Checking Satisfiability of Tree Pattern Queries for Active XML Documents

HAI-TAO MA[1], ZHONG-XIAO HAO[1,2], YAN ZHU[3]

[1] School of Computer Science and Technology, Harbin Institute of Technology, Heilongjiang, 150001, China
[2] College of Computer Science and Technology, Harbin University of Science and Technology, Heilongjiang, China
[3]College of Information Science and Engineering, Yanshan University, Hebei, China
[1]mahaitao@hit.edu.cn
[2]haozx@hrbust.edu.cn
[3]zhuxiaoyan@88mail.ysu.edu.cn

Abstract. Satisfiability is an important problem of queries for XML documents. This paper focuses on the satisfiability of tree pattern queries for Active XML(AXML for short) documents conforming to a given AXML schema. An AXML document is an XML document where some data is given explicitly and other parts are defined intensionally by means of embedded calls to Web services, which can be invoked to generate data. For the efficient evaluation of a query over an AXML document, one should check whether there exists an (A)XML document obtained from the original one by invoking some Web services, on which the query has a non-empty answer. An algorithm for checking satisfiability of tree pattern queries for AXML documents that runs polynomial time is proposed based on tree automata theory. Then experiments were made to verify the utility of satisfiability checking as a preprocessing step in queries procession. Our results show that the check takes a negligible fraction of the time needed for processing the query while often yielding substantial savings.

Keywords: Active XML, tree pattern queries, tree automata, satisfiability.

(Received October 30, 2007 / Accepted February 02, 2008)

1 Introduction

Active XML(AXML for short) Documents, as a powerful tool for distributed data management, were firstly proposed by Serge Abiteboul et al. in [2]. An AXML document is an XML document where some of the data is given explicitly and other parts are given intensionally, by means of embedded calls to Web services. When one of these calls is invoked, the result will be returned and inserted into the original document. Introducing Web services improves the dynamic and flexibility of XML documents largely. However, it brings many new problems such as documents rewriting [15],documents containment [3] and lazy queries for documents [1], and so on.

XPath is a subset of the standard query language for XML and is based on a basic paradigm of finding bindings of variables by matching tree patterns against a database. In this paper, we model queries as tree patterns described by Miklau and Suciu [14].

Detecting whether a given query is satisfiable, i.e., whether there exist any documents satisfying the query is an important problem of queries for XML documents. If one can decide, at compiling time, that a query is not satisfiable, then many unnecessary computations of queries processing can be avoided. Satisfiability of queries for XML documents has been widely investigated in recent years [5, 9, 11]. However, the corresponding problem of AXML documents, to our knowledge, has not

been well presented and traditional methods for XML documents become useless to solve it.

The focus of this paper is on the satisfiability problem of queries for AXML documents under a given AXML schema. The problem is more intricate than XML because we should consider all Web service calls embedded in it that may return some useful data contributing to the query results.

The only paper we found about AXML queries is [1]. The authors proposed a lazy manner of tree pattern queries evaluation and payed more attention to how to decide the relevant function calls. Here we focused on the satisfiability of tree pattern queries for AXML documents.

Satisfiability of queries for XML documents had been widely investigated in the past years. For instance, the author of [9] discussed the satisfiability problem of XPath expressions and obtained polynomial time (PTIME) bounds of some fragments of XPath. In [11], the authors addressed a tree pattern formalism with expressiveness incomparable to XPath and showed that the satisfiability problem is NP-complete for several restrictions of this pattern language in the absence of document type definition(DTDs). In [5] the authors explored the satisfiability problem associated with XPath in the presence of DTDs. However, these methods proposed by above references can not be used directly to solve the satisfiability of queries for AXML documents. The reason is that AXML documents contain embedded Web services, and we should consider all these intentional parts when checking the satisfiability of queries for AXML documents.

Tree automata theory has been attracted more attention along with the development of XML in recent years. Neven Frank surveyed the relationships among automata, Logic and XML in [16], and pointed out that tree automata can be used to define XML schema languages and check XML documents validation. Tree automata theory has been used to solve XML typechecking problem [13] and optimize XPath based on DTD [8]. In [8], the authors defined a new product tree automaton building from DTD and XPath query expression to generate the optimized form of XPath queries. This paper also define a new product tree automaton building from AXML schemas and tree pattern queries, but our goal is to decide satisfiability of queries for AXML documents conforming to a given schema by deciding the emptiness of the product automaton.

