Chasing One's Tail: XPath Containment Under Cyclic DTDs

Manizheh Montazerian
Dept. of Computer Science and Info. Systems
Birkbeck, University of London
montazerian_mahtab@yahoo.co.uk

Peter T. Wood
Dept. of Computer Science and Info. Systems
Birkbeck, University of London
ptw@dcs.bbk.ac.uk

ABSTRACT

The problem of finding subclasses of XPath queries and document type definitions (DTDs) for which containment can be tested efficiently has been much studied. Along the way, a number of constraints inferred from DTDs have been used to characterise containment in terms of the chase procedure. However, previous attempts have resulted in procedures that are non-terminating for cyclic DTDs, even when the queries include only the child and descendant operators (i.e., no predicates or wildcards). In this paper, we introduce a rewriting of such XPath queries in the presence of cyclic (but simplified) DTDs, using an operator that generalises the child and descendant operators, and show that doing so allows us to produce a complete procedure for containment using the chase and previous classes of constraints. We also characterise a set of constraints that allows for a complete chase procedure for this fragment of XPath in the case of general, non-cyclic DTDs.

1. INTRODUCTION

XPath forms a crucial component in many web applications that involve the processing of XML messages. One fundamental type of static analysis for XPath queries is to determine the containment relationship between pairs of queries. A query P *contains* a query Q if the answers to P always include all the answers to Q. Containment tests are useful in query optimisation, query processing using views, and query cache utilisation, for example. As a result, the containment problem for XPath has been the subject of considerable study, both in general [1, 5, 7] and in the case when the XML data being queried is assumed to conform to a Document Type Definition (DTD) or other forms of constraint [2, 3, 6, 8, 10].

As has become common practice, we indicate the fragment of XPath under study by listing the operators permitted. For example, $\mathrm{XP}(/, [\,], *, //)$ denotes the fragment in which the child, predicate, wildcard and descendant operators are permitted. It is well-known that containment can be solved efficiently for each of the fragments $\mathrm{XP}(/, [], *)$, $\mathrm{XP}(/, [], //)$ and $\mathrm{XP}(/, //, *)$, but is coNP-complete for $\mathrm{XP}(/, [], *, //)$ [5].

In this paper, we are specifically interested in XPath queries in the presence of DTDs; that is, we assume that, along with each query (or queries) we are given a DTD, with the assumption that the queries are to be evaluated on documents conforming to the DTD. In this case, we denote the XPath fragment by including "DTD" in the notation, so $\mathrm{XP}(\mathrm{DTD}, //)$, for example, denotes the fragment where we are given XPath expressions using only the descendant operator along with a DTD.

Testing containment in the presence of DTDs is considerably harder than in the case without DTDs; even containment for $\mathrm{XP}(\mathrm{DTD}, /, [\,])$ is coNP-complete [9]. A classification of tractable, intractable and even undecidable fragments is given in [6]. In this paper, we focus on $\mathrm{XP}(\mathrm{DTD}, /, //)$ because containment for this fragment can be decided is in PTIME and the fragment is sufficient to demonstrate the problems with previous approaches.

Neven and Schwentick [6] have already shown that containment of queries in $\mathrm{XP}(\mathrm{DTD}, /, //)$ can be decided in PTIME by using tree automata techniques. On the other hand, for many classes of queries, including relational queries, testing whether P contains Q is done by applying the well-known and intuitively appealing *chase* procedure to Q, followed by deciding whether there is a *containment mapping* from P to the chase of Q. So it would be of practical benefit, as well as intellectual satisfaction, to extend the use of the chase procedure and containment mappings to the case of deciding containment for $\mathrm{XP}(\mathrm{DTD}, /, //)$ in PTIME, as well. However, in order to achieve this, there are two problems to overcome.

The first problem is that there may not be a containment mapping between two equivalent queries in $\mathrm{XP}(\mathrm{DTD}, /, //)$. This is similar to the situation pointed out by Miklau and Suciu [5] in the case of $\mathrm{XP}(/, //, *)$. They show that, given queries $P = a/{*}//b$ and $Q = a//{*}/b$, there is no containment mapping from P to Q or from Q to P, despite the fact that P and Q are equivalent. A similar problem arises for $\mathrm{XP}(\mathrm{DTD}, /, //)$ with queries $P = a/a//a$ and $Q = a//a/a$ and DTD D with the single rule $a \to a^*$. Note that this rule is recursive and that the DTD is therefore *cyclic*.

The second problem is to find a suitable set of constraints with which to chase a query, while ensuring that the resulting chase procedure terminates in polynomial time. Recently, Wang and Yu [8] studied containment for the fragment $\mathrm{XP}(\mathrm{DTD}, /, [], //)$, proposing new types of constraints inferred from DTDs and two different chase procedures. How-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. This article was presented at:
DBPL '11.
Copyright 2011.

ever, their first chase procedure does not terminate on the example above, while for a generalisation of the above query to include n descendant edges, their second chase procedure results in a set of chased queries whose size is exponential in n.

In this paper (Section 4), we provide a way of ensuring that the chase of a query Q in XP(DTD, /, //) with respect to DTD D produces a single, unique result whose size is polynomial in the size of Q and D, even in the presence of cyclic DTDs. This is done by introducing a new operator || into XPath, which loosely speaking denotes either / or //, and replacing certain groups of / and // operators in a query by || operators. Thus the result of chasing a query Q in XP(DTD, /, //) with constraints derived from DTD D is a query in XP(DTD, /, //, ||), which we call a *generalised* query. For example, the generalised query resulting from chasing the query $a//a/a$ above with constraints derived from the DTD rule $a \rightarrow a^*$ is the generalised query $a||a||a$. Given a query P in XP(DTD, /, //) and a query Q in XP(DTD, /, //, ||), we then define a generalised form of containment mapping from P to Q, and present an algorithm that decides whether there is a generalised containment mapping from P to Q in time polynomial in the sizes of P and Q.

