mapping from Q_1^2 into L_1 if and only if Q_1^1's head variable belongs to the result of evaluating Q_1^2 on $I(L_1)$. But the latter evaluation can be performed in PTIME in both the size of $I(L_1)$ (hence L_1) and of Q_1^2. This is because the absence of equality tests makes Q_2 an *acyclic* query, for which Yannakakis shows PTIME evaluation (in the combined expression and data complexity) [1].

We therefore only need to guess the homomorphisms for the chase steps, which can be done in NP in the size of the dependencies. But in the absence of (or if we fix) the `SXICs` in C, we can find any homomorphism from a dependency d in PTIME in the size of Q_1^1 by simply trying all mappings (their number is exponential only in the size of d). •

Remarks. In practice the decision procedure from theorem 5.1 is typically invoked repeatedly to check containment under the *same* set C of bounded SXICs. In this scenario, we can consider C fixed, in which case the complexity bounds in the theorem are only in the size of the simple XPath expressions. In particular, if $C = \emptyset$ (there are no integrity constraints), we obtain upper bounds for containment over *all* XML documents.

Note that if we disallow disjunction, containment is in NP, and thus no harder than for relational conjunctive queries. We will see in section 6 that this situation changes for extensions of simple XPath expressions: adding navigation to wildcard children or to ancestors raises complexity of containment to Π_2^p-hard (theorem 3.1) even in the absence of disjunction!

5.2 Undecidability

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 result in theorem 2.2 showing undecidability of containment :

Proof of theorem 2.2: By reduction from the following undecidable problem: Given context-free grammar $G = (\Sigma, N, S, P)$ where Σ is the set of terminals (containing at least two symbols), N the nonterminals, $S \in N$ the start symbol, $P \subseteq N \times (\Sigma \cup N)^*$ the productions, and $L(G)$ the language generated by G, the question whether $L(G) = \Sigma^*$ is undecidable [13].

Note. For the sake of presentation simplicity, the reduction we show below is to containment in the presence of bounded SXICs and DTDs. However, a careful analysis of the used DTD features reveals that these are captured as SXICs of two forms: $\forall x\ [//A\ x \rightarrow (\exists y\ x\ ./A\ y) \vee (\exists y\ x\ ./B\ y)]$ and $\forall x\ [//A\ x \rightarrow (\exists y\ x\ ./@s\ y)]$. These are not bounded SXICs: note the illegal existential quantification of y and recall that the definition allows at most the quantification of x, and only in the $x\ .@s\ y$ atom.

The reduction. Given context-free grammar $G = (\Sigma, N, S, P)$, we construct an instance (DTD_G, D_G, $XP_1 \subseteq XP_2$) such that XP_1 is contained in XP_2 over all XML documents conforming to the description DTD_G and satisfying the dependencies in D_G if and only if $\Sigma^* \subseteq L(G)$. We first show DTD_G, which does not exercise all features of DTDs. The features of DTD_G used to prove undecidability can be easily shown to be fully captured by SXICs:

```
<!ELEMENT B  (A|E)>      <!ATTLIST B             <!ATTLIST A
<!ELEMENT A  (A|E)>               i  #ID,                 i   #ID,
<!ELEMENT E  (PCDATA)>            S  #IDREFS>             sym (a1|a2|...|an),
                                                          N1  #IDREFS,
                                                          ...
                                                          Nk  #IDREFS>
```

B,E,A are fresh names, $a_1, \ldots, a_n$ are the alphabet symbols in Σ, $N_1, \ldots, N_k$ are the nonterminals in N. Every document conforming to DTD_G is a list (unary tree) of elements, whose head is tagged B and unique leaf tagged E. The inner elements (if any) of the list are tagged A, and their *sym* attribute contains a symbol of Σ. Every document conforming to DTD_G thus corresponds to a word $w \in \Sigma^*$, and every pair s, t of A-elements such that t is a descendant of s determines a substring of w.

The set of dependencies D_G (shown shortly) is designed such that, whenever a document conforms to the DTD_G and satisfies D_G, the following *claim* holds: for every pair s, t of A-elements with t a descendant of s, let u be the corresponding substring of w (if $s = t$, u is the unit length string given by the value of t's sym attribute). Then for every $1 \leq j \leq k$ such that

there is a derivation of u starting from nonterminal N_j, the value of the attribute i of t is a token of the value of the N_j attribute of s [2]. Furthermore, the S attribute of the B-element contains all tokens of the S attribute of the first A-element, if any.