In a recent paper, we showed that tree automata also can be used to solve the problem of AXML document rewriting [12]. AXML document rewriting is to decide whether an AXML document can be translated into another one conforming to a given target schema by invoking some web services embedded in it. To solve it, we built two tree automata corresponding to the AXML document and AXML schema and checked whether their intersection is empty or not. Here we not only use an extended tree automaton to represent AXML schemas, but also use tree automata to abstract tree pattern queries and check the satisfiability of queries for AXML documents conforming to the schema by deciding the product automaton building from these two automata emptiness.

In the present paper, we use tree automata to represent query expressions and AXML schemas in a unified framework. More precisely, we firstly build a new tree automaton, AXML schema tree automaton(ASTA), to capture the set of documents conforming to the given schema. According to the tree pattern query expression, then we build a regular tree automaton which describes the set of documents that contain the query paths. To check whether there exist any AXML documents satisfying the query, it is sufficient to test whether the intersection of these two sets is empty or not. For this purpose, we build a product tree automaton from ASTA and tree pattern query tree automaton and test its emptiness. If the product automaton is non-empty, it shows that the query on this document is satisfied, otherwise, unsatisfied.

The main contributions of this paper are as follows:

- We propose a new tree automaton, ASTA, to capture the set of AXML documents conforming to the given schema.
- Given an AXML document t, AXML schema D and a tree pattern query q, we propose an algorithm that checks the satisfiability of query q for document t conforming to schema D based on ASTA.
- Finally, we perform experiments to verify our algorithm and show that this check takes a negligible fraction of the time needed for processing the query while often yielding substantial savings.

The paper is organized as follows: Section 2 describes the basic preliminaries knowledge about this paper. In Section 3 we propose an algorithm for building an equivalent ASTA corresponding to a given AXML schema. An algorithm of checking satisfiability of tree pattern queries for AXML documents conforming to a given schema is presented in Section 4. The experimental results are studied in Section 5. Section 6 concludes the paper.

2 Preliminaries

2.1 AXML Documents

We model AXML documents as ordered labeled trees with data and function nodes. The latter correspond to services calls. Firstly, we recall some definitions of trees from [7] which we used to abstract AXML documents and tree pattern queries in the following sections.

Definition 1 *(Ordered Tree.) A rooted tree is a tree in which one of the nodes is distinguished from the others. The distinguished node is called the root of the tree.*

An ordered tree is a rooted tree in which the children of each node are ordered.

A labeled tree is a rooted tree in which each node is assigned a label.

We assume the existence of some disjoint domains: $\mathcal{N}$ of nodes, $\mathcal{L}$ of labels, $\mathcal{F}$ of function names, and $\mathcal{D}$ of data values. We review the definition of AXML documents from [15].

Definition 2 *An AXML document t is an expression (T, λ), where $T = (N, E, <)$ is an ordered tree. $N \subset \mathcal{N}$ is a finite set of nodes, $E \subset N \times N$ are the edges, $<$ associates with each node in N a total order on its children, and $\lambda : N \rightarrow \mathcal{L} \cup \mathcal{F} \cup \mathcal{D}$ is a labeling function for the nodes, where only leaf nodes may be assigned data values from $\mathcal{D}$.*

Nodes with a label in $\mathcal{L}$ and nodes in $\mathcal{D}$ are called *data nodes* while those with a label in $\mathcal{F}$ are called *function nodes*. The children subtrees of a function node are the parameters of the call. When a function node is called, these subtrees are passed to it and returned results (one node or an output subtree) will replace the function node in the document.

For example, Figure 1 shows an AXML document. Function nodes are denoted as square nodes and data values are quoted. This document contains a list of hotels, some of which are given explicitly and some only intensionally, through an embedded call to the *GetHotels* function. The document details for each *hotel* its *name*, *address*, *rating*, and some nearby restaurants or museums. When the first *GetRestos* call is invoked with the address of the hotel as a parameter, it returns a list of *restaurant* elements to replace the function node, as showed in Figure 2.

Analogous to XML, AXML schemas represent the specification which AXML documents must conform to. To simplify the presentation, we consider a simple DTD-like schema specification from [15].

