Efficient Revalidation of XML Documents

Mukund Raghavachari and Oded Shmueli

Abstract—We study the problem of schema revalidation where XML data known to conform to one schema must be validated with respect to another schema. Such revalidation algorithms have applications in schema evolution, query processing, XML-based programming languages, and other domains. We describe how knowledge of conformance to an XML Schema may be used to determine conformance to another XML Schema efficiently. We examine both the situation where an XML document is modified before it is revalidated and the situation where it is unmodified.

Index Terms—XML, XML Schema, validation, updates, subtyping.

1 Introduction

XML has emerged as a universal data interchange format across many domains—databases, Web Services, messaging systems, etc. A core feature of XML is the ability to constrain the structure of data using specifications such as XML Schema [1] and DTDs [2]. These formalisms allow XML processors to *validate* data to ensure that data satisfy expected structural and integrity constraints.

Consider the following XQuery [3] expression, which illustrates a common usage pattern:

```
import schema namespace default="http://source"
import schema namespace target="http://target"
FOR $b in document("purchaseOrder.xml")/purchaseOrder
RETURN validate {<target:billOfSale> {$b/billTo}
                 {$b/items}</target:billOfSale>}
```

In the sample query, the two **import** statements import XML Schema declarations into the processor. The subsequent **FOR** statement retrieves data from a document (perhaps, resident in a database) and constructs a result XML fragment from the document's contents. The semantics of the **validate** operator is to validate the constructed XML fragment with respect to the XML Schema declaration of billOfSale found in the XML Schema associated with the "target" prefix.

One mechanism for implementing the **validate** operation in an XQuery processor is to traverse the constructed XML fragment and validate each element in the XML tree explicitly. When XML data are large, this process can be inefficient—the entire XML fragment must be materialized in memory and validated. In many situations, one may have knowledge of the validity of the source document (in the example, purchaseOrder.xml) with respect to some schema. For example, when a document is inserted into a database, it might be validated with respect to some schema (in our example, http://source). In this paper, we show that knowledge of the validity of an XML fragment with respect to one schema can be used to improve the performance of validation of that XML fragment with respect to another schema. Efficient revalidation of XML data is important in many domains:

- *Schema Evolution.* As a schema evolves over time, data that conforms to older versions of the schema may need to be verified with respect to the new schema.
- *Information Integration.* An intracompany schema used by a business might differ slightly from a standard, external schema—XML data valid with respect to one may need to be checked for conformance to the other.
- *Incremental Validation.* The problem of incremental validation is to examine whether XML data known to correspond to a schema is still valid with respect to that schema after a series of updates [4]. When the label of a node is modified, the subtree rooted at the node is known to conform to one type and its consistency with a new type must be verified.
- *XML Programming Languages.* In programming languages that support XML as a first-class construct, one needs to verify whether a value known to be of one XML type belongs to another XML type. For example, Levin and Pierce [5] and Frisch [6] consider the problem of efficient compilation of pattern matching (detecting whether a value matches a pattern), which is similar to the schema revalidation problem.

The scenario we consider is the following: An XML fragment that is valid with respect to a *source* schema type A must be validated with respect to a *target* schema type B. We refer to this as the *schema revalidation* problem. If the XML fragment may be modified before revalidation, we refer to this problem as *schema revalidation with modifications*. We present techniques that take advantage of similarities (and differences) between the schema types A and B to avoid validating portions of a document explicitly. Consider the two XML Schema element declarations for purchaseOrder shown in Fig. 1. The sole difference between

- *M. Raghavachari is with the IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532. E-mail: raghavac@us.ibm.com.*
- *O. Shmueli is with the Technion Israel Insititute of Technology, Taub Building, Haifa 3200, Israel. E-mail: oshmu@cs.technion.ac.il.*

Manuscript received 26 May 2005; revised 4 Jan. 2006; accepted 9 Sept. 2006; published online 19 Jan. 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0211-0505. Digital Object Identifier no. 10.1109/TKDE.2007.1004.

1041-4347/07/$25.00 © 2007 IEEE Published by the IEEE Computer Society

```
<xsd:element name=”purchaseOrder” type=”POType1”/>
<xsd:complexType name=”POType1”>
  <xsd:sequence>
    <xsd:element name=”shipTo” type=”USAddress”/>
    <xsd:element name=”billTo” type=”USAddress” minOccurs=”0”/>
    <xsd:element name=”items” type=”Items”/>
  </xsd:sequence>
</xsd:complexType>
```

(a)

```
<xsd:element name=”purchaseOrder” type=”POType2”/>
<xsd:complexType name=”POType2”>
  <xsd:sequence>
    <xsd:element name=”shipTo” type=”USAddress”/>
    <xsd:element name=”billTo” type=”USAddress”/>
    <xsd:element name=”items” type=”Items”/>
  </xsd:sequence>
</xsd:complexType>
```

(b)

Fig. 1. Schema fragments defining a purchaseOrder element in (a) source schema and (b) target schema.

the two is that where the billTo element is optional in the schema of Fig. 1a, it is required in the schema of Fig. 1b. Not all XML documents valid with respect to the first schema are valid with respect to the second—only those with a billTo element are valid. Given a document valid according to the schema of Fig. 1a, an ideal validator would only verify the presence of a billTo element and ignore the validation of the other components.