The above results consider only what we term *simplified* DTDs, where the DTD can be represented by a simple directed graph. Simplified DTDs have also been considered in [4, 8], for example. In this paper we also consider the containment problem for queries in XP(DTD, /, //) under *general* but non-recursive DTDs. Many constraints for use with the chase have been introduced in previous papers [4, 8, 10] but only for restricted classes of DTDs. We show that these constraints are insufficient in the case of containment for queries in XP(DTD, /, //) with respect to general, non-recursive DTDs. To address this, we introduce a new constraint, called a *family constraint*, which generalises previous constraints, and show that using family constraints along with other previous constraints enables a complete chase procedure for checking containment (Section 3).

2. BACKGROUND

We use the terms XPath query and tree pattern interchangeably. An XPath query in XP(/, [ ], //) can be represented by a tree pattern that uses *child* edges and *descendant* edges. For example, the tree pattern on the right of Figure 1 corresponds to the XPath query $a[b//g]//e/g$. We will consider only Boolean queries in this paper, and restrict ourselves to queries in XP(/, //); hence, each initial tree pattern corresponds to a path, although the chase can transform such a query into one in XP(/, [ ], //).

Given a pattern P, we denote the set of nodes of P by $N(P)$, the set of edges of P by $E(P)$ (where each edge is a pair $(x, y) \in N(P) \times N(P)$), the root of P by $root(P)$, and the label of a node $x \in N(P)$ by $\lambda(x)$. The set of child edges is denoted by $E_/(P)$, while the set of descendant edges is denoted by $E_{//}(P)$. When P is in XP(/, //) we denote the single leaf node in P by $leaf(P)$.

Tree patterns are queries to be evaluated on trees. Because we will ultimately consider only trees that conform to a given DTD, it is sufficient for now to consider trees whose node labels are drawn from a finite alphabet Σ. We denote the set of all trees whose node labels are drawn from Σ by T_Σ. We use the same notation for trees as that defined for tree patterns above (except of course there are no descendant edges in a tree).

Given a tree pattern P and a tree $t \in T_\Sigma$, a *homomorphism* from P to t is a function $h : N(P) \rightarrow N(t)$ that is

- *root-preserving*: $h(root(P)) = root(t)$,
- *label-preserving*: for each $x \in N(P)$, $\lambda(x) = \lambda(h(x))$,
- *child-edge-preserving*: for each $(x, y) \in E_/(P)$, we have $(h(x), h(y)) \in E(t)$, and
- *descendant-edge-preserving*: for each $(x, y) \in E_{//}(P)$, we have $(h(x), h(y)) \in E^+(t)$, where $E^+(t)$ denotes the transitive closure of the edge relation $E(t)$.

Then the evaluation of a Boolean query P on tree $t \in T_\Sigma$, denoted $P(t)$, is *true* if and only if there is a homomorphism from $N(P)$ to $N(t)$. We also say that t *satisfies* Q. We denote by $SAT(Q)$ the set of trees in T_Σ that satisfy Q.

We say that query P *contains* query Q, denoted $P \supseteq Q$, if and only if, for each tree $t \in T_\Sigma$, $Q(t)$ implies $P(t)$. Given a pair of patterns P and Q a *containment mapping* from P to Q is a function $h : N(P) \rightarrow N(Q)$ satisfying the same conditions as for a homomorphism defined above (with Q substituted for t), except that in the third condition we require that $(h(x), h(y)) \in E_/(Q)$ (child edges must map to child edges). For queries P and Q in XP(/, [ ], //), $P \supseteq Q$ if and only if there is a containment mapping from P to Q [1].

Now assume that we are given a DTD D, and we wish to consider containment of queries under the assumption that the trees being queried *satisfy* D. We denote by $SAT(D)$ the set of trees satisfying DTD D. We say that query P *D-contains* query Q, denoted $P \supseteq_D Q$, if and only if, for each tree $t \in SAT(D)$, $Q(t)$ implies $P(t)$.

Often when studying D-containment of queries, a simplified form of DTD, one that can be modelled by a simple directed graph G, is considered [4, 8]. Each element in D is modelled as a node in G, with an edge from node a to node b in G if an a-element can have a b-element as a child. The DTD is recursive if the graph is cyclic. Edges in G can be labelled with one of 1, ?, + or *, with the usual DTD semantics, although this does capture general disjunction or grouping of elements as provided by DTDs. We will consider only the case where each edge is effectively labelled with *, similar to [4], and will call DTDs that can be modelled by such a graph *simplified*.

One way to test D-containment is to derive a set C of constraints from D, chase Q with C, and find a containment mapping from P to the chased query. Although this method works for some classes of query and DTD [4, 8, 10], it sometimes fails to terminate for queries in XP(/, //) and simplified DTDs, as pointed out in the Introduction.

For non-recursive DTDs, the following five types of constraint, which can be inferred from a DTD, have been considered. The first is from [10], while the last three are from [4].

1. A *sibling constraint (SC)* is of the form $a : b \downarrow c$ and states that, whenever an a-node has a b-node as a child, then it also has a c-node as a child. A special form of SC is $a : \emptyset \downarrow c$, which states that every a-node has a c-node as a child.

2. A *cousin constraint (CC)* is of the form $a : b \Downarrow c$ and states that, whenever an a-node has a b-node as a descendant, then it also has a c-node as a descendant. A

special form of CC is $a : \emptyset \Downarrow c$, which states that every a-node has a c-node as a descendant.

