A Decision Procedure for XPath Satisfiability in the Presence of DTD Containing Choice

Yu Zhang[1,2], Yihua Cao[1], and Xuhuao Li[1]

[1] Department of Computer Science & Technology,
University of Science & Technology of China, Hefei, 230027, China
[2] Anhui Province Key Lab of Software in Computing and Communication, Hefei, China
yuzhang@ustc.edu.cn

Abstract. XPath satisfiability is one of the most basic problems of XML query optimization. A satisfiability decision framework, named *SAT-DTD*, is proposed to determine, given a set of XPath queries P and a DTD τ, which subset of P are satisfiable by an XML tree conforming to DTD τ. In the framework, an indexed NFA is constructed from the set of XPath queries P, and then the NFA is driven by simple API for DTD (SAD, something like SAX) events, derived from DTD τ, to evaluate the predicates in P and to decide the satisfiability of P. Especially, DTD choice (*i.e.* '|' operator) is taken into consideration, and an algorithm, named *SAT-DTD_C*, which bases on *SAT-DTD*, is put forward to determine the unsatisfiability caused by DTD choice. At last, the complexity of the algorithms is analyzed, and the correctness of the algorithms is tested by experiments.

Keywords: DTD choice; XPath satisfiability; automaton.

1 Introduction

XPath [1] is a query language used to navigate the node or node set in XML files. As a sub-language of XSLT, XQuery *et al.*, XPath has been widely used in XML query, transformation and update.

To improve the efficiency on XML access control and query transformation, the query containment problem received a great attention recently [2-6]. The general formulation of this problem is as follows: given two XPath queries p and q, check whether for any tree t, the results from evaluating p always contain those of q, and if so we call that p contains q, denoted as $q \subseteq p$. In the literature, much attention has been paid to analyzing the complexity [2,3], to simplifying the NP-complete containment problem or to converting it into the homomorphism problem among XPath trees in order to exploit EXPTime or PTime approximate algorithms [2,4-6].

XPath satisfiability refers to the state that given an XPath query p, if there exists an XML document d, so that the results from evaluating p are nonempty, then p is satisfiable, denoted as SAT(p). Furthermore, XPath satisfiability can be considered together with an XML specification definition (*i.e.* DTD or XML Schema) τ, that is, if there exists an XML document d such that d conforms to τ and the answer of p is

Y. Zhang et al. (Eds.): APWeb 2008, LNCS 4976, pp. 202 – 213, 2008.

nonempty, then we say (p, τ) is satisfiable, denoted as SAT(p, τ). As a new hot spot of research, XPath satisfiability is subsumed by the complement of the containment problem for XPath, and is considered to be far from tight than the latter [8].

Hidders first discussed XPath satisfiability in [10]. He introduced TDG (Tree Description Graph) to describe XPath expression and analyzed the complexity of deciding TDG's satisfiability, however, he did not consider any XML specification languages. Later Lakshmanan *et al.* began to study the XPath satisfiability in the presence of XML Schema [11], the Node Identity Constraint (NIC), which is common in XPath 2.0, was discussed and a method for deciding the satisfiability of TPQ (Tree Pattern Query) was presented, but they did not consider the choice element in XML Schema Definition (XSD). Benedikt and Fan *et al.* [8] analyzed a variety of factors contributing to the complexity of XPath satisfiability in the round, such as with or without DTD, with or without data values, with or without predicates etc. Other related researches include: Marx studied conditional axes in XPath 2.0 [12]; Geerts and Fan discussed sibling axes in the presence of XML Schema [13].

All the papers above mainly studied the complexity of XPath satisfiability under various factors, most of which did not provide with verifying algorithms on the satisfiability. Particularly, deciding the satisfiability of TPQ will make the complexity ascending from PTime to NP-complete, if DTD contains choice (*i.e.* '|' operator). It is of great significance to propose and optimize suitable algorithms for identifying and checking the unsatisfiability for a main class of XPaths under this common situation.

In this paper, we intend to design algorithms on deciding the satisfiability for a set of XPath queries P in the presence of DTD τ, in which a subset of XPath 1.0, denoted as $XP^{\{//,*,[\,]\}}$ (those include descendent axes, wildcards and predicates), and DTD with or without choice are considered. Our major contributions are as follows:

- A framework based on automaton techniques, named *SAT-DTD*, is proposed to decide which subset of P are satisfiable in the presence of τ without choice. In the framework, an indexed NFA constructed from P is driven by a sequence of SAD (simple API for DTD) events derived from τ, to decide the satisfiability for P.
- Based on the framework *SAT-DTD*, an algorithm, named *SAT-DTD_C*, is proposed to identify direct and indirect conflict of XML elements caused by DTD choice and to decide the unsatisfiability of XPath queries caused by the conflict.
- The complexities of the above algorithms are analyzed, and the experimental results demonstrate the correctness and the efficiency of our techniques.