Definition 3 *An AXML document schema is an expression (L, F, τ), where $L \subset \mathcal{L}$ and $F \subset \mathcal{F}$ are finite set of labels and function names respectively, τ is a function that maps each label name $l \in L$ to a regular expression over $L \cup F$ or to the keyword "data"(for atomic data), and maps each function name $f \in F$ to a pair of such expressions, called the input and output types of f and denoted by $\tau_{in}(f)$ and $\tau_{out}(f)$.*

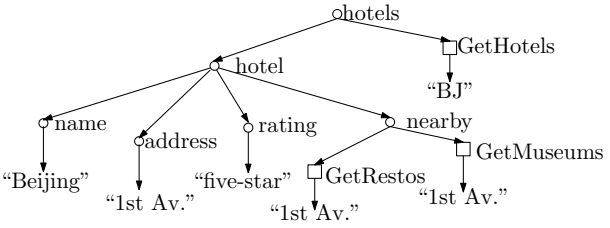

Figure 1: A sample AXML document

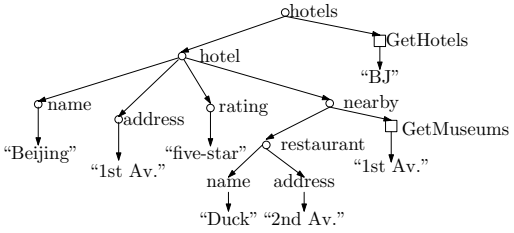

Figure 2: AXML document after a call

For instance, the following is an example of AXML schema.

Example 1 *An AXML schema $D = (L, F, \tau)$ defines as follows:*

data:

$\tau(hotels) = hotel^*.GetHotels,$
$\tau(hotel) = name.address.rating.nearby,$
$\tau(nearby) = restaurant^*.GetRestos.museum^*.$
$GetMuseums,$
$\tau(restaurant) = name.address,$
$\tau(museum) = name.address,$
$\tau(name) = \tau(address) = data,$
$\tau(rating) = data.$

functions:

$\tau_{in}(GetHotels) = data,$
$\tau_{out}(GetHotels) = hotel^*,$
$\tau_{in}(GetRestos) = data,$
$\tau_{out}(GetRestos) = restaurant^*,$
$\tau_{in}(GetMuseums) = data,$
$\tau_{out}(GetMuseums) = museum^*.$

Then AXML schema D defines the set of AXML documents conforming to it.

2.2 Tree Pattern Queries

We model queries as tree patterns described by Miklau and Suciu in [14].

Definition 4 *A* tree pattern query *is a labeled tree whose nodes are labeled by variable names, constants (element names and data values), or the label wildcard "$*$". The nodes labeled by variable names are called* variable nodes *and those labeled by element names and data values are called* constant nodes*. The tree also has a distinguished set of edges called* descendant edges *and a distinguished of nodes called the* result nodes*.*

Figure 3 shows a tree pattern query. Descendant edges are represented by double lines and result nodes are pointed by variables.

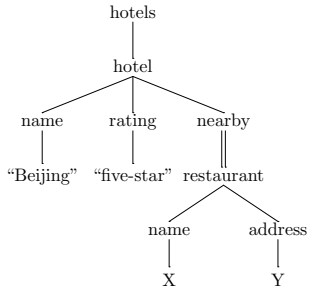

Figure 3: A tree pattern query

To represent the problem of satisfiability, we use the notion of query embedding from [1].

Definition 5 *Given a tree pattern query q and an AXML document t, an embedding of q into t is a tree homomorphism ρ from the nodes of q to the data nodes of t, mapping the root of q to that of t, preserving the parent-child and ancestor-descendant relationships (for regular and descendant edges, respectively.), mapping each constant node of q to a data node of t with the same label, and such that all the variable nodes of q with the same variable name are mapped to data nodes having identical labels.*

We say that an AXML document t *satisfies* a tree pattern query q provided there is an embedding ρ from q to t. A query q is *satisfiable* if there is an AXML document t that satisfies q. Then, our satisfiability problem can be defined as follows.

Problem: Given a tree pattern query q and an AXML document t conforming to schema D, we should check whether document t satisfies q.

According to above definition, for instance, there not exist any embedding from query q(showed in Figure 3) to document t(showed in Figure 1), that is, q is unsatisfiable for t. Observing document t, we can find that it contains three function nodes: *GetHotels*,*GetRestos* and *GetMuseums*. Furthermore, we can rewrite document t to t'(showed in Figure 2) if we invoke function node *GetRestos* according to schema D defined in Example 1. Now query q is satisfiable for document t'. If we choose to invoke function node *GetHotels*, it will return a list of *hotel* nodes containing some *restaurant* descendant nodes or function node *GetRestos*, which also maybe satisfy query q.