3. A *parent-child constraint (PC)* is of the form $a \Downarrow^1 b$ and states that, whenever a b-node is a descendant of an a-node, then it is necessarily a child.

4. An *intermediate node constraint (IC)* is of the form $a \overset{c}{\to} b$ and states that, whenever there is a path from an a-node to a b-node, there is a c-node on the path.

While the above constraints allow for complete containment tests when the DTD is simplified and non-recursive [4], this is no longer the case for general non-recursive DTDs. For completeness in this case (as shown in Section 3), we need a new constraint that is a generalisation of sibling and cousin constraints.

5. A *family constraint (FC)* is of the form $a[\$_1 b] \Downarrow [\$_2 c]$, where each of $\$_1$ and $\$_2$ is either / or //. It states that, whenever an a-node has a b-node as a child (if $\$_1$ is /) or descendant (if $\$_1$ is //), then it also has a c-node as a child (if $\$_2$ is /) or descendant (if $\$_2$ is //), respectively. As special cases, if every a-node must have a c-node as a child (or descendant), we write $a \Downarrow [/c]$ (or $a \Downarrow [//c]$).

When both $\$_1$ and $\$_2$ are / (respectively //), then an FC corresponds to an SC (respectively CC).

Example 1. Consider the following DTD, which is not simplified because of the disjunction in the first rule:

$$
\begin{aligned}
a &\to b \mid (c,(d|e)) \\
b &\to c \\
d &\to f \\
e &\to f
\end{aligned}
$$

If an a-node has a c-child then it must have an f-descendant, and if an a-node has an f-descendant then it must have a c-child. So the FCs $a[/c] \Downarrow [//f]$ and $a[//f] \Downarrow [/c]$ hold. Note that an a-node can have a c-descendant without having an f-descendant, so a CC will not capture the first constraint. □

Wang and Yu [8] introduced three further types of constraint which are necessary to handle situations that arise with recursive DTDs (and are expressed with respect to the graph of a simplified DTD):

6. A *child-of-first-node constraint* (*CFN*) is of the form $a \xrightarrow{/b} b$ and states that on every path from an a-node to a b-node, the node immediately following the a-node must be a b-node.

7. A *parent-of-last-node constraint* (*PLN*) is of the form $a \xrightarrow{a/} b$ and states that on every path from an a-node to a b-node, the node immediately preceding the b-node must be an a-node.

8. An *essential edge constraint* (*EE*) is of the form $a \xrightarrow{/} b$ and states that every path from an a-node to a b-node must contain an edge from an a-node to a b-node.

To simplify the notation and presentation, we will say:

1. (a, b) is *initial* if and only if the CFN $a \xrightarrow{/b} b$ holds,
2. (a, b) is *final* if and only if the PLN $a \xrightarrow{a/} b$ holds, and
3. (a, b) is *essential* if and only if the EE $a \xrightarrow{/} b$ holds.

We collectively call these last three constraints *essential constraints*, abbreviated ECs.

Note that a pair being initial or final implies that it is essential, although a pair can be essential without being either initial or final. For example, if $E(G) = \{(a,a),(a,b),(b,b)\}$, then (a, b) is essential but neither initial nor final. Also note that if a pair (a, b) is both initial and final, this does not imply that the only path from a to b is (a, b). For example, if $E(G) = \{(a,b),(b,a)\}$, then (a, b) is both initial and final, but there are paths of the form $a(ba)^*b$ from a to b.

A *chasing sequence* of a query Q by a set of constraints C is a sequence $Q_0, \ldots, Q_k$ such that $Q_0 = Q$, for each $0 \le i \le k-1$, Q_{i+1} is the result of applying some constraint in C to Q_i, and no constraint can be applied to Q_k. Given a chasing sequence $Q_0, \ldots, Q_k$ of Q by C, we call Q_k the *chase* of Q by C (which turns out to be unique), denoted Q^C. Constraints such as PCs, ICs and FCs are applied to a query in the obvious way (we give examples in the next section). ECs, by contrast, are used to transform a query to a generalised query by replacing some occurrences of / and // by $||$, as explained in Section 4.

3. NON-RECURSIVE DTDS

Our goal in this section is to prove that, given a pair of queries P and Q in XP(/, //) along with a non-recursive DTD D, chasing Q with the set C of parent-child (PC), intermediate node (IC) and family constraints (FC) inferred from D to give Q^C, followed by checking for a containment mapping from P to Q^C gives a complete procedure for determining whether $P \supseteq_D Q$. To do so, we first show that there are trees satisfying Q and D to which there are one-to-one homomorphisms from Q^C. We then show that, given such a tree t, there is a homomorphism h from P to t that maps nodes of P to nodes in t that are in the image of one of these one-to-one homomorphisms g from Q^C. Then the composition of h with the inverse of g gives a containment mapping from P to Q^C.

It turns out that Q^C needs to be minimal for the existence of such a one-to-one homomorphism from Q^C to be guaranteed.

Example 2. Consider the following DTD D:

$$
\begin{aligned}
a &\to b|c \\
b &\to d \\
c &\to d
\end{aligned}
$$

We can infer the FCs $b \Downarrow [/d]$, $c \Downarrow [/d]$ and $a \Downarrow [//d]$. Given a query $Q = a/b$, the chase might first add a d-descendant to a followed by a d-child to b. Now there is no one-to-one homomorphism from Q^C to any tree in $SAT(D)$. The problem is that the d-descendant of a is Q^C is redundant. □

Given a query Q in XP(DTD, /, //) and a DTD D, the *models* of Q with respect to D, denoted $Mod(Q, D)$, are those trees in $SAT(Q) \cap SAT(D)$. Since the chase of Q with respect to a set of constraints C inferred from D preserves D-equivalence of queries, $Mod(Q, D) = Mod(Q^C, D)$.