The rest of this paper is organized as follows: Section 2 presents some basic concepts. Section 3 and 4 describe *SAT-DTD* and *SAT-DTD_C* respectively. Section 5 analyzes the complexities of our techniques, shows the experimental results and indicates potential optimization points. Section 6 concludes this paper.

2 Basic Concepts

2.1 DTD

A subset of DTD which only contains element declarations is considered here. Algorithms based on the subset could be easily extended on DTD that further contains attribute-list declarations.

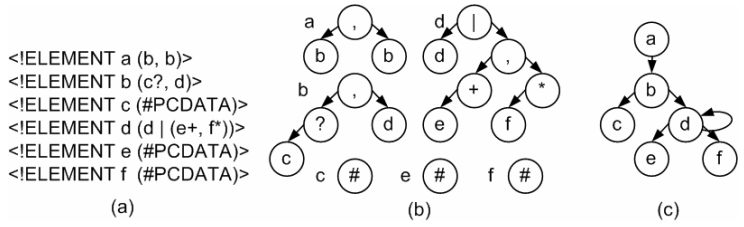

Fig. 1. (a) A DTD document; (b) the corresponding content model trees of (a) in Xerces; (c) the corresponding DTD graph of (a)

DTD document (e.g. Fig.1(a)) declares the content model for each XML element. The existence of operator '?', '*', '+', especially '|' in an element declaration makes the content model complex. There are many representations of content model in practice. Xerces [14] represents each element's content model as a binary tree, called *content model tree* (e.g. Fig.1(b)) in order to assist in validating XML document. [4] proposed a kind of DTD graph (e.g. Fig.1(c)) to simplify the DTD. We can see that DTD graph gives a monolithic view of DTD, while content model tree similar to Xerces gives a partial but detailed view. So we combine these two models to represent DTD.

Definition 1. Given a DTD document τ, the corresponding **DTD Graph** is a directed graph $G=(N, C)$, where each node $n \in N$ corresponds to one element in τ, each edge $< n_1, n_2 > \in C$ represents that the element corresponding to n_2 is the sub-element of the element corresponding to n_1 in τ.

Definition 2. Given a DTD document τ, for any element e declared in τ, the corresponding **Content Model Tree** (CM Tree) of its declaration $decl_e$ is a binary tree $T_e^{(CM)}=(N,C)$, where $N = N_b \cup N_f$ is the node set, and C is the edge set. Each branch node $n_b \in N_b$ corresponds to an operator in $decl_e$ (such as '|'), each leaf node $n_f \in N_f$ corresponds to a sub-element of e. For each edge $< n_1, n_2 > \in C$, where $n_1 \in N_b$, $n_2 \in N$, if $n_2 \in N_b$, then n_2 is the next level operator of n_1 in $decl_e$; if $n_2 \in N_f$, then n_2 is the operand of n_1 in $decl_e$.

2.2 XPath

Fig.2 gives the grammar of $XP^{\{//,*,[\,]\}}$. Generally this kind of XPath can be represented in tree pattern [2] or in automaton [7,9,15]. We choose the latter to describe a set of *to-be-decided* XPath queries P, and DTD τ will be converted into a sequence of events to drive the automaton. Now we briefly introduce some related concepts.

[1] P ::= / E | // E
[2] E ::= E/E | E//E | E**[**Q**]** | label | **text( )** | * | **@*** | . | **@**label
[3] Q ::= E | E Op Const | Q **and** Q | Q **or** Q |
not(Q) | func(Q*)
[4] Op ::= **<** | **≤**| **>** | **≥** | **=** | **≠** | * | **div** | **+** | **-**

Fig. 2. XPath fragment supported

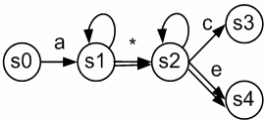

Fig. 3. Corresponding automaton of /a//*[c]//e

Definition 3. Given an XPath query p, the **main path expression** of p is the remaining expression of p in which all predicates are removed. The **nested path expression** is the XPath expression appearing in one of the top level predicates of p.

For example, if p is /a//*[c]//e, then /a//*//e is the main path expression of p, and c is the nested path expression of p.

Definition 4. Given a set of XPath queries P, the corresponding **Nondeterministic Finite Automaton** (NFA) A_P is incrementally constructed from the *main* and *nested path expressions* in P by path sharing technique in [9]. States in A_P can be further labeled with the following kinds: a **result state** is the state matching at least one *main path expression* in P, and the path from the initial state to one of the result states is called the **main path** in A_P; a **leaf state** is the state matching at least one *nested path expression* in P; a **branch state** is the state in a *main path* of A_P which branches to one or more leaf states; an **APS** (After-Predicate State) is the state in a *main path* of A_P whose parental state is a branch state; an **FPS** (First-location-step-of-Predicate State) is the state matching the first location step of some *nested path expression* in P.