We omit the proof of the claim, but illustrate for the grammar $S \to cS \mid cc$ and word $w = ccc$. An XML document corresponding to w which conforms to DTD_G and satisfies the claim is

```
<B i=''0'' S=''2 3''>
  <A sym=''c'' i=''1'' S=''2 3''>  <A sym=''c'' i=''2'' S=''3''>  <A sym=''c'' i=''3'' S=''''>
<E>any text goes here</E></A></A></A></B>
```

Now we have $w \in L(G)$ if and only if there is a derivation of w in G starting from S, which by the claim is equivalent to the i-attribute in the parent of the E-element being among the tokens of the S-attribute in the B-element. Therefore, $\Sigma^* \subseteq L(G)$ is equivalent to the containment

$$//.[/\mathtt{E}]/@i \ \subseteq /\mathtt{B}/@S$$

which we pick for $XP_1 \subseteq XP_2$.

We now show the dependencies D_G. For every production $p \in P$, we construct a dependency $(prod_p)$ as illustrated by the following example. Let R, T be nonterminals and a, b alphabet symbols in the production $R \to aRbT$. The corresponding dependency is

$$(prod_p) \quad \forall x, y \, [\, x \ ./S[@sym = "a"]/\mathtt{id}(@R)/S[@sym = "b"]/\mathtt{id}(@T)/@i \ \ y \ \to \ x \ ./@R \ \ y \,]$$

We enforce that the tokens in the S-attribute of the first A-element be included in the S-attribute of the B-element with the SXIC

$$(start_{\mathtt{B}}) \quad \forall x, y \, [\, /\mathtt{B} \ x \ \ \wedge \ \ x \ ./\mathtt{A}/@S \ \ y \ \ \to \ \ x \ ./@S \ \ y \,]$$

Furthermore, we may assume without loss of generality that G has at most one ϵ-production, namely $S \to \epsilon$ (see the procedure for elimination of ϵ-productions employed when bringing a grammar in Chomsky Normal Form [13]). If $S \to \epsilon \in P$, add to D_G the SXIC

$$(d_\epsilon) \quad \forall x, y \, [\, /\mathtt{B} \ x \ \ \wedge \ \ x \ ./@i \ \ y \ \ \to \ \ x \ ./@S \ \ y \,] \quad \bullet$$

Remark. The undecidability result of theorem 2.2 does not preclude us from using the procedures in theorem 5.1 and section 6 for checking containment even under arbitrary SXICs. If the chase terminates, then containment holds if and only if and any of the conditions (1),(2),(3) in theorem 5.1 are satisfied. The problem is that for arbitrary SXICs the chase may diverge. We can always impose a threshold after which we stop the chase and check the conditions. This would result in a *sound*, but *incomplete* procedure for checking containment. Our experience with the chase for the relational/OO data model [15] suggests that there are many practical cases in which the chase terminates even if the SXICs are not bounded.

6 Detailed Treatment for Extensions of Simple XPath

Note that the translation of enriched XPath expressions is compatible with that of simple XPath expressions, and the addition of the wildcard child, parent and ancestor navigation is a very natural extension, which doesn't even require new schema elements in Σ_{XML}. We chose

[2] Recall that an IDREFS attribute a models a *set* of IDREF attributes, represented by the set of whitespace-delimited tokens of a's string value.

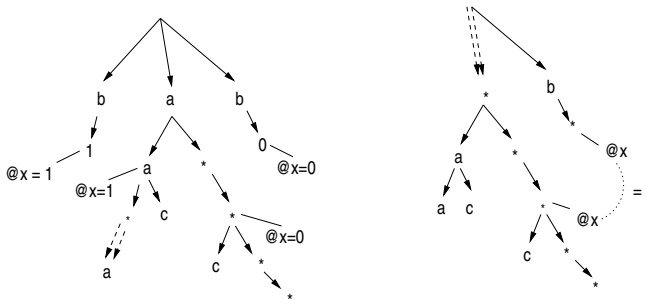

Figure 1: XPath expressions in counterexample 6.1

to handle these extensions separately because, innocuous as they may seem, they change complexity bounds dramatically. It turns out that the dependencies in Σ_{XML} become insufficient in reasoning about wildcard expressions. Here is a counterexample to theorem 5.1.