Similar to Miklau and Suciu [5], we define the set of canonical trees in $Mod(Q, D)$ as those trees that "look like" Q^C.

A tree $t \in Mod(Q, D)$ is *minimal* if deleting any set of subtrees from t results in a tree no longer in $Mod(Q, D)$. A minimal tree $t \in Mod(Q, D)$ is *canonical* if t can be transformed to (a tree isomorphic to) Q^C by removing entire subtrees from t and replacing paths of degree-two nodes in t by descendant edges. (Of course, the resulting tree may no longer be in $Mod(Q, D)$.) The set of canonical trees for Q and D is denoted by $CMod(Q, D)$. We need to prove that canonical trees exist; it turns out that this is the case when Q^C is minimal. Having to work with minimal queries is not a problem, since queries in XP(/, [], //) can be minimised in polynomial time [1].

Lemma 1. *If Q^C is minimal, then $CMod(Q, C) \neq \emptyset$.*

An isomorphism f from the transformed tree t' for a tree $t \in CMod(Q, D)$ gives rise to a one-to-one mapping from the nodes of t to the nodes of Q^C. The inverse of f is a one-to-one mapping from Q^C to t. We will call such a mapping a *canonical homomorphism*. The following example shows that there is not necessarily a unique canonical homomorphism from a minimal Q^C to a canonical tree t.

Example 3. Consider the following DTD D

$$\begin{array}{lcl} a & \rightarrow & b, b, (c|d) \\ b & \rightarrow & (e|i) \\ c & \rightarrow & e \\ d & \rightarrow & e \\ e & \rightarrow & h \\ i & \rightarrow & h \end{array}$$

along with the query $Q = a/b$. The set of constraints C inferred from D includes (among others) $a \Downarrow [//e]$, $b \Downarrow [//h]$ and $e \Downarrow [/h]$. The minimal chase Q^C of Q with respect to C is shown on the right of Figure 1. Note that while Q is in XP(/, //), Q^C is in XP(/, [ ], //). A canonical model $t \in CMod(Q, D)$ is shown on the left of Figure 1. Also shown are two canonical homomorphisms f (dotted) and g (dashed) from $N(Q^C)$ to $N(t)$. $\square$

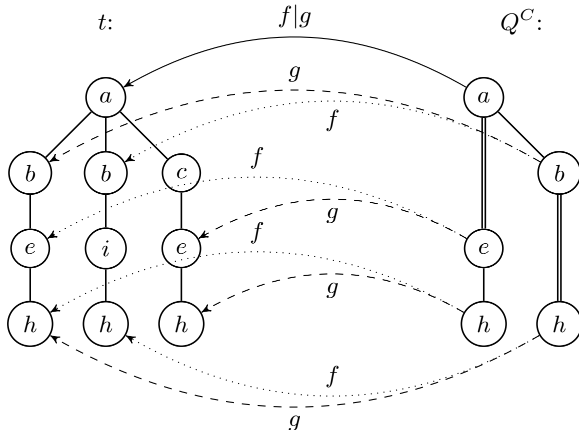

Figure 1: Example tree t and chase Q^C of query Q.

We now define the notion of a node being *essential* in a canonical tree in $CMod(Q, D)$. Let v be a node in a tree t. An *ancestral forest* of v in t is defined inductively as follows:

- each forest formed by the subtree rooted at v along with the subtrees rooted at zero or more siblings of v forms an ancestral forest of v,
- each forest formed by the subtree rooted at an ancestor u of v along with the subtrees rooted at zero or more siblings of u forms an ancestral forest of v

Given a canonical tree $t \in CMod(Q, D)$, a node $v \in N(t)$ is *inessential* in t if t can be transformed to $t' \in CMod(Q, D)$ by replacing an ancestral forest of v by a forest in which no node is labelled $\lambda(v)$. A node $v \in N(t)$ is *essential* in t if it is not inessential.

Example 4. Consider the canonical tree t in Figure 1. All the nodes labelled a, b and h are essential, as is the righthand node labelled e. The lefthand node labelled e and the nodes labelled c and i are inessential since the subtrees rooted at them can be replaced by subtrees not including nodes labelled e, c or i, respectively. $\square$

In all the results that follow we assume that P and Q are queries in XP(DTD, /, //), D is a non-recursive DTD, C is the set of PCs, ICs and FCs inferred from D, Q^C is the minimal chase of Q with respect to C, and t is a canonical tree in $CMod(Q, D)$.

Lemma 2. *Each essential node in t is in the image of some homomorphism from $N(Q^C)$ to $N(t)$.*

Lemma 3. *If $P \supseteq_D Q$, then there must be a tree $t \in CMod(Q, D)$ and a homomorphism h from $N(P)$ to $N(t)$ such that, for each $v \in N(P)$, $h(v)$ is essential.*

Theorem 1. *If $P \supseteq_D Q$, then there is a containment mapping from P to Q^C.*

4. RECURSIVE DTDS

We now turn to considering how to use the chase to test query containment for queries in XP(DTD, /, //) with respect to a potentially recursive DTD. As pointed out in Section 2, we will restrict ourselves to simplified DTDs, ones that can be represented by simple, directed graphs. As a result, we will use the terms simplified DTD and graph interchangeably.