Fig.3 shows the corresponding automaton of XPath /a//*[c]//e. Circles in the figure represent states, where s_4 is a result state and also an APS state, s_3 is a leaf state and also an FPS state, s_2 is a branch state. The child axes and descendant axes are represented by line with arrow and crewel with arrow, respectively.

Suppose that set *Preds* contains all top level predicates appearing in a set of XPath queries P, and set $P' \supseteq P$ contains all nested path expressions in *Preds*. We can construct the corresponding NFA and label the state kinds according to definition 4, the set of NFA states is denoted as *States*. In order to accelerate the running of NFA, the following indices are further added to some kinds of the NFA states.

- Add index table LR(s) to result state s, where LR(s) $\subseteq P$. $\forall p \in P$, if state s matches the main path expression of p, then $p \in$ LR(s);
- Add index table LP(s) to leaf state s, where LP(s) $\subseteq P' \times Preds$. $\forall p \in P$, for each predicate *pred* in p, if state s matches nested path expression q in *pred*, then <q, *pred*>$\in$ LP(s);
- Add index table LB(s) to branch state s, where LB(s) $\subseteq Preds$. $\forall p \in P$, for each location step *ls* of p and each predicate *pred* in *ls*, if s is the state matching *ls*, then *pred*$\in$ LB(s).

Particularly, in order to decide the XPath unsatisfiability caused by DTD choice (details are to be discussed in section 4), more indices are added to the NFA:

- Add index table LS(s) to result state or leaf state s, where LS(s) $\subseteq$ *States*. $\forall p \in$ LR(s) or <q, *pred*>$\in$ LP(s), if state t is the APS state of p or the FPS state of *pred*, then $t \in$ LS(s);
- Add parental state index *parent* and table LAPS(s) to APS state s, where *parent*$\in$ *States* and LAPS(s) $\subseteq P$. $\forall p \in P$, if there exists at least one predicate of p on the location step corresponding to *parent*, then $p \in$ LAPS(s);
- Add parental state index *parent* and table LFPS(s) to FPS state s, where *parent*$\in$ *States* and LFPS(s) $\subseteq P' \times Preds$. $\forall p \in P$, if *pred* is the top level predicate of p, and s is the state matching the first location step of a nested path expression q in *pred*, then <q, *pred*>$\in$ LFPS(s).

3 Deciding XPath Satisfiability in the Presence of DTD

Similar to the approach of converting XML into SAX events [16], DTD is firstly converted into SAD (simple API for DTD) events, and then be input to an NFA constructed from the *to-be-decided* XPath set to decide the XPath satisfiability.

3.1 SAD Events

Currently four kinds of SAD events have been defined: startDTDDocument( ), startElementDecl(*a*), endElementDecl(*a*), endDTDDocument( ), where *a* is the element name. First the DTD document is read and parsed into a DTD graph, and then the graph is converted into a tree for the convenience of issuing SAD events.

Definition 5. Given a DTD document τ and the corresponding DTD graph *G*, we can create the corresponding ***DTD tree*** *T* from *G*:

1) if *G* is a tree, then *T* = *G*;
2) if *G* is a DAG, then *T* is the expanded tree of *G* constructed as below:
   i) construct the root *r'* of *T* from the root *r* of *G*, and add *r'* to a queue *Q*.
   ii) if *Q* is empty, then the construction completes. Else, take a node *v'* out of the *Q* and get its corresponding node *v* in *G*, then construct the corresponding edges and child nodes of node *v'* in *T* according to the outgoing edges of node *v*, add all the constructed child nodes of node *v'* to *Q*.
   iii) repeat ii).
3) if *G* contains a circle, then we can deduce from the set of actual XML documents that for each ring $ring_i$ ($0 \le i \le k$, *k* is the number of rings in *G*), its possible maximum expanded depth $depth(ring_i)$ satisfies that the actual maximum expanded depth of $ring_i$ in *G* is not larger than $depth(ring_i)$.

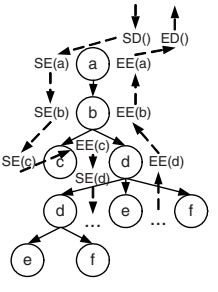

Fig. 4. Generating SAD events according to DTD tree

After converting a DTD graph to a DTD tree, we can depth-first traverse the DTD tree to issue SAD events. For instance, Fig.4 shows the corresponding DTD tree expanded from the DTD graph in Fig.1(c) and its SAD events propagated. In Fig.1(c) the self loop of element *d* is expanded by 2 levels. SD, SE, EE, ED in Fig.4 is short for startDTDDocument, startElementDecl, endElementDecl, endDTDDocument respectively, the parameter *a* followed by SE and EE is the element name. This figure omits the content model information.