Example 6.1 There are simple XPath expressions p, p' extended with wildcard child navigation such that p is contained in p' but $\mathcal{T}(p)$ is not contained in $\mathcal{T}(p')$ under Σ_{XML}:

$$
\begin{aligned}
p \;&= /[\; b/1[@x ='' 1''] \texttt{ and} \\
&\qquad a[a[@x ='' 1'' \texttt{ and } * //a \texttt{ and } c] \texttt{ and } * / * [c \texttt{ and } * / * \texttt{ and } @x ='' 0'']] \\
&\qquad b/0[@x ='' 0'']] \\
p' \;&= /[\; .// * [a[a \texttt{ and } c] \texttt{ and} \\
&\qquad\qquad * / * [c \texttt{ and } * / * \texttt{ and } @x = /b/ * /@x]]
\end{aligned}
$$

In case the reader finds the graphical representation useful, we refer to figure 6, in which we depict child navigation steps with single arrows and descendant navigation steps with double, dashed arrows. The tag names are used to label the nodes (* is used for wildcards), and solid non-arrow lines associate attributes with nodes. $@x = 0$ indicates that the string value of the x-attributes is "0". The dotted line represents an equality condition on x-attributes.

To see that p is contained in p', observe that $a//a$ in p is equivalent to $a/a \cup a/*//a$, and hence p is equivalent to $p_1 \cup p_2$ where p_1, p_2 are obtained by replacing the subpath $a//a$ with a/a, respectively $a/*//a$ in p. But both p_1, p_2 are contained in p', as witnessed by the containment mappings matching the x-attributes in p' against the "0"-valued x-attributes of p_1, respectively the "1"-valued x-attributes of p_2.

On the other hand, according to the chase theorem [1], $\mathcal{T}(p)$ is not contained in $\mathcal{T}(p')$ under Σ_{XML} because there is no containment mapping from $\mathcal{T}(p')$ into $chase_{\Sigma_{\text{XML}}}(\mathcal{T}(p))$. Intuitively, what Σ_{XML} does not capture is the *minimality* of `desc`: it only states that the latter contains the reflexive transitive closure of `child`, but it doesn't rule out pairs of nodes that aren't reachable via `child` navigation steps. Σ_{XML}-instances containing such a pair $(s,t) \in N \times N$ are counterexamples for the containment: subpath $*//a$ in p is satisfied by the nodes r, q where $\texttt{child}(r,s), \texttt{desc}(s,q), \texttt{tag}(q,a)$ even if s has no immediate child, while $.// * /a$ in p' is not. •

It turns out however that theorem 5.1 holds if p_1, the contained wildcard Xpath expression, is //-free.

We will use this observation to extend our decision procedure to handle wildcard expressions. First, we introduce some notation. Observe that any //-free XPath expression is equivalent

to a finite union of `ancestor-or-self`-free and `ancestor`-free expressions. For instance, $/a/b/$`ancestor-or-self` is equivalent to $(/a/b \cup /a/b/.. \cup /a/b/../..)$. There is no point in instantiating the occurrence of `ancestor-or-self` to more parent navigation steps (..) since the resulting expression would be unsatisfiable, that is empty over all documents. We denote the set of `ancestor`-*free and* `ancestor-or-self`-*free* paths in this finite union with $\texttt{af}(p)$.

Proposition 6.2 *Let C be a set of tree* `SXIC`*s, let p be a //-free wildcard XPath expression, and let* $\texttt{af}(p) = \{p_1, \ldots, p_n\}$. *Then p is contained in wildcard expression p' under C if and only if both items of theorem 5.1 are satisfied when substituting p_i for p_1 and p' for p_2, for every $1 \leq i \leq n$.*

Recall that tree `SXIC`s are restricted bounded `SXIC`s, so the chase with them is defined. By proposition 6.2, the decision procedure for containment of simple XPath expressions given in theorem 5.1 can be used to decide containment of //-free wildcard XPath expressions under tree `SXIC`s.