From above observation, we can see that decide the satisfiability of a given query for AXML documents is more intricate than XML documents, because we should consider all these function nodes that maybe return some useful data contributing to the query. We also can see that AXML schemas determine whether an AXML document conforming to it satisfies the given query or not. It is important to note that queries only match the data nodes of the document, because function nodes are the only means to get the data they represent.

2.3 Tree Automata

Tree automata are extended from traditional string automata and the main differences between them are transition functions of states. In string automata, there is just one state before a transition, while tree automata support a set of states. This special property makes tree automata more capable to deal with tree-structure data such as XML documents. We recall the definition of *unranked non-deterministic tree automata(NTA)* from [6].

Definition 6 *An unranked non-deterministic tree automaton is a tuple $B = (Q, \Sigma, \delta, A)$, where Q is a finite set of states, Σ is a finite alphabet, $A \subseteq Q$ is the set of final or accepted states, and δ is a function $Q \times \Sigma \rightarrow 2^{Q^*}$, or denoted as $f(q_1, \ldots, q_n) - a \rightarrow q$, such that f is a regular string language over Q, for every $a \in \Sigma$ and $q_1, \ldots, q_n \in Q$.*

The set of all accepted trees is denoted by $L(B)$ and is called a *regular tree language*. In the sequence of this paper, when we say tree automata we always mean unranked non-deterministic tree automata.

In [8] the authors pointed that we can build an equivalent tree automaton which describes all the documents

that contain the query paths conforming to a given tree query pattern expression. This is illustrated next.

Example 2 *Let us consider a simple tree pattern query* $q = a//*/[d][e]$ *which first checks if the root is labeled with* a *and it has a* b*-child node; if not, it returns the empty set; otherwise, it returns all* b*'s descendants that have both a* d*-child and a* e*-child: the* d *and* e *children may occur in any order. Then we define* $B = (Q, \Sigma, \delta, A)$ *as follows:*

$Q = \{s_0, s_1, s_2, s_3, s_4\}$,

$\Sigma = \{a, b, d, e\}$,

$\delta = \{\epsilon - d \rightarrow s_3, \epsilon - e \rightarrow s_4, s_3 s_4 - all \rightarrow s_2, s_2 - all \rightarrow s_2, s_2 - b \rightarrow s_1, s_1 - a \rightarrow s_0\}$,

$A = \{s_0\}$,

Here label ***all*** *represents arbitrary symbol in* Σ*, that is,* ***all*** *maybe one of* a, b, d, e *.*

Intuitively, B *works as follows:* B *assigns* s_3 *and* s_4 *to* d*-labeled leaf and* e*-labeled leaf; further,* B *assigns* s_2 *to any labeled node with children nodes labeled* d *and* e*; and,* B *assigns* s_1 *to* b*-labeled node with descendant nodes labeled* d *and* e*. Finally,* B *accepts when the root is labeled with* a*.*

An AXML schema defines the set of AXML documents conforming to it. It is already known that a tree automaton represents the set of trees accepted by it. Since AXML documents can be abstracted as labeled trees, we can extend tree automata to represent AXML schema and then this type of tree automata will only accept all such AXML documents that conform to the schema.

3 ASTAs

In this section, we represent the construction of ASTA, which efficiently describes the set of the AXML documents conforming to a given schema.

The difficulty of building ASTA is how to define the transition functions. From Section 2.3 we know that in the definition of a tree automaton, the transition functions are a set of regular string languages over states. To define these transition functions, we make use of *nondeterministic finite automata(NFAs)* to represent the transition functions of ASTA for their equivalent to regular expressions; we refer unfamiliar reader to [10] for details. Since AXML schema introduces function elements, we should process these parts firstly. For each function node in a schema, we preprocess it as follows: for example, the definition of data element v is $\tau(v) = a.b.G.e$, G is a function element and whose output type is a disjunctive formula such as $\tau_{out}(G) = c|d$, when we define the transition function of element v, besides to define the function $\{q_a.q_b.q_G.q_e - lab(v) \rightarrow q_v\}$ ($lab(v)$ means the label of element v), we should also add the following two functions to it: $\{q_a.q_b.q_c.q_e - lab(v) \rightarrow q_v\}$ and $\{q_a.q_b.q_d.q_e - lab(v) \rightarrow q_v\}$. Then we get the final transition function of node v is $\{q_a.q_b.(q_G|q_c|q_d).q_e - lab(v) \rightarrow q_v\}$. In other words, if element v has a function element as its sub-element, then all of the function output types should be added to the set of transition functions of v. When a function element is invoked, one of the output instances will be returned to replace the function element and become a sub-element of the parent element of this function element. If the parameters of function elements and returned results contain other function elements, we should begin with the deepest function elements and recursively process outward.