3.2 SAD Events Handling: Deciding Satisfiability

The execution process of NFA is defined by the handling rules on SAD events. This process completes the decision of XPath satisfiability. We first discuss the decision procedure without considering DTD choice.

3.2.1 State Transition

Function *trans*(*stack*, *a*) does transition from the top *state set* of *stack* on the input *a* by the following rule: all states reached from any state in the top *state set* along edge

labeled '/*a*', '/*', '//*a*' or '//*' are matched states. Meanwhile, to handle multiple match, *trans* records the match layer *matchlayer* for each runtime state *s*, see section 3.2.4.

Algorithm.1 SAT-DTD

s_0: the initial state; *stack*: the NFA runtime stack, whose frame is a set of NFA states.

startDTDDocument() //briefly, SD
(1) *push*({s_0}, *stack*);

endElementDecl(*a*) //briefly, EE
(1) *sset* = *pop*(*stack*);
(2) for each *s* in *sset*{
(3) for each *pred* in LB(*s*)
(4) *resetPredRes*(*pred*);
}

endDTDDocument() //briefly, ED
(1) stop or wait another DTD document;

startElementDecl(*a*) //briefly, SE
(1) *sset* = *trans*(*stack*, *a*);
(2) *push*(*sset*, *stack*);
(3) for each *s* in *sset*{
(4) for each <*q*, *pred*> in LP(*s*)
(5) *evaluatePred*(*pred*, *q*);
(6) for each *p* in LR(*s*)
(7) *setRes*(*p*);
}

3.2.2 Deciding the Satisfiability of Main Path Expression

Given an XPath *p*, consider its main path expression p_m first. If after *trans*(*stack*, *a*), a result state *s* is reached, and *p* is in LR(*s*), then p_m is satisfiable according to the former definitions. So we use *setRes*(*p*) to record that p_m is matchable. (see line (6)~(7) in SE event handler of algorithm 1)

Predicates may contain XPath expressions, *i.e*. nested path expressions. Suppose *p*'s predicate *pred* contains an XPath *q*, namely, *pred* looks like […*q*…]. Denote *q*'s main path expression as q_m. If after a transition, result state *s* is reached and <q_m, *pred*> is in LP(*s*), then q_m is satisfiable. Next, we need to decide the satisfiability of predicates. (see line (4)~(5) in SE event handler of algorithm 1)

3.2.3 Predicates Evaluation

The satisfiability decision on predicates is mainly done through *evaluatePred*(*pred*,*q*).

Definition 6. Given a DTD τ and an XPath *p*, *pred* is a predicate of *p*, *e* is the general expression in *pred*. Suppose that *D* is the set of XML documents conforming to τ. *e* is satisfiable if there exists $d \in D$ so that the evaluation of *e* in *d* is true.

evaluatePred(*pred*, *q*) uses a bottom-up approach to evaluate the validity of *pred*. In line (4) of algorithm 2, *e*'s satisfiability is decided by evaluating *func*(…), e.g. *not*(*Q*), its satisfiability is opposite to the parameter *Q*.

In EE event handler, if current NFA backtracking state *s* is a branch state, then get each predicate *pred* from LB(*s*), and reset the evaluating status of *pred* (see line (2)~(4) in EE handler of algorithm 1) to prepare for next predicate evaluation.

Algorithm.2 evaluatePred

evaluatePred(*pred*, *e*)
(1) switch (*e.type*) {// see Fig.2
(2) case *E*: *e.valid* = true;
(3) case *E* Op Const: *e.valid* = *E.valid*;
(4) case *func*(Q_1, ..., Q_n): //include not(*Q*)
e.valid=*func*(Q_1.*valid*,..., Q_n.*valid*);
(5) case Q_1 or/and Q_2:
e.valid = Q_1.*valid* or/and Q_2.*valid*;
}
(6) if (*e.parent* != null)
(7) return *evaluatePred*(*pred*, *e.parent*);
(8) else *setPredRes*(*p*, *e.valid*);
(9) return *e.valid*;

3.2.4 Multiple Match

Multiple match is difficult for XSIEQ [15], an XPath query engine on XML stream. It also occurs in deciding XPath satisfiability.

Definition 7. Given a DTD τ and an XPath p, when deciding p's satisfiability using τ, one location step ls of p may match more than one DTD elements in different depths. This occurrence is called **multiple match**, ls is called **multiple match occurrence point**.