We next show how to use proposition 6.2 to decide containment even if p contains navigation along the descendant axis. First, observe that $// = \bigcup_{0 \leq k} *^k$, where $*^k$ is short for the concatenation of k wildcard navigation steps. More generally, every wildcard XPath expression p with n occurrences of // is equivalent to an infinite union of //-free queries: denoting with $p(k_1, \ldots, k_n)$ the result of replacing the i^{th} occurrence of // in p with the concatenation of k_i wildcard navigation steps, p is equivalent to $\bigcup_{0 \leq k_1, \ldots, 0 \leq k_n} p(k_1, \ldots, k_n)$.

Therefore, the containment of p in p' reduces to checking the containment of each $p(k_1, \ldots, k_n)$ in p_2, which is done according to proposition 6.2. This still doesn't give us a decision procedure, since there are infinitely many containments to be checked. The key observation to our containment decision procedure is that it is sufficient to check the containment for only finitely many //-free queries in the union. For arbitrary p, we denote with $wts(p)$ the *wildcard tag size*, i.e. the number of $*$ navigation steps in p. For instance, $wts(//a/*/b/../c/@*) = 1$ (note that wildcard attributes $@*$ are not counted). Furthermore, we denote with $ps(p)$ the *parent size*, i.e. the number of .. navigation steps in p. Recalling that `ancestor` is syntactic sugar for $../$`ancestor-or-self`, this means we count `ancestor` navigation steps as well: $ps(//a/../\texttt{ancestor}) = 2$.

Proposition 6.3 *Let C be a set of bounded* `SXIC`*s, p_1, p_2 be wildcard XPath expressions and let $l \stackrel{\text{def}}{=} wts(p_2) + ps(p_2) + ps(p_1) + 1$. Then*

$$p_1 \subseteq_C p_2 \Leftrightarrow \bigcup_{0 \leq k_1 \leq l, \ldots, 0 \leq k_n \leq l} p_1(k_1, \ldots, k_n) \subseteq_C p_2$$

This result gives us the following decision procedure for containment of p_1 in p_2:

Step 1: We first translate away the disjunction (| and `or`), obtaining finite unions U_1, U_2 of XPaths.

Step 2: We next use proposition 6.3 to obtain from U_1 a finite union of //-free queries DF_1, which must be checked for containment in U_2.

Step 3: Containment of DF_1 in U_2 is decided using the following easy result:

Proposition 6.4 *The union of //-free wildcard XPath expressions $\bigcup_{i=1}^{n} p_i$ is contained in the union of wildcard XPath expressions $\bigcup_{j=1}^{m} p'_j$ under the bounded* `SXIC`*s C if and only if for every $1 \leq i \leq n$ there is a $1 \leq j \leq m$ such that $p_i \subseteq_C p'_j$.*

Step 4: Finally, checking each //-free p_i for containment in p'_j is done using proposition 6.2.

Given this decision procedure, the proofs of theorems 3.2 and 3.3, claiming Π_2^p upper bounds for diverse extensions of simple XPath are straightforward adaptations of the proof of theorem 5.1. We illustrate for the case of simple XPaths with wildcard child navigation under tree SXICs:

Proof of theorem 3.3: We prove equivalently that non-containment is in Σ_2^p, that is it is decidable by an NP machine with an NP oracle. By proposition 6.3, it is enough if the machine exhibits a //-free query in the finite union which is not contained in U_2. To this end, the machine guesses $p_1 \in DF_1$ and $p_2 \in U_2$, computes l, guesses $0 \leq k_1, \ldots, k_n \leq l$, guesses p in $af(p_1(k_1, \ldots, k_n))$ and next continues like in the proof of theorem 5.1. •

Given the presence of | and `or` in the fragment of wildcard XPath expressions, it is not surprising that the algorithm is assymptotically optimal (we'll see shortly that its lower bound is Π_2^p as well): [16] shows that containment of conjunctive queries with union is Π_2^p-complete. The upper bound however does not follow from [16]: the decision procedure of [16] works in the absence of dependencies, and hence must be extended to work under the ones in Σ_{XML}.

However, we prove a stronger result in theorem 3.1, showing Π_2^p-hardness even for containment of disjunction-free extensions of simple XPath.

Proof of theorem 3.1: We only show the proof for the extension with wildcard child navigation, which is the most interesting one.