We start our investigation by first considering the simpler setting in which the simplified DTD G is also non-recursive, that is, the directed graph G is acyclic. Note that FCs cannot arise in a simplified DTD because all children are optional and no grouping is allowed. Hence, perhaps it is not surprising that it turns out that, given queries P and Q in XP(DTD, /, //), we can test whether $P \supseteq_G Q$ by first chasing Q with (only) the set of ICs and PCs implied by G to give Q^C, and then checking whether there is a containment mapping from $N(P)$ to $N(Q^C)$.

Lemma 4. *Let G be a simplified, non-recursive DTD and P and Q be in XP(DTD, /, //). Let C be the set of ICs and PCs inferred from G, and Q^C be the result of chasing Q with C. If $P \supseteq_G Q$, then there is a containment mapping from P to Q^C.*

When trying to extend the above result to queries in XP(DTD, /, //) with recursive DTDs, the situation becomes much more complicated. The problem is that, when the

(DTD) graph G has cycles, there can be pairs of equivalent queries without containment mappings between them. The simplest example of this situation is that given in the Introduction, namely, for queries $P = a/a//a$ and $Q = a//a/a$ and DTD G with rule $a \rightarrow a^*$. Every path through G comprises only nodes labelled a, so $P \equiv_G Q$, but there is no containment mapping from P to Q or from Q to P.

To address the above problem, we propose introducing a new notation which allows one query form to denote a number of equivalent queries in XP(DTD, /, //). First we introduce a new XPath operator || which, loosely speaking, denotes either / or //. Then we allow for consecutive occurrences of || in a query to be assigned to groups, indicated by superscripts, representing group numbers, on the || operators. A group of || operators denotes the set of subqueries obtained by replacing each || within the group by either / or //, as long as at least one // remains in the group. As an example, $a \,||^1\, a \,||^1\, a$ denotes the pair of queries $a/a//a$ and $a//a/a$. We call a query using the || operator a *generalised* query.

To decide whether $P \supseteq_G Q$, for queries P and Q in XP(DTD, /, //) and potentially recursive DTD G, we will proceed as follows. We will first chase Q with the set C of PCs and ICs derived from G to give Q^C. We will then form a generalised query $Q^C_{||}$ from Q^C and find a containment mapping from P to $Q^C_{||}$. In the following two subsections, we define the notion of a generalised query, and show how to transform a given query Q into a generalised query that is G-equivalent to Q. In Section 4.3 we define what it means for a containment mapping to exist from a query to a generalised query, and provide a polynomial-time algorithm for deciding whether such a containment mapping exists.

4.1 Generalised Queries

A *generalised* query $Q_{||}$ is an XPath query in which, in addition to the operators / and //, the operator || can be used. Occurrences of || in $Q_{||}$ must be assigned to *groups* which are numbered using positive integers. More specifically, each occurrence of || in $Q_{||}$ must be assigned to either one group or two consecutively numbered groups: if operator || is in group i, this is indicated by $||^i$; if || is in groups i and $i+1$, this is written ${}^i||^{i+1}$. Finally, only consecutive occurrences of || can belong to the same group.

A generalised query represents a set of queries defined as follows. An *instantiation* of a generalised query $Q_{||}$ is a query in XP(DTD, /, //) obtained by replacing each occurrence of || in $Q_{||}$ by either / or // while ensuring that in each group at least one || is replaced by a // operator. The *expansion* of a generalised query $Q_{||}$, denoted $Q^X_{||}$, is the set of instantiations of $Q_{||}$. The meaning of $Q^X_{||}$ is defined as one would expect. Let $Q^X_{||} = \{Q_1, \ldots, Q_m\}$. Given a tree t, $Q^X_{||}(t)$ is defined as $Q_1(t) \cup \cdots \cup Q_m(t)$, or, in the case of Boolean queries, as $Q_1(t) \vee \cdots \vee Q_m(t)$.

Example 5. $Q_{||} = a_1//a_2||^1a_3{}^1||^2a_4||^2a_5/a_6$ is a generalised query in which the || operators have been placed in two groups. The expansion $Q^X_{||}$ of $Q_{||}$ is a set of five queries, corresponding to five instantiations of groups 1 and 2. The instantiations of groups 1 and 2 are (1) $a_2/a_3//a_4/a_5$, (2) $a_2//a_3/a_4//a_5$, (3) $a_2//a_3//a_4/a_5$, (4) $a_2/a_3//a_4//a_5$ and (5) $a_2//a_3//a_4//a_5$. Note that $a_2//a_3/a_4/a_5$ is not an instantiation of groups 1 and 2 because both occurrences of || in group 2 have been replaced with /. Note further that the size of $Q^X_{||}$ can be exponential in the size of $Q_{||}$. □

4.2 Transforming a Query to a Generalised Query

Let $Q = a_0\$_1a_1\$_2 \cdots \$_na_n$ be a query in XP(DTD, /, //), where each $\$_i$ is either / or //. We call a sequence $a_i\$_{i+1} \cdots \$_ja_j$, $0 \le i \le n-1$ and $i < j \le n$, a *query fragment* of Q. In this section, we identify those fragments of a query that need to be generalised in order to ensure that we can always find a containment mapping to them. We call these fragments *essential* fragments, because they are identified using the essential constraints defined in Section 2.

Given a simplified DTD graph $G = (N(G), E(G))$, we need to work with the *strongly connected components* (SCCs) of G. We call an SCC *trivial* if it is a singleton set $\{v\}$ and $(v, v) \notin E(G)$; otherwise, it is called *non-trivial.* (Of course, if G is non-recursive, then every SCC in G is trivial and no part of the following transformation is performed.)