Given an AXML schema D, the algorithm showed in Figure 4 represents the process of constructing an equivalent ASTA.

Algorithm Construction of ASTA.
Input: AXML schema $D = (L, F, \tau)$.
Output: ASTA $B = (Q, \Sigma, \delta, A)$.
1: $\Sigma = L \cup F$
2: **for each** element $v_i \in D$ with definition $\tau(v_i) = data$ in D **do**
3: $\delta = \delta \cup \{\epsilon\} - lab(v_i) \rightarrow q_i$
4: $Q = Q \cup \{q_i\}$
5: **end for**
6: **for each** element v_j with subelements $v_{j_1}, \ldots, v_{j_m}$ in D **do**
7: $\delta := \delta \cup \{f(q_{j_1} \cdots q_{j_m}) - lab(v_j) \rightarrow q_j\}$, where $q_{j_1}, \ldots, q_{j_m} \in Q$
8: **end for**
9: **for each** node element v_f with output type R_{f_j} occurring in v_j's subelements **do**
10: $\delta := \delta \cup \{\{f(q_{j_1} \cdots q_{j_{i-1}} \cdot q_{f_j} \cdots q_{j_m}) - lab(v_j) \rightarrow q_j\}$
11: **end for**
12: $A = \{q_{root}\}$

Figure 4: Building ASTA

We give the intuition of the above algorithm next. Firstly, we defined the finite alphabet Σ and processed the *atomic data element* to initialize ASTA B. Then processed the *data elements* in D by building a finite string automaton to represent the regular expression occurring in the content models. For each function element, we built two finite string automata to represent the input type language and output type language respectively. Finally, we defined the final state q_{root} to be the root element of D, because the state of root element is the only accepted state.

The algorithm can be performed in polynomial time, because the only work needs is to scan schema D once.

Theorem 1 *Given an AXML schema* D*, we can build an equivalent ASTA in PTIME.*

The correctness of above algorithm is obvious, because unranked tree automata are an abstraction of the various XML schema proposals. Moreover, the unranked

tree automata are equivalent to the specialized DTDs of Papakonstantinou and Vianu [17]. In our paper, AXML schema is an extended of DTDs and naturally can be abstracted by extended contex-free grammars.

4 Checking Satisfiability

According to Section 3, given an AXML schema D, we can build an equivalent ASTA B_1 that represents all such documents conforming to D. On the other hand, we can construct another regular tree automaton B_2, according to a given tree pattern query q, which defines all such documents containing the query paths of q. Then the intersection of $L(B_1)$ and $L(B_2)$ is all the documents both conforming to D and satisfying query q. Therefore, we translate the satisfiability problem of query q for AXML documents into the non-empty problem of product tree automaton B building from B_1 and B_2. Algorithm showed in Figure 5 represents this idea.