The proof is by reduction from the Π_2^p-complete $\forall\exists 3 - SAT$ problem [14]: the instances of this problem are first-order sentences ϕ of general form
$\forall x_1 \ldots \forall x_n \exists y_1 \ldots \exists y_m \ \bigwedge_{i=1}^{l} C_i$, where each clause C_i is a disjunction of three literals which are any of the variables $x_1, \ldots, x_n, y_1, \ldots, y_m$ or their complements. ϕ is a "yes" instance if and only if it is valid.

For every instance ϕ, we construct the instance $p_1 \subseteq p_2$, where ϕ's variables appear as attribute and variable names, and p_1, p_2 contains occurrences of $@x_i, \$x_i$ for every $1 \leq i \leq n$, and occurrences of $@y_j, \$y_j$ for every $1 \leq j \leq m$. We use the notation $p_1(k_1, \ldots, k_n)$ introduced for proposition 6.3. The containment holds if and only if $p_1(k_1, \ldots, k_n) \subseteq p_2$ for all $0 \leq k_i$. We *claim* that the reduction is defined such that the latter holds if and only if ϕ has a satisfying assignment which makes x_i false if $k_i = 0$, and true if $k_i > 0$. This makes ϕ valid if and only if $p_1 \subseteq p_2$. The claim is proved after we give the construction.

Both p_1, p_2 return either the root of the document itself, or an empty node set: they have the form $/[q_i]$ where q_1, q_2 are qualifiers. q_1 is constructed as the conjunction of $7l + m + n$ subexpressions:

- for every clause C_i, let u_i, v_i, w_i be its variables, and $a_{i,1}, \ldots, a_{i,7}$ the seven satisfying assignments for C_i. For every $1 \leq i \leq l$ and $1 \leq j \leq 7$, q_1 contains the subexpression $C_i[@u_i = a_{i,j}(u_i), @v_i = a_{i,j}(v_i), @w_i = a_{i,j}(w_i)]$.
- for every y_j, we add the *existential gadget* $y_j[@y_j ='' 0''$ `and` $@y_j ='' 1'']$ to q_1.
- We also add n copies of a *universal gadget* (one copy for every x_i). The universal gadget (defined shortly) is denoted $U(l)$ and it is a wildcard XPath subexpression having occurrences of $@l$ for some attribute name l. For every x_i, the corresponding copy of U has $@l$ substituted with $@x_i$, denoted $U(x_i)$.

This completes the construction of q_1, up to the specification of the universal gadget. First we show the construction of q_2, which contains $l + m + n$ subexpressions:

- for every $1 \leq i \leq l$, q_2 contains $C_i[@u_i = \$u_i, @v_i = \$v_i, @w_i = \$w_i)]$ where, as before, u_i, v_i, w_i are the variables occuring in clause C_i. Note how they give both the names of the attributes and the names of variables.
- for every $1 \leq j \leq m$, q_2 contains the subexpression $y_j[@y_j = \$y_j]$.
- for every $1 \leq i \leq n$, q_2 contains a copy of a *satisfaction gadget* (defined shortly). The satisfaction gadget is denoted $S(l)$ and it is a wildcard XPath subexpression with occurences of $@l$ and $\$l$ for some l. For every x_i, q_2 contains a copy $S(x_i)$ in which $@l, \$l$ are substituted by $@x_i, \$x_i$.

We exemplify the construction so far on a $\forall\exists 2 - SAT$ instance for simplicity sake:

$$\phi = \forall x \forall y \exists z \underbrace{(x \vee \overline{y})}_{C_1} \wedge \underbrace{(y \vee z)}_{C_2}$$

$$\begin{aligned} p = /[\ & C_1[@x ='' 0'' \texttt{ and } @y ='' 0''] \texttt{ and } C_1[@x ='' 1'' \texttt{ and } @y ='' 0''] \texttt{ and } C_1[@x ='' 1'' \texttt{ and } @y ='' 1''] \texttt{ and} \\ & C_2[@y ='' 0'' \texttt{ and } @z ='' 1''] \texttt{ and } C_2[@y ='' 1'' \texttt{ and } @z ='' 0''] \texttt{ and } C_2[@y ='' 1'' \texttt{ and } @z ='' 1''] \texttt{ and} \\ & z[@z ='' 0'' \texttt{ and } @z ='' 1''] \texttt{ and} \\ & U(x) \texttt{ and } U(y)\ ] \end{aligned}$$

$$p' = /[\ C_1[@x = \$x \texttt{ and } @y = \$y] \texttt{ and } C_2[@y = \$y \texttt{ and } @z = \$z] \texttt{ and } z[@z = \$z] \texttt{ and } S(x) \texttt{ and } S(y)\ ]$$