Recall the definition of essential, initial and final pairs of nodes from Section 2. An *essential fragment* in Q is a maximal sequence of essential pairs in Q such that

1. if the first pair is not preceded by the operator // in Q, then it must be initial,
2. if the last pair is not followed by the operator // in Q, then it must be final, and
3. for each pair (x, y) in the sequence, either x or y is in a non-trivial SCC of G.

Example 6. Consider the simplified DTD G shown in Figure 2, in which all nodes are in the same SCC. By definition, only pairs of nodes that correspond to edges can be essential. All such pairs, except (b, e) and (d, c), are essential. Pairs (b, e) and (d, c) are not essential because there are paths from b to e and from d to c that do not include the edges (b, e) and (d, c), respectively. In addition, (a, b), (c, d) and (e, a) are all both initial and final. (b, c) is neither initial nor final because there is a path from b to c that starts with (b, e) and ends with (d, c). For similar reasons, (d, e) is neither initial nor final.

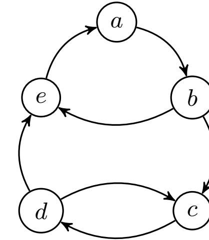

Figure 2: Graph of a simplified DTD.

Consider the query $Q_1 = a//b//c//d$. This fragment is essential because each pair is essential, and because (a, b) is initial and (c, d) is final.

Now consider the query $Q_2 = a//b//e//c//d$. Neither (b, e) nor (e, c) is essential, the latter because there is no edge $(e, c) \in E(G)$. So there are two essential fragments of length one in Q_2, namely, $a//b$ and $c//d$. □

Allocating operators to groups in an essential fragment really only makes sense when the fragment contains at least

two operators. So we first consider the special case of an essential fragment comprising a single pair (x, y). If the operator between x and y is /, there is nothing to be done. However, if the operator is //, we can replace it by / while preserving G-equivalence in the following situations:

- if x is preceded by // and y is followed by //,
- if x is preceded by //, y is followed by / (or is the final query node), and x is in a non-trivial SCC S while y is either in a trivial SCC or also in S,
- if x is preceded by / (or is the first query node), y is followed by //, and y is in a non-trivial SCC S while x is either in a trivial SCC or also in S.

EXAMPLE 7. Recall the query Q_2 from Example 6. Since $a//b$ is followed by // and $c//d$ is preceded by // (and a, b, c and d are all in the same SCC), the // operator in each can be replaced by / to give the G-equivalent query $a/b//e//c/d$. This query is G-equivalent to Q_2 because any path from a to e via b in G must start with (a, b), while any path from e to d via c must end with (c, d) □

After dealing with essential fragments of length one, we can assign the operators in longer fragments to groups using the following rules for rewriting. We start with an essential fragment $a_1\ \$_1\ a_2\ \$_2\ \cdots\ \$_{n-1}\ a_n$, $n > 2$, of Q. We now perform the following three steps:

1. (Overlapping groups) Rewrite the fragment assigning operators to groups as follows: $a_1\,\$_1^1\,a_2\,{}^1\$_2^2\cdots\$_{n-1}^{n-2}\,a_n$. Note that the first and last operators in the fragment are each in only one group.
2. (Extend groups) Apply the rule
$$a_{j-1}\ {}^{i-2}\$_{j-1}^{i-1}\ a_j\ {}^{i-1}\$_j^{i}\ a_{j+1}\ \Rightarrow\ a_{j-1}\ \$_{j-1}^{i-2}\ a_j\ {}^{i-2}\$_j^{i}\ a_{j+1}$$
for any (a_{j-1}, a_j), $2 \leq j \leq n$, that is both initial and final.
3. (Separate groups) Apply the rule
$$a_{j-1}\ {}^{i-2}\$_{j-1}^{i-1}\ a_j\ {}^{i-1}\$_j^{i}\ a_{j+1}\ \Rightarrow\ a_{j-1}\ \$_{j-1}^{i-2}\ a_j\ \$_j^{i}\ a_{j+1}$$
for any (a_j, a_{j+1}) that is initial and (a_{j-1}, a_j) that is final, $2 \leq j \leq n-1$.

Step (1) is applied once, then step (2) as many times as possible, and finally step (3) as many times as possible. After applying each rule, we re-number the groups to ensure that their numbers remain consecutive (so that further rule applications can occur). At the end, if any group i contains only / operators, we remove the group number i from each operator in the group. For each remaining group, we replace each $\$_i$ by $||$.

We now try to give the intuition behind the above steps. For simplicity, assume that the initial essential fragment Q uses only // operators. By assigning all the operators to overlapping groups in a generalised query $Q_{||}$, the most specific queries in the expansion of $Q_{||}$ are ones in which the operators / and // alternate. This is correct when all of the "internal" pairs of Q are only essential, but not initial nor final. If we have pairs that are both initial and final, then we can extend groups, as in step (2) above, which will allow for more occurrences of || within a group to be replaced by / in an expansion. Finally, we can separate two groups by removing an intervening group as in step (3) if the last pair in the first group is final and the first pair in the second group is initial.

EXAMPLE 8. Recall the query Q_1 from Example 6. Step (1) of the rewriting produces $a//^1b^1//^2c//^2d$. Since (b, c) is neither initial nor final, steps (2) and (3) cannot be applied. Finally, each // is replaced by || to give the generalised query $a||^1b^1||^2c||^2d$. So Q_1 is G-equivalent to each of $a/b//c/d$ and $a//b/c//d$, where each group has a single // operator. Note that Q_1 is *not* G-equivalent to more specific queries such as $a/b/c//d$ or $a//b/c/d$.