Algorithm Checking satisfiability.
Input: AXML schema D and tree pattern query q.
Output: **true** or **false**.
1: Building following automata:
2: (a) Tree automaton $B_1 = (Q_1, \Sigma, \delta_1, A_1)$, corresponding q.
3: (b) ASTA $B_2 = (Q_2, \Sigma, \delta_2, A_2)$, corresponding D.
4: (c) An empty tree automaton $B = (Q, \Sigma, \delta, A)$.
5: $Q = \{(q_1, q_2) | \exists q_1 \in Q_1, q_2 \in Q_2, s.t. \{\epsilon\} - a \rightarrow q_1$ and $\{\epsilon\} - a \rightarrow q_2$ for all $a \in \Sigma\}$,
6: $\delta = \{\{\epsilon\} - a \rightarrow (q_1, q_2)\}$, for all $(q_1, q_2) \in Q\}$,
7: $A = \{(q_1, q_2) | q_1 \in A_1, q_2 \in A_2\}$.
8: **for all** transitions s.t. $\{f((q_{11}, q_{21}) \dots (q_{1n}, q_{2n}) - a \rightarrow (q_1, q_2) | f(q_{11}, \dots, q_{1n}) - a \rightarrow q_1 \in \delta_1, \exists f(q'_{21}, \dots, q'_{2n}) - a \rightarrow q_2 \in \delta_2, f(q_{21}, \dots, q_{2n}) \subseteq f(q'_{21}, \dots, q'_{2n})$ and $\{(q_{11}, q_{21}), \dots, (q_{1n}, q_{2n}) \in Q\}\}$ **do**
9: $Q = \{Q \cup (q_1, q_2)\}$
10: $\delta = \delta \cup \{f((q_{11}, q_{21}) \dots (q_{1n}, q_{2n})) - a \rightarrow (q_1, q_2)\}$
11: **if** $L(B) \neq \phi$ **then**
12: **return true**
13: **else**
14: **return false**
15: **end if**
16: **end for**

Figure 5: Checking satisfiability of AXML documents

We should notice that the product automaton B is a regular tree automaton, because only final XML documents would contribute to the query. That is, we only care about whether there exists an XML document, which can be obtained by invoking some function nodes in the original AXML document, satisfying the given query or not.

Example 3 *Given a query expression $q = //a/[b][c]$, and an AXML schema D defines as follows:*

data nodes: $\tau(p) = d.e$, $\tau(e) = G|a$, $\tau(a) = b.c$, $\tau(b) = \tau(c) = \tau(e) = data$,

function nodes: $\tau_{in}(G) = data$, $\tau_{out}(G) = a$, we check the satisfiability of q for document t conforming to D.

(1)Firstly, we build the automata corresponding to q and D respectively. Automaton $B_1 = (Q_1, \Sigma_1, \delta_1, A_1)$, where $Q_1 = \{s_1, s_2, s_3, s_4\}$, $\delta_1 = \{\epsilon - b \rightarrow s_4, \epsilon - c \rightarrow s_3, s_3.s_4 - a \rightarrow s_2, s_2 - all \rightarrow s_1, s_1 - all \rightarrow s_1\}$. $A_1 = \{s_1\}$. ASTA $B_2 = (Q_2, \Sigma, \delta_2, A_2)$, where, $Q_2 = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7\}$, $\delta_2 = \{\epsilon - b \rightarrow q_6, \epsilon - c \rightarrow q_7, q_6.q_7 - a \rightarrow q_5, \epsilon - G \rightarrow q_4, q_4 | q_5 - e \rightarrow q_3, \epsilon - d \rightarrow q_2, q_2.q_3 - p \rightarrow q_1\}$, $A_2 = \{q_1\}$.

(2)Next, we build the product automaton B of B_1 and B_2. Automaton $B = (Q, \Sigma, \delta, A)$, where $Q = \{\langle q_6, s_4\rangle, \langle q_7, s_3\rangle, \langle q_5, s_2\rangle, \langle q_3, s_1\rangle, \langle q_1, s_1\rangle\}$, $\delta = \{\epsilon - b \rightarrow \langle q_6, s_4\rangle, \epsilon - c \rightarrow \langle q_7, s_3\rangle, \langle q_6, s_4\rangle.\langle q_7, s_3\rangle - a \rightarrow \langle q_5, s_2\rangle, \langle q_5, s_2\rangle - e \rightarrow \langle q_3, s_1\rangle, \langle q_3, s_1\rangle - p \rightarrow \langle q_1, s_1\rangle\}$, $A = \{\langle q_1, s_1\rangle\}$.

(3)Finally, we test the automaton B is non-empty or not.

We analyze the complexity and correctness of the algorithm of Figure 5 next.

Theorem 2 *The complexity of the above algorithm is in PTIME.*

Proof. The algorithm contains three tree automata constructions, everyone of which can be done in PTIME according to Theorem 1. In the initial process,the algorithm only scans elements once and it needs a linear time. Then there is a for-loop that makes a linear number of scanning the set of transition functions, it also can be done in PTIME. Finally, deciding whether tree automata B is empty has already been known in PTIME[16].