The necessary condition of multiple match occurrences is that the location steps contain descendent axes. A serious problem caused by multiple match is that either the evaluation among predicates on multiple match occurrence point or the evaluation between the predicate and the main path expression are out of sync. Let's take the latter case for example. Suppose that multiple match occurrence point ls matches two DTD elements e_1 and e_2, and there is a predicate $pred$ on ls, the XPath expression on the right side of ls is denoted as p_r. If the evaluation result is that $pred$ is satisfied on input e_1, but p_r is satisfied on input e_2, then the evaluation between $pred$ and p_r are out of sync, and may falsely evaluate the satisfiability of the whole XPath expression p.

Theorem 1. Given a DTD τ and an XPath p, assume that location step ls_i of p contains n predicates and can match m elements, *i.e.* $e_1,\ldots,e_m$, in the same path in τ. p can be represented as: $\ldots//ls_i[pred_{i1}][\ldots][pred_{in}]/ls_{i+1}/\ldots$. Then the XPath fragment $//ls_i[pred_{i1}][\ldots][pred_{in}]/ls_{i+1}$ can be satisfied if the following condition is met: when ls_i matches e_j $(1 \le j \le m)$, $pred_{i1}$, …, $pred_{in}$ and XPath fragment $/ls_{i+1}$ can also be satisfied.

Proof (briefly)**.** It can be inferred from the definition of XPath satisfiability.

To handle multiple match occurrences, according to theorem 1, for any multiple match occurrence point ls of p, there should be at least one matched element to make all predicates on ls and the right side XPath fragment of ls satisfied simultaneously. For example, if we use DTD in Fig.1(a) to decide the satisfiability for XPath set:{/a//*[c]//e,/a//*[c]/e}, then the satisfied XPath is /a//*[c]//e, and the reason why /a//*[c]/e is not satisfied is that on multiple match occurrence point //*, predicate [c] and main path expression fragment /e cannot be satisfied simultaneously.

4 XPath Unsatisfiability Caused by DTD Choice

DTD element declaration may contain operator '|', which states that its operands cannot occur in the same XML document simultaneously. To simplify the statement, in this section we use the term *node* (in DTD tree) instead of *element* (in DTD).

4.1 Conflict and XPath Unsatisfiability

Definition 8. If DTD formulates that certain two nodes cannot occur in the same XML document simultaneously, then there is a conflict between the two nodes, we call it **node conflict**. The conflict between sibling nodes is **direct conflict**; conflict between other nodes is **indirect conflict**.

Apparently, DTD can only declare conflicts between sibling nodes using operator '|', so all indirect conflicts are caused by direct conflicts. The way to find direct conflicts from indirect conflicts is as follows: given two nodes n_1 and n_2, if n_1 and n_2

conflict indirectly (they cannot be sibling nodes), the layers of n_1 and n_2 are $layer_1$ and $layer_2$ respectively, assume $layer_1 \leq layer_2$, their least recent common ancestor n_a ($layer_a < layer_1$), and n_a's two child nodes n_{ac1}, n_{ac2} are **AOS** (Ancestor-Or-Self) nodes of n_1, n_2 respectively, then the indirect conflict between n_1 and n_2 is caused by the direct conflict between n_{ac1} and n_{ac2}. The following theorem gives the necessary and sufficient condition of direct conflict between nodes.

Theorem 2. Assume $E=(e_1,\ldots,e_n)$ is the sub-element tuple of DTD element e, $T_e^{(CM)}$ is e's content model tree. The necessary and sufficient conditions of e_i and e_j conflict directly are: 1) their least recent common ancestor in $T_e^{(CM)}$ is operator '|', denoted as n_{a1}; and 2) the least recent ancestor of n_{a1} (except operator '|'), denoted as n_{a2}, is not '+' or '*'; and 3) if n_{a2} is '?', then there are no '+' or '*' in the path from n_{a2} to its least recent binary operator ancestor, such as ','.

Proof. If e_i and e_j conflict directly, then condition 1) must be true, consider n_{a1}'s least recent ancestor (except '|') (*i.e* condition 2)): if n_{a2} is ',', then there is definitely direct conflict between e_i and e_j; n_{a2} cannot be '+' or '*', this means that e_i and e_j can occur more than once without resulting conflicts; n_{a2} may be '?', if so, it needs more consideration (*i.e*. condition 3)). When n_{a2} is '?', assume its least recent ancestor n_{a3}, no matter ',' or '|', e_i and e_j conflict or not is unrelated to n_{a3}, we do not need to consider further about n_{a3}'s ancestors. The existence of '+' or '*' in the path between n_{a2} and n_{a3} also indicates that e_i and e_j can occur more than once, without resulting conflicts.