We will focus on the validation of XML documents with respect to the structural constraints of XML schemas. At the core of our techniques are efficient algorithms for revalidating strings known to be recognizable by a deterministic finite state automaton (DFA) according to another DFA. We present an optimal algorithm for revalidation with respect to DFAs. The amount of auxiliary state required by our algorithm for schema revalidation is proportional to the product of the size of the two schema types (typically much smaller than the size of the XML data).

We describe our algorithms in terms of an abstraction of XML Schema, *Abstract XML Schema*, which models the structural constraints of XML Schema. We have run experiments comparing our techniques with full validation using an XML parser, Xerces 2.4. In our experiments, our algorithms achieve 30-95 percent performance improvement over Xerces 2.4.

The contributions of this paper are the following:

1. An abstraction of XML Schema, *Abstract XML Schema*, which captures the structural constraints of XML Schema precisely.
2. Efficient algorithms for schema revalidation (with and without modifications) of XML with respect to XML Schema types. The auxiliary state information required by our algorithm is proportional to the size of the schemas, and is independent of the size of the document. When the schemas under consideration are similar, our algorithms use the similarities to avoid traversing the document where possible. When the schemas are not similar, the algorithms detect quickly that revalidation will fail, again avoiding unnecessary traversals of the document.
3. Efficient algorithms for revalidation of strings with and without modifications according to deterministic finite state automata. These algorithms are essential for the efficient revalidation of the content of elements.
4. Experiments validating the utility of our solutions.

Structure of the Paper. In Section 2, we discuss related work. In Section 3, we define Abstract XML Schema and the formalisms used in the paper. In Section 4, we define the algorithm for schema revalidation (with and without modifications). The algorithm relies on an efficient solution to the problem of string revalidation according to finite state automata, which is provided in Section 5. We discuss the complexity and optimality of our algorithms in Section 6. We report on experiments in Section 7, and conclude in Section 8.

2 Related Work

Schema revalidation is related to the problem of *incremental validation* of XML, studied by Papakonstantinou and Vianu [7], and by Barbosa et al. [8]. Given a document that is known to conform to a schema and a sequence of updates applied to the document, the incremental validation problem is to determine whether the modified document is still valid with respect to the original schema. Previous algorithms maintain state information with each node in the document that is used to validate a document incrementally [4], [8]. In general, the amount of auxiliary state stored with a document can be quite large. When the schema in question uses a restricted language of regular expressions for content models, for example, *conflict-free* [8] or *local* [4] regular expressions, optimizations can be applied to improve the efficiency of incremental validation.

Kane et al. [9] use a technique based on query modification for handling the incremental validation problem. Bouchou and Halfeld-Ferrari Alves [10] present an algorithm that validates each update using a technique based on tree automata. Again, both algorithms consider only the case where the schema to which the document must conform after modification is the same as the original schema. Moreover, they validate each update incrementally, and do not handle the case of validation after a sequence of updates has been performed.

Our algorithm handles the more general *revalidation* problem, but is applicable to the incremental validation problem. The primary technique introduced in this paper requires little or no state stored with document nodes, but may be less efficient than previous algorithms for incremental validation in validating a document incrementally after a sequence of updates. When the reduction of the storage size of a document is essential, for example, with main-memory XML processors, our algorithm may offer an appropriate solution for the incremental validation problem.

Levin and Pierce [5] and Frisch [6] study the efficient compilation of pattern matching expressions in programming languages. Given a variable or expression of a known type, the compiler must generate code that, at runtime, detects which of a set of provided patterns match the value referred to by the variable or expression. A pattern may be viewed as a type as well and, therefore, the problem reduces to detecting when a value known to be of one type is also a value of another type. Many of the optimizations performed by the compiler are similar to those required for the revalidation problem—for maximum efficiency, one ought to inspect only the portions of a value that are truly necessary.