Theorem 3 *The algorithm returns true if and only if an AXML document t conforming to schema D satisfies the given query q.*

Proof. Suppose that document t conforms to D and satisfies the given query q, we prove $L(B) \neq \phi$. According to algorithm of Figure 4, we know that ASTA B_1 defines the set of AXML documents conforming to D, hence $t \in L(B_1)$. On the other hand, tree automaton B_2 built from query q represents all the documents that contain the query paths. From our assumption, document t has an accept running both on tree automata B_1 and B_2, namely we have got $t \in L(B_2)$. Therefore, the product tree automaton B built from B_1 and B_2 is not empty, that is $L(B) \neq \phi$.

Suppose $L(B) \neq \phi$, we prove that t conforming to D satisfies query q. From $L(B) \neq \phi$, we can build an AXML document t', produced by t by calling some function nodes in it, satisfies $t' \in L(B)$. However, $L(B) = L(B_1) \cap L(B_2)$, we have $t' \in L(B_1)$ and

$t' \in L(B_2)$. The former shows that t' is an AXML document that conforms to schema D, and the latter proves that t' satisfies the query q.

5 Experiments

Experiments environment To study the effectiveness of testing satisfiability, we systematically ran a range of experiments to measure the savings and overhead of various documents size.

The documents schema we use have a structure very similar to Example 1. They all consist of *hotel* elements, which may include calls to the functions: *GetRestos*, and *GetMuseums*; we also use an extra *GetHotels* function whose result is a set of hotels. We use the ToXgene[4] XML generator to produce documents. We fix the number of function calls in the document by instructing ToXgene to be about 30% of the total number of elements. Since we check the satisfiability of queries, we do not consider the form of queries that contain the label wildcard "*". We report experiments performed with three queries:

q1: /hotels/hotel
q2: /hotels/hotel[rating='five-star']
q3: //hotel/[rating='five-star'][address='1st Av.']

We used Xalan to evaluate our queries and implemented our satisfiability tests in Java. We performed our experiments on a workstation running Fedora Core 6 with 256MBytes of RAM and a PIII 1.2GHz CPU. All values reported were the average of 5 trials after dropping the maximum and minimum.

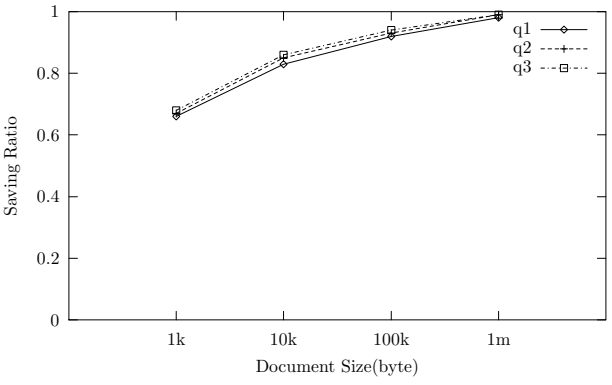

Figure 6: Saving ratio experiment results

Saving & Overhead Ratios Let c be the time taken to determine the satisfiability of a query q and let e be the time it takes to evaluate the query over the document. The *saving ratio* S_q obtained by using satisfiability check on unsatisfiable queries is defined as $S_q = \frac{e-c}{e}$ and the *overhead ratio* incurred by doing satisfiability check on satisfiable queries is defined as $O_q = \frac{e+c}{e}$. Intuitively, the closer to 1 the two ratios are the better.

Figure 6 and Figure 7 show the variation of savings and overhead ratios with document size for the three queries. On unsatisfiable queries, satisfiability check leads to phenomenal savings. Our saving ratio is between about 0.6 and 0.9. On satisfiable check, we expect the overhead ratio to decrease as the document size increases. Indeed, this behavior can be observed from the figures. Overall, our results show that the overhead is a negligible fraction of the evaluation time.

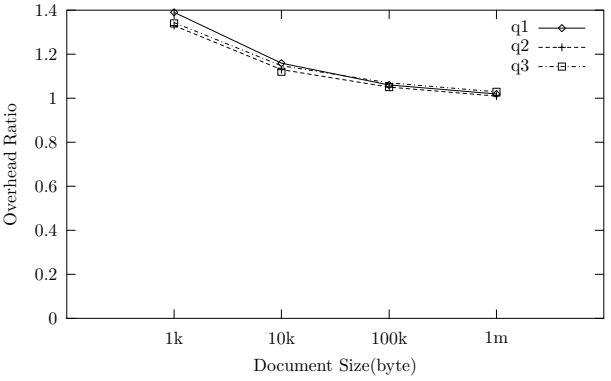

Figure 7: Overhead ratio experiment results

The same conclusions were also obtained from the experiments on the various number of function nodes in document. Though the evaluation time of queries and satisfiable check time were increased, the saving and overhead ratios are almost same to above experiments. We omit these results for brevity.