We now specify the universal and satisfaction gadgets. Recalling counterexample 6.1, $U(l)$ is a copy of of p, with x acting as l, and $S(l)$ is a copy of p', with x acting as l.

We still have to prove the claim. According to proposition 6.3, $p \subseteq p'$ if and only if $p(k_1, \ldots, k_n) \subseteq p'$ both for $k_i = 0$ and $k_i > 0$. Recalling the discussion in counterexample 6.1, the containment mapping corresponding to $k_i = 0$ binds $\$x_i$ to "0", while that corresponding to $k_i > 0$ binds $\$x_i$ to "1". Moreover, it is easy to see that any containment mapping from p' to p corresponds to a satisfying assignment of ϕ. Therefore, $p_1 \subseteq p_2$ if and only if every truth assignment to the x_is has an extension to the y_js that satisfies all clauses of ϕ (or, equivalently, if and only if ϕ is valid). •

Remark. It is interesting to see that the Π_2^p lower bound is reached even in the absence of disjunction when both //- and $*$ navigation steps are allowed. We found this result to be counterintuitive, as the presence of // or $*$ in isolation results in NP-complete containment complexity (item (2) in theorem 2.1). It is only their *interaction* that increases complexity. The intuition behind this is the fact that // expresses disjunction implicitly in the presence of $*$: $//a$ is equivalent to $(/a) \mid (/*//a)$.

7 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. We consider the ordered XML data model, and extend our XPath fragments with order-related predicates:

$$p ::= \texttt{preceding-sibling}(p) \mid \texttt{following-sibling}(p) \mid \texttt{preceding}(p) \mid \texttt{following}(p)$$

We take the view of the XPath 1.0 specification [18], according to which an XPath expression evaluates to a node *set*, thus restricting the impact of order only to checking the predicates above (this situation will change with the XPath 2.0 specification however).

Let Σ^O_{XML} be the extensionof Σ_{XML} with the binary relations `preceding-sibling`, `preceding` and the constraints

$$
\begin{array}{rl}
(\text{trans}_{ps}) & \forall x,y,z\ [\ \texttt{preceding-sibling}(x,y) \wedge \texttt{preceding-sibling}(y,z) \rightarrow \texttt{preceding-sibling}(x,z)] \\
(\text{min}_{ps}) & \forall x,y\ [\ \texttt{preceding-sibling}(x,y) \rightarrow \exists z\ \texttt{child}(z,x) \wedge \texttt{child}(z,y)] \\
(\text{total}_{ps}) & \forall x,y,z\ [\ \texttt{child}(x,y) \wedge \texttt{child}(x,z) \rightarrow y = z \vee \texttt{preceding-sibling}(y,z) \vee \texttt{preceding-sibling}(z,y)] \\
(\text{base}_{p}) & \forall x,y,z,u\ [\ \texttt{desc}(x,z) \wedge \texttt{preceding-sibling}(x,y) \wedge \texttt{desc}(y,u) \rightarrow \texttt{preceding}(y,u)] \\
(\text{trans}_{p}) & \forall x,y,z\ [\ \texttt{preceding}(x,y) \wedge \texttt{preceding}(y,z) \rightarrow \texttt{preceding}(x,z)] \\
(\text{min}_{p}) & \forall x,y\ [\ \texttt{preceding}(x,y) \rightarrow \exists u,v\ \texttt{preceding-sibling}(u,v) \wedge \texttt{desc}(u,x) \wedge \texttt{desc}(v,y)] \\
(\text{total}_{p}) & \forall x,y\ [\ N(x) \wedge N(y) \rightarrow \texttt{desc}(x,y) \vee \texttt{preceding}(x,y) \vee \texttt{preceding}(y,x)]
\end{array}
$$