According to theorem 2, we can conclude a further condition on node conflict influencing XPath satisfiability.

Definition 9. Given two node sets A and B, if for each node n_a in A, n_a conflicts with each node n_b in B, then there is conflict between A and B, we call it **node set conflict**.

Theorem 3. Assume XPath p is represented as $\ldots//ls_i[pred_{i1}][\ldots][pred_{in}]/ls_{i+1}/\ldots$, and node set N_i contains nodes in the input DTD tree which match the location step ls_i, any conflict among the following node sets (*i.e. node set conflict*) will result in the unsatisfiability of p: 1) the node set N_{ik} whose elements are from the *topmost layer* and further selected from N_i by $pred_{ik}$ ($1 \leq k \leq n$), where nodes in *topmost layer* are those *closest* to root node in the DTD tree; 2) the node set N_{i+1} whose elements are from the *topmost layer* and further selected from N_i by the next location step $/ls_{i+1}$.

Proof. Just take the node sets N_{ik} selected by $pred_{ik}$ ($1 \leq k \leq n$) and N_{i+1} selected by the next location step $/ls_{i+1}$ for example: if the two node sets conflict each other, then according to definition 9, $pred_{ik}$ and $/ls_{i+1}$ cannot be satisfied simultaneously; if there is no conflict between the two node sets, then $pred_{ik}$ and $/ls_{i+1}$ can always select certain nodes to avoid conflicts, thus they are both satisfied. So, if there are node set conflicts in any two of the above node sets 1) and 2), p is unsatisfiable.

4.2 Deciding XPath Unsatisfiability Caused by Conflict

According to theorem 2, direct conflict can be statically decided. Assume $E=(e_1,\ldots, e_n)$ is the sub-element tuple of DTD element e, then for each n-bit bitset

BS_i=(b_{i1},…, b_{in}) of every node e_i: b_{ij}= 0 indicates that there is no conflict between e_j and e_i; else vice versa. Node conflicts can be detected by traversing content model trees.

Algorithm 3 gives the decision procedure for XPath satisfiability in the presence of DTD containing choice, which bases on the framework *SAT-DTD*. In the algorithm, the node conflicts caused by DTD choice are identified to decide the unsatisfiabilty of XPath expressions according to theorem 3.

Algorithm.3 SAT-DTD_C (*see Algorithm 1 for other event handlers***)**

s_0: the initial state; *stack*: the NFA runtime stack, whose frame is a set of NFA states.

```
startElementDecl(a) //briefly, SE
// see algorithm 1
(3) for each s in sset{
…
//(8~11) add matched element(ME for short)
 (8)   for each <q,pred> in LFPS(s)
 (9)    s.addME(q, a);
(10)   for each p in LAPS(s)
(11)    s.addME(p,a);
 // promote the matched elements.
(12)   for each st in LS(s){
(13)    for each <q,pred> in LFPS(st)
(14)     st.parent.addMEs(q, st.getME(q));
(15)    for each p in LAPS(st)
(16)     st.parent.addMEs(p, st.getME(p));
      }
     }
```

```
endElementDecl(a) //briefly, EE
// see algorithm 1
(2) for each s in sset{
…
// clear matched element in APS or FPS.
 (5)  if (s.isFirstMatch() and (s.isAPS() or s.isFPS()))
 (6)   s.clearAllMEs();
//check conflicts while reaching branch state
 (7)  if (s.isBranchState())
 (8)   for each p which has predicates
         pred1, ..., predn corresponding to s{
 (9)    for any p1, p2 in {pred1,..., predn, p}
(10)     checkConflict(s.getMEs(p1), s.getMEs(p2));
(11)    if (p.isLeftMostBranchState(s) and
          p.noConflicts( ))
(12)     p.satisfied = true;
        }
 // clear matched elements in branch state
(13)   s.clearAllMEs();
    }
```

The operations in algorithm 3 are as follows:

In SE event handler,

1) *add* operation: when reaching FPS or APS states, record currently matched nodes for each corresponding XPath expression beforehand. (line(8)~(11));

2) *promote* operation: when reaching result states or leaf states, the corresponding XPath expression is matched. Find all the corresponding APS and FPS states according to LS(*s*) index (line (12)), then promote the matched nodes recorded previously on those states to the parental branch state, in order to decide satisfiability (line (13)~(16)). A same XPath can be matched more than once, so it can be promoted multiple times.

In EE event handler,

1) *clear* operation: when backtracking to APS, FPS or branch states, clean matched nodes (line(5)~(6), (13)) to prepare for the next decision.