Now consider the query $Q_3 = a//b//c//d//e//a$. Once again, the whole fragment is essential, but this time step (2) of the rewriting can be applied because (c, d) is both initial and final. So the rewriting produces $a||^1b^1||^2c||^2d^2||^3e||^3a$, with three operators being in group 2. Hence, Q_3 is G-equivalent to (among others) $a/b//c/d/e//a$, $a//b/c/d//e/a$ and $a//b/c//d/e//a$. For example, in the first case, any path from b to a via c, d and e in G must pass through the sequence (c, d, e), so it is safe to separate c, d and e by / operators. Similarly, in the second case, any path from a to e via b, c and d in G must pass through the sequence (b, c, d), so it is safe to separate b, c and d by / operators.

Starting with any of the three queries above would give rise to the same rewriting since both / and // operators within a group are replaced by || operators. □

Given a query Q, we denote by $Q_{||}$ the result of applying the above rewriting to each essential fragment of Q.

LEMMA 5. *Let Q be in XP(/, //), G be a simplified DTD, and C be the set of PCs and ICs inferred from G. Then $Q \equiv_G Q_{||}^C$.*

4.3 Generalised Containment Mappings

We now need to define the notion of a containment mapping from a query P in XP(/, //) to a generalised query Q in XP(/, //, ||), by taking into account how child and descendant edges in P can map to || edges in Q. The basic idea is that no group in Q must have each pair in it mapped to by a child edge of P.

Let P be in XP(/, //) and Q be in XP(/, //, ||). A (generalised) containment mapping from P to Q is a function $h : N(P) \to N(Q)$ satisfying the root-preserving and label-preserving properties as before, as well as:

- *child-edge-preserving*: for each $(x, y) \in E_/(P)$, we have $(h(x), h(y)) \in E_/(Q) \cup E_{||}(Q)$,
- *descendant-edge-preserving*: for each $(x, y) \in E_{//}(P)$, we have $(h(x), h(y)) \in E^+(Q)$, and
- *group-preserving*: it is not the case that, in some group in Q, each pair (u, v) in the group has both $h^{-1}(u)$ and $h^{-1}(v)$ defined and has $(h^{-1}(u), h^{-1}(v)) \in E_/(P)$.

THEOREM 2. *Let P and Q be in XP(/, //), G be a simplified DTD, and C be the set of PCs and ICs inferred from G. Then $P \supseteq_G Q$ if and only if there is a generalised containment mapping from P to $Q_{||}^C$.*

We now present Algorithm 1 which, given queries P in XP(/, //) and Q in XP(/, //, ||), checks whether there is a

generalised containment mapping from P to Q, and hence, by Theorem 2, whether $P \supseteq_G Q$. The algorithm is modelled after that used in [5]. Two Boolean tables $C(u, x)$ and $D(u, x)$, for $u \in N(P)$ and $x \in N(Q)$, are constructed. $C(u, x)$ is *true* if there is a containment mapping from the subtree rooted at u to the subtree rooted at x; $D(u, x)$ is *true* if there is a containment mapping from the subtree rooted at u to the subtree rooted at x or one of its descendants.

Of course we also need to ensure that the group-preserving property of containment mappings is satisfied. The structure of a generalised query ensures that each $||$ operator is in either one or two groups. We consider a node x in Q to be in the group(s) of the operator that follows x in Q. Two consecutive nodes in a query can have at most one group in common. A third Boolean table $G(u, x, i)$ in Algorithm 1 denotes whether, by mapping u in P to x in Q, group i in Q has had a // operator mapped to it. We use $last(x, i)$ to denote whether node x in group i in Q is the last node in group i (when processing nodes bottom-up or from right-to-left).

Algorithm 1: Decide if $P \supseteq Q$

Input: queries P in XP(/, //), Q in XP(/, //, $||$)
Output: whether $P \supseteq Q$

1 **foreach** $u \in N(P)$ *and* x *in group* i *in* $N(Q)$ **do**
2     $G(u, x, i) \leftarrow false$
3 **foreach** $x \in N(Q)$ **do** // bottom-up
4   **foreach** $u \in N(P)$ **do** // bottom-up
5   **if** $leaf(u) \wedge \lambda(u) = \lambda(x)$ **then**
6   **foreach** *group* i *of* x **do** $G(u, x, i) \leftarrow true$
7             $C(u, x) \leftarrow D(u, x) \leftarrow true$; **break**
8   **if** $leaf(x)$ **then**
9             $C(u, x) \leftarrow D(u, x) \leftarrow false$; **break**
10   let y be the child of x // x is not a leaf
11   **if** $\lambda(u) \neq \lambda(x)$ **then**
12             $C(u, x) \leftarrow false$; $D(u, x) \leftarrow D(u, y)$; **break**
13   let v be the child of u // u is not a leaf
14   **if** $(u, v) \in E_/(P)$ **then**
15   **if** $(x, y) \in E_/(Q)$ **then** $C(u, x) \leftarrow C(v, y)$
16   **else if** $(x, y) \in E_{||}(Q)$ **then**
17   **if** x *and* y *are in the same group* i **then**
18                     $G(u, x, i) \leftarrow G(v, y, i)$
19   **if** $last(x, i) \wedge \neg G(u, x, i)$ **then**
20                     $C(u, x) \leftarrow false$
21   **else**
22                     $C(u, x) \leftarrow C(v, y)$
23   **else** $C(u, x) \leftarrow false$ // $(x, y) \in E_{//}(Q)$
24   **else**// $(u, v) \in E_{//}(P)$
25   **if** $(x, y) \in E_{||}(Q)$ **then**
26   **foreach** *group* i *of* x **do**
                    $G(u, x, i) \leftarrow D(v, y)$
27             $C(u, x) \leftarrow D(v, y)$
28         $D(u, x) \leftarrow C(u, x) \vee D(u, y)$
29 **output** $C(root(P), root(Q))$

EXAMPLE 9. Recall the query Q_1 from Examples 6 and 8, rewritten as the generalised query $a||^1b^1||^2c||^2d$. Let us refer to this query as Q, with P being $a/b//c/d$.