We provide the first-order translation

$$
\begin{array}{rcl}
\mathcal{T}(x, \texttt{preceding-sibling}(p), y) & = & \{\texttt{preceding-sibling}(x,y)\} \cup \mathcal{T}(y,p,z) \\
\mathcal{T}(x, \texttt{preceding}(p), y) & = & \{\texttt{preceding}(x,y)\} \cup \mathcal{T}(y,p,z) \\
\mathcal{T}(x, \texttt{following-sibling}(p), y) & = & \mathcal{T}(y, \texttt{preceding-sibling}(p), x) \\
\mathcal{T}(x, \texttt{following}(p), y) & = & \mathcal{T}(y, \texttt{preceding}(p), x)
\end{array}
$$

Theorem 7.1 *If we add the* `preceding-sibling` *and* `following-sibling` *predicates to the XPath fragments in section 3 and use* Σ^O_{XML} *above, the algorithm in section 6 remains a decision procedure for containment, and the complexity results carry over.*

If the XPath expressions contain `following` and `preceding` as well, the 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. $/a$[`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 [20]. 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] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

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

[3] Peter Buneman, Susan Davidson, Wenfei Fan, Carmem Hara, and Wang-Chiew Tan. Keys for xml. In *WWW10*, May 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 ACM SIGACT Symposium on the Theory of Computing*, pages 342–354, 1981.

[7] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for XML. In *Proc. of 8th International WWWW Conference*, 1999.

[8] Alin Deutsch and Val Tannen. Containment of Classes of XPath Expressions Under Integrity Constraints. Technical Report MS-CIS-01-21, University of Pennsylvania, 2001.

[9] Wenfei Fan and Leonid Libkin. On XML Constraints in the Presence of DTDs. In *Proceedings of PODS, May 2001, Sanata Barbara, CA, USA*. ACM, 2001.

[10] Wenfei Fan and Jérôme Siméon. Integrity Constraints for XML. In *ACM-SIGMOD, May 15-17, 2000, Dallas, Texas, USA*, pages 23–34. ACM, 2000.

[11] Daniela Florescu, Alon Y. Levy, and Dan Suciu. Query containment for conjunctive queries with regular expressions. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*. ACM Press, 1998.

[12] Gösta Grahne and Alberto O. Mendelzon. Tableau techniques for querying information sources through global schemas. In Catriel Beeri and Peter Buneman, editors, *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, volume 1540 of *Lecture Notes in Computer Science*, pages 332–347. Springer, 1999.

[13] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[14] Christo H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[15] Lucian Popa, Alin Deutsch, Arnaud Sahuguet, and Val Tannen. A Chase Too Far? In *Proceedings of ACM SIGMOD International Conference on Management of Data*, May 2000.

[16] Yehoushua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27:633–655, 1980.

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

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

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

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

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

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

A Chasing With Disjunctive Embedded Dependencies

We translate SXICs to first order logic statements by translating every path atom $v\ p\ w$ to the conjunction of all goals in $\mathcal{T}(v,p,w)$, and universally quantifying all fresh variables introduced during the translation. For example, the translation of SXIC $\mathcal{T}(key_{s,p}) = key$ above is

$$(key) \;=\; \forall x,y,s,u,v\ [\mathtt{desc}(\mathtt{root},u) \wedge \mathtt{child}(u,x) \wedge \mathtt{tag}(x,person) \wedge \mathtt{attr}(x,ssn,s) \wedge \mathtt{desc}(\mathtt{root},v) \wedge \mathtt{child}(v,y) \wedge \mathtt{tag}(y,person) \wedge \mathtt{attr}(y,ssn,s) \rightarrow x = y\ ]$$

and $\forall x\ [\ //A/@sex\ x \;\rightarrow\; x = "m" \;\vee\; x = "f"\ ]$ translates to

$$\forall x,u,v\ [\ \mathtt{desc}(\mathtt{root},u) \wedge \mathtt{child}(u,v) \wedge \mathtt{tag}(v,A) \wedge \mathtt{attr}(v,"sex",x) \rightarrow\; x = "m" \vee x = "f"\ ].$$