2) *check* operation: when backtracking to branch state *s* (line(7)), for each XPath *p* that contains predicate(s) on location step corresponding to *s*, decide the conflicts and record them (line(8)~(10)). Finally, if there is no conflict occurrence, *p* is satisfiable (line(11)~(12)).

5 Complexity Analysis, Optimization and Experimental

In this section we first analyze the complexity of *SAT-DTD* and *SAT-DTD_C*, then propose optimization approaches on *SAT-DTD_C*, and finally check the correctness through experiments.

5.1 Analysis on the Complexity of SAT-DTD Algorithms

According to the conclusion in [8], for the XPath with descendent axes and predicates, the complexity of its satisfiability decision is PTIME only if DTD does not contain choice; and the complexity is NP-complete for arbitrary DTD.

5.1.1 Complexity of SAT-DTD Algorithm without Considering DTD Choice

The time complexity of building an automaton from XPath queries is polynomial-time depending on the size of XPath queries. Therefore we focus on the runtime complexity of *SAT-DTD* (see algorithm 1).

Theorem 4. Assume the length of DTD event sequence is $2m$, there are x XPath queries, n states in automaton, q predicates, average x_q nested path expressions and op operators in each predicate respectively, average q_l predicates in each location step with predicates, average u multiple match occurrences in each multiple match occurrence point. Then the time complexity is $O(m\cdot(n+x+x_q\cdot op\cdot q+(x_q\cdot op\cdot q_l+x)\cdot u\cdot q/q_l))$.

Proof. Assume every time all states participate in state transition, then the maximum times of state transition is $O(m\cdot n)$ (for tree NFA, on average each state will transform once to another state). If every time the element start events reach all the result states, the complexity upper bound of main path expression's satisfiability decision is $O(m\cdot x)$, so the complexity upper bound of the satisfiability decision of nested path expressions is $O(m\cdot x_q\cdot q)$. And, each predicate expression needs op computations, the upper bound of predicate computation complexity is $O(m\cdot x_q\cdot op\cdot q)$.

On each multiple match occurrence point, every predicate and the main path expression need additional satisfiability decision at the same time, the time complexity is $O((m\cdot x_q\cdot op\cdot q_l+m\cdot x)\cdot u)$.

Finally the result is $O(m\cdot n+m\cdot x+m\cdot x_q\cdot op\cdot q+ (m\cdot x_q\cdot op\cdot q_l+m\cdot x)\cdot u\cdot q/q_l)$.

From theorem 4 we can see that multiple match has a great impact on the efficiency of *SAT-DTD* algorithm. But the probability of multiple match occurrence is low except that an XPath expression contains "//*".

5.1.2 Complexity of SAT-DTD_C Algorithm

Suppose a DTD contains w elements, the average length of an element's content model is l, then the complexity of deciding node conflict is $O(w\cdot l^2)$.

Theorem 5 gives the extra complexity of runtime *SAT-DTD_C* compared with *SAT-DTD* without considering DTD choice.

Theorem 5. Suppose the number of all predicates is q, each contains x_q XPaths on average; each location step that contains predicate(s) has q_l predicates on average; each FPS state or APS state has v matches on average. The *SAT-DTD_C* algorithm has an extra complexity of $O(v^{q_l\cdot x_q+l}\cdot q/q_l)$ than *SAT-DTD* without considering DTD choice.

Proof. The time complexity of detecting node set conflict in each location step with predicate(s) is $O(v^{q_l \cdot x_q + l})$, and there are q/q_l such location steps.

5.2 Optimization on SAT-DTD_C Algorithm

Theorem 5 points out that *SAT-DTD_C* is exponential as it has parameter q_l and x_q in the exponent. It can be optimized in the following aspects.

- Add functions to determine whether the content models contain operator '|'. For those exclude '|', the complexity of *SAT-DTD_C* will not exceed *SAT-DTD* without considering DTD choice.
- From the observation we see that the complexity of *SAT-DTD_C* is mainly caused by detecting node set conflicts, thus buffer with a lookup table can be used to reduce the complexity of *SAT-DTD_C*.
- Filter out those XPath expressions with high q_l and x_q, ignore them.