6 Conclusion

We investigated the problem of satisfiability of tree patter queries for AXML documents and proposed efficient algorithms to solve it based on tree automata theory. We complemented our analytical results with an extensive set of experiments. While satisfiability checking can effect substantial savings in query evaluation, our results demonstrated that it incurred negligible overhead over satisfiable queries.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful suggestions on earlier versions of this paper. Zhongxiao Hao is supported by the Natural Science Foundation of Heilongjiang Province of China (F00-06).

References

[1] Abiteboul, S., Benjelloun, O., Cautis, B., Manolescu, I., Milo, T., and Preda, N. Lazy Query Evaluation for Active XML. *Proceedings of the ACM SIGMOD International Conference on Management of Data(SIGMOD)*, pages 227–238, 2004.

[2] Abiteboul, S., Benjelloun, O., Manolescu, I., Milo, T., and Weber, R. Active XML: Peer-to-Peer Data and Web Services Integration. *Proceedings of the International Conference on Very Large DataBases(VLDB)*, pages 1087–1090, 2002.

[3] Abiteboul, S., Benjelloun, O., and Milo, T. Positive Active XML. *Proceedings of the ACM Symposium on Principles of Database Systems(PODS)*, pages 35–45, 2004.

[4] Barbosa, D., Mendelzon, A. O., Keenleyside, J., and Lyons, K. A. Toxgene: An extensible template-based data generator for xml. In *Proceedings of the Fifth International Workshop on the Web and Databases(WebDB2002)*, pages 49–54, 2002.

[5] Benedikt, M., Fan, W., and Geerts, F. XPath Satisfiability in the Presence of DTDs. *Proceedings of the ACM Symposium on Principles of Database Systems(PODS)*, pages 25–36, 2005.

[6] Bruggemann-Klein, A., Murata, M., and Wood, D. Regular Tree and Regular Hedge Languages over Unranked Alphabets. Technical report, HKUST-TCSC-2001-0, The Hong Kong University of Science and Technology, 2001.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction To Algorithms*. High Education Press, Beijing, second edition, 2002.

[8] Gao, J., Yang, D., Tang, S., and Wang, T. DTD Based Deterministic XPath Rewriting and Logical Optimization. *Journal of Software(Chinese)*, 15(12):1860–1868, 2004.

[9] Hidders, J. Satisfiability of XPath Expressions. *Proceedings of the 9th International Conference on Data Base Programming Languages (DBPL)*, pages 21–36, 2003.

[10] Hopcroft, J. E., Motwani, R., and Ullman, J. *Introduction to Automata Theory, Languages, and Computation*. Tsinghua University Press, Beijing, second edition, 2002.

[11] L. Lakshmanan, H. W., G. Ramesh and Zhao, Z. On Testing Satisfiability of Tree Pattern Queries. *Proceedings of the International Conference on Very Large DataBases(VLDB)*, pages 120–131, 2004.

[12] Ma, H., Hao, Z., and Zhu, Y. Active XML Document Rewriting Based on Tree Automata Theory. *Wuhan University Journal Natural Sciences*, 11(5):1325–1329, 2006.

[13] Martens, W. and Neven, F. Typechecking Top-Down Uniform Unranked Tree Transducers. *Proceedings of International Conference on Database Theory(ICDT)*, pages 64–78, 2003.

[14] Miklau, G. and Suciu, D. Containment and Equivalence for a Fragment of XPath. *JACM*, 51(1):2–45, 2004.

[15] Milo, T., Abiteboul, S., Amann, B., Benjelloun, O., and Ngoc, F. D. Exchanging Intensional XML Data. *Proceedings of the ACM SIGMOD International Conference on Management of Data(SIGMOD)*, pages 289–300, 2003.

[16] Neven, F. Automata, Logic and XML. *Proceedings of the 16th Intl Workshop Computer Science Logic(CSL)*, pages 2–26, 2002.

[17] Papakonstantinou, Y. and Vianu, V. DTD Inference for Views of XML Data. In *PODS '00: Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 35–46, New York, NY, USA, 2000. ACM Press.