DEDs. We introduce a new class of relational dependencies, which expresses all dependencies in Σ_{XML} and the dependencies resulting from the translation of SXICs. Their general form is

$$\forall x_1 \ldots \forall x_n\ [\phi(x_1,\ldots,x_n) \rightarrow \bigvee_{i=1}^{l} \exists z_{i,1} \ldots \exists z_{i,k_i}\ \psi_i(x_1,\ldots,x_n,z_{i,1},\ldots,z_{i,k_i})] \tag{2}$$

where ϕ, ψ_i are conjunctions of *relational atoms* of the form $R(w_1,\ldots,w_l)$ and *equality atoms* of the form $w = w'$, where $w_1,\ldots,w_l,w,w'$ are variables. ϕ may be the empty conjunction. We call such dependencies *disjunctive embedded dependencies (DEDs)*, because they contain the classical embedded dependencies [1] as the particular case $l = 1$.

We extend the classical relational *chase* [2], which is a proof procedure for query containment under embedded dependencies. First a bit of notation:

A *homomorphism* from ϕ_1 into ϕ_2 is a mapping h from the variables of ϕ_1 into those of ϕ_2 such that (i) for every equality atom $w = w'$ in ϕ_1, $h(w) = h(w')$ follows from the equality atoms of ϕ_2 and (ii) for every relational atom $R(w_1,\ldots,w_l)$ in ϕ_1, there is an atom $R(v_1,\ldots,v_l)$ in ϕ_2 such that $v_i = h(w_i)$ follows from the equality atoms of ϕ_2. Given conjunctive queries $Q_1(x_1,\ldots,x_n) \leftarrow \phi_1(x_1,\ldots,x_n,y_1,\ldots,y_m)$ and $Q_2(u_1,\ldots,u_n) \leftarrow \phi_1(u_1,\ldots,u_n,v_1,\ldots,v_k)$ (ϕ_1, ϕ_2 are conjunctions of relational and equality atoms), a *containment mapping* from Q_1 to Q_2 is a homomorphism m from ϕ_1 to ϕ_2 such that $m(u_i) = x_i$ for $1 \leq i \leq n$.

Chase with DEDs. Let d be a DED of general form (2), Q be a conjunctive query and let h be a homomorphism from ϕ into Q. We say that the *chase step* of Q with d using h is *applicable*, if h allows no extension which is a homomorphism from $\phi \wedge \psi_i$ into Q for any $1 \leq i \leq l$. In this case, the *result* of applying this chase step is the union of queries $\bigcup_{i=1}^{l} Q_i$, where each Q_i is defined as $Q \wedge \psi_i(h(x_1),\ldots,h(x_n),f_{i,1},\ldots,f_{i,k_i})$, where the $f_{i,j}$'s are fresh variables.

For example, chasing $Q(x,y) \leftarrow a(x,y)$ with $\forall u \forall v\ [a(u,v) \rightarrow b(u,v) \vee c(u,v)]$ results in $Q_b \cup Q_c$ with $Q_b(x,y) \leftarrow a(x,y), b(x,y)$ and $Q_c(x,y) \leftarrow a(x,y), c(x,y)$.

If we continue applying chase steps to each Q_i (with DEDs from a set D), we build a *chase tree* rooted at Q, whose subtrees are the chase trees rooted at the Q_i's. The leaves of the chase tree are conjunctive queries to which no chase step with any DED from D applies. In general, the chase may diverge, thus building an infinite tree, but when it terminates, we define its *result* to be the set of leaves of the chase tree, denoted $chase_D(Q)$.

Theorem A.1 *Given conjunctive queries Q_1, Q_2 and the set D of DEDs, assume that the chase of Q_1 with D terminates. Then we have: (1) Q_1 is equivalent to the union of the leaves*

of the chase tree, and (2) Q_1 is contained in Q_2 under D if and only if there is a containment mapping from Q_2 into every leaf $L \in chase_D(Q_1)$.

The proof is omitted, but it is a straightforward generalization of the classical proof given in [2] for the case of embedded dependencies (recall these are DEDs without disjunction). In fact, we retrieve that result as a particular case of theorem A.1 by observing that in the absence of disjunction, the chase tree degenerates into what [2] calls a chase sequence, having a single leaf. Another particular case is obtained when our DEDs contain no existentials, and only equalities between a variable and a constant are allowed on the right-hand side of the implication in general form (2). Such constraints and the idea of chase tree were introduced in [12], in the context of incomplete databases.