5.3 Correctness Checking Experiments

Table 1. The decision results of XPath satisfiability using *SAT-DTD*, where SAT represents the actual satisfiability, T represents *satisfiable*, and F represents *unsatisfiable*

XPath query	*SAT*	*SAT-DTD*	Remark
//category[description]/name	T	T	
//text[bold]/keyword	T	T	see condition 2) in theorem 2, there is no conflict between *bold* and *keyword*
//*[text="sth"]/parlist	F	F	there is conflict between *text* and *parlist*
//category[.//listitem]/text	F	F	there is conflict between the parent of *listitem*, *parlist*, and *text*
//category[.//listitem]//text	T	T	there is no conflict between the parent of *listitem*, *parlist*, and some *text*

Ordinary DTD documents contain only a small portion of operator '|', making the experiments focused on it more difficult to carry out. The experiments discussed in this sub-section focus on the correctness, that is, whether *SAT-DTD* can present a correct decision of XPath satisfiability. Table.1 shows some evaluation results in common circumstances. The DTD document used in this experiment is a fragment in XMark [17] described in Fig.5. The experiment result shows that *SAT-DTD* can correctly decide the XPath satisfiability in typical circumstances with wildcards and descendent axes.

```
<!ELEMENT site       (categories)>
<!ELEMENT categories (category+)>
<!ELEMENT category   (name, description)>
<!ELEMENT name       (#PCDATA)>
<!ELEMENT description (text | parlist)>
<!ELEMENT text       (#PCDATA | bold)*>
<!ELEMENT bold       (#PCDATA | bold)*>
<!ELEMENT parlist    (listitem)*>
<!ELEMENT listitem   (text | parlist)*>
```

Fig. 5. Fragment of xmark.dtd.

6 Conclusion

The proposed *SAT-DTD* algorithm can decide XPath satisfiability correctly. *SAT-DTD* without considering DTD choice has been applied to our XML access control

system to optimize access control rules and queries. Experiments have proved the correctness of the algorithm. Our next task is to apply optimized *SAT-DTD_C* to practical systems, in the meantime we will consider more factors that influence XPath satisfiability, such as: predicates of various properties, operators '?','*', '+' in DTD, etc.

Acknowledgments. This work has been supported by the National Natural Science Foundation of China under Grant No. 60673126.

References

[1] Clark, J., DeRose, S.: XPath Version 1.0. W3C Recommendation (1999), http://www.w3.org/TR/xpath
[2] Miklau, G., Suciu, D.: Containment and equivalence for a fragment of XPath. Journal of the ACM 51(1), 2–45 (2004)
[3] Wood, P.: Containment of XPath Fragments under DTD Constraints. In: Proceedings of Int. Conference on Database Theory (2003)
[4] Böttcher, S., Steinmetz, R.: A DTD Graph Based XPath Query Subsumption Test. In: Proceedings of XML Database Symposium at VLDB, Berlin, Germany (2003)
[5] Liao, Y., Feng, J., Zhang, Y., Zhou, L.: Hidden Conditioned Homomorphism for XPath Fragment Containment. In: Li Lee, M., Tan, K.-L., Wuwongse, V. (eds.) DASFAA 2006. LNCS, vol. 3882, pp. 454–467. Springer, Heidelberg (2006)
[6] Yoo, S., Son, J.H., Kim, M.H.: Maintaining Homomorphism Information of XPath Patterns. In: IASTED-DBA, pp. 192–197 (2005)
[7] Fu, M., Zhang, Y.: Homomorphism Resolving of XPath Trees Based on Automata. In: Dong, G., Lin, X., Wang, W., Yang, Y., Yu, J.X. (eds.) APWeb/WAIM 2007. LNCS, vol. 4505, pp. 821–828. Springer, Heidelberg (2007)
[8] Benedikt, M., Fan, W., Geerts, F.: XPath Satisfiability in the Presence of DTDs. In: Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 25–36 (2005)
[9] Diao, Y., Altinel, M., Franklin, M.J., et al.: Path Sharing and Predicate Evaluation for High-Performance XML Filtering. ACM TODS 28(4), 467–516 (2003)
[10] Hidders, J.: Satisfiability of XPath expressions. In: Proceedings of the 9th International Workshop on Database Programming Languages, pp. 21–36 (2003)
[11] Lakshmanan, L., Ramesh, G., Wang, H., et al.: On testing astisfiablity of the tree pattern queries. In: VLDB (2004)
[12] Marx, M.: XPath with conditional axis relations. In: EDBT (2004)
[13] Geerts, F., Fan, W.: Satisfiability of XPath Queries with Sibling Axis. In: Bierman, G., Koch, C. (eds.) DBPL 2005. LNCS, vol. 3774, pp. 122–137. Springer, Heidelberg (2005)
[14] Xerces2 Java Parser 2.6.0., http://xerces.apache.org/xerces2-j/
[15] Zhang, Y., Wu, N.: XSIEQ - An XML Stream Query System with immediate Evaluation. Mini-Micro Systems 27(8), 1514–1518 (2006)
[16] Sax Project Organization, SAX: Simple API for XML. (2001), http://www.saxproject.org
[17] Schmidt, A., Waas, F., et al.: A Benchmark for XML Data Management. In: Proceedings of the 28th VLDB Conference, Hongkong, China (2002)