The crucial step in applying Algorithm 1 is when processing $b \in N(Q)$ and $b \in N(P)$. Since $(b, c) \in E_{//}(P)$, line 24 applies. Since $(b, c) \in E_{||}(Q)$ and b is in both groups 1 and 2, $G(b, b, 1)$ and $G(b, b, 2)$ are set to $D(c, c)$, which is *true*, in line 26. Then when $a \in N(Q)$ and $a \in N(P)$ are processed, because $(a, b) \in E_/(P)$, line 14 applies. Since $(a, b) \in E_{||}(Q)$ and both a and b are in group 1, $G(a, a, 1)$ is set equal to $G(b, b, 1)$ which is *true*. Then although $last(a, 1)$ is *true*, $\neg G(a, a, 1)$ is *false*, so line 22 sets $C(a, a)$ to $C(b, b)$ which is *true*.

Assume instead that P is simply a/b. Now when processing $b \in N(Q)$ and $b \in N(P)$, because $b \in N(P)$ is a leaf, line 6 applies, setting $G(b, b, 1)$ to *true*. Processing then proceeds as above for $a \in N(Q)$ and $a \in N(P)$.

Now assume that P is $a/b//d$. When processing $b \in N(Q)$ and $b \in N(P)$, line 24 applies because $(b, d) \in E_{//}(P)$. Because there is a descendant of $c \in N(Q)$ to which $d \in N(P)$ can map, $D(d, c)$ is *true*. Hence $G(b, b, 1)$ is set to *true* in line 26.

On the other hand, assume that P is $a/b/c$. When processing $b \in N(Q)$ and $b \in N(P)$, line 16 applies because $(b, c) \in E_/(P)$ and $(b, c) \in E_{||}(Q)$. Nodes b and c are both in group 2, so $G(b, b, 2)$ is set to $G(c, c, 2)$ which is *true* by virtue of c being a leaf in P and line 6. Although b is last in group 2, $G(b, b, 2)$ is *true*, so line 22 sets $C(b, b)$ to $C(c, c)$ which is *true*. When processing $a \in N(Q)$ and $a \in N(P)$, line 16 applies again because $(a, b) \in E_/(P)$ and $(a, b) \in E_{||}(Q)$. Nodes a and b are both in group 1, so $G(a, a, 1)$ is set to $G(b, b, 1)$ which is *false*. Now a is last in group 1 and $G(a, a, 1)$ is *false*, so $C(a, a)$ is set to *false* in line 20. $\square$

PROPOSITION 1. *For P in XP(/, //) and Q in XP(/, //, $||$), Algorithm 1 correctly determines whether there is a generalised containment mapping from P to Q. If P has n nodes and Q has m nodes, then Algorithm 1 runs in time $O(nm)$.*

5. CONCLUSION AND FUTURE WORK

We have studied the problem of using the chase procedure and containment mappings to test containment of XPath queries in XP(DTD, /, //). Previous approaches considered either simplified, non-recursive DTDs [4] or simplified recursive DTDs but without the guarantee of the chase terminating [8]. Although both papers applied their techniques to XP(DTD, /, [], //), the incompleteness of the first in the presence of *general* non-recursive DTDs and the non-termination of the second apply also to the smaller XPath fragment XP(DTD, /, //).

In this paper, we have introduced a new constraint, called a *family constraint*, which generalises the sibling and cousin constraints used previously and restores completeness to the chase procedure for testing containment of queries in XP(DTD, /, //) in the presence of *general* non-recursive DTDS. In the case of simplified *recursive* DTDs, we have introduced a generalised form of XPath syntax, which allows for the representation of an exponential number of equivalent queries, and a generalised form of containment mapping so that the chase can still be used as a polynomial-time test for containment.

We believe these results can be combined in order to provide a polynomial-time test for containment of queries in XP(DTD, /, //) using the chase in the presence of fully general DTDs.

6. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their constructive comments which helped to improve the paper.

7. REFERENCES

[1] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Tree pattern query minimization. *The VLDB Journal*, 11(4):315–331, 2002.

[2] A. Deutsch and V. Tannen. XPath queries and constraints, containment and reformulation. *Theoretical Computer Science*, 336(1):57–87, 2005.

[3] P. Genevès, N. Layaïda, and A. Schmitt. Efficient static analysis of XML paths and types. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 342–351, 2007.

[4] L. V. S. Lakshmanan, H. W. Wang, and Z. J. Zhao. Answering tree pattern queries using views. In *Proc. 32nd Int. Conf. on Very Large Data Bases*, pages 571–582, 2006.

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

[6] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *Logical Methods in Computer Science*, 2(3):1–30, 2006.

[7] B. ten Cate and C. Lutz. The complexity of query containment in expressive fragments of XPath 2.0. *J. ACM*, 56(6):31:1–31:48, September 2009.

[8] J. Wang and J. X. Yu. Chasing tree patterns under recursive DTDs. In *Proc. 15th Int. Conf. on Database Systems for Advanced Applications*, pages 250–261, 2010.

[9] P. T. Wood. Minimising simple XPath expressions. In *Proc. WebDB 2001: Int. Workshop on the Web and Databases*, pages 13–18, 2001.

[10] P. T. Wood. Containment for XPath fragments under DTD constraints. In *Proc. 9th Int. Conf. on Database Theory*, pages 300–314, 2003.