While the problems of schema revalidation and efficient pattern matching are similar, the different models used for XML values makes comparison difficult. The previous work on pattern matching operate on tree automata, which are *ranked*, whereas XML Schemas are *unranked*. The canonical mechanism for handling XML documents and XML Schemas in terms of tree automata is to convert XML documents into binary trees and XML Schema types into appropriate tree automata states [11]. As a result of the conversion, these algorithms do not take advantage of the fact that XML Schema content models are 1-unambiguous—the stated worst-case complexity in their algorithms is exponential. Our formulation takes advantage of efficient containment algorithms for 1-unambiguous regular expressions—in this situation, our revalidation algorithm is polynomial (as discussed in Section 6). Furthermore, pattern-matching compilation algorithms do not address revalidation with modifications since the languages considered have immutable values.

The subsumption of XML schema types used in our algorithm for revalidation is similar to Kuper and Siméon's notion of type subsumption [12]. Their type system is more general than our Abstract XML Schema. A subsumption mapping is provided between types such that if one schema is subsumed by another and values conforming to the subsumed schema are annotated with types, then by applying the subsumption mapping to these type annotations, one obtains an annotation for the subsuming schema. Our solution is more general in that we do not require either schema to be subsumed by the other, but do handle the case where this occurs. Furthermore, we do not require type annotations on nodes. Finally, we consider the notion of disjoint types in addition to subsumption in the revalidation of documents.

3 PRELIMINARIES

We present basic definitions including the abstractions used for XML Schemas and documents.

3.1 Deterministic Finite State Automata

A *deterministic finite state automaton (DFA)* is a 5-tuple $(Q, \Sigma, \delta, q^0, F)$, where Q is a finite set of states, Σ is a finite alphabet of symbols, $q^0 \in Q$ is the start state, $F \subseteq Q$ is a set of final, or accepting, states, and δ is the transition function. A *string* is a finite sequence of 0 or more symbols of Σ, where ϵ is the string with 0 symbols. We denote a string s with n symbols as $s_1 \cdot s_2 \cdot \ldots \cdot s_n$. δ is a total function from $Q \times \Sigma$ to Q. We use $\delta(q, \sigma) \to q'$, where $q, q' \in Q$, $\sigma \in \Sigma$, to denote that δ maps (q, σ) to q'. For string s and state q, $\delta(q, s) \to q'$ denotes the state q' reached by operating on s one symbol at a time, where $\delta(q, \epsilon) = q$ for all q in Q. A string s is *accepted* by a DFA if $\delta(q^0, s) \in F$; s is *rejected* by a DFA if s is not accepted by it. The *size* of a DFA is $|Q| \times |\Sigma|$.

The language accepted (or recognized) by a DFA M, denoted $L(M)$, is the set of strings accepted by M. We define $L_M(q)$, $q \in Q$, as $\{s \mid \delta(q, s) \in F\}$. For a DFA M, if a string $s = s_0 \cdot \ldots \cdot s_n$ is in $L(M)$, and $\delta(q^0, s_0 \cdot s_1 \cdot \ldots \cdot s_i) = q'$, $1 \le i < n$, then $s_{i+1} \cdot \ldots \cdot s_n$ is in $L_M(q')$ (we will drop the subscript M when the automaton is clear from the context).

A state $q \in Q$ is *reachable* if there exists $s \in \Sigma^*$, $\delta(q^0, s) \to q$. It is straightforward to convert a DFA with unreachable states into an equivalent one that contains only reachable states in time linear in the size of the automaton [13]. We, therefore, assume that all states in Q are reachable. A state $q \in Q$ is a *dead state* if for all $s \in \Sigma^*$, $\delta(q, s) \notin F$—no final state is reachable from a dead state. Again, we can identify and remove all dead states in time linear in the size of the automaton [13].

Given two DFAs,

$$M_1 = (Q_1, \Sigma_1, \delta_1, q_1^0, F_1) \text{ and } M_2 = (Q_2, \Sigma_2, \delta_2, q_2^0, F_2),$$

one can derive an *intersection automaton* M, such that M accepts exactly the language $L(M_1) \cap L(M_2)$. Intuitively, an intersection automaton evaluates a string on both M_1 and M_2 in parallel and accepts only if both would. Formally, $M = (Q, \Sigma, \delta, q^0, F)$ where

$$q^0 = (q_1^0, q_2^0), Q = Q_1 \times Q_2, F = F_1 \times F_2, \text{ and}$$
$$\delta((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)).$$

Since M_1 and M_2 are deterministic, M is deterministic as well.

3.2 Immediate Decision Automata

We introduce *immediate decision automata* as modified DFAs that accept or reject strings without necessarily scanning the entire string. Formally, an immediate decision automaton M_{immed} is a 7-tuple, $(Q, \Sigma, \delta, q^0, F, IA, IR)$, where $IA, IR \subseteq Q$ are disjoint sets. As M_{immed} processes s, if after evaluating a proper prefix x of s (i.e., $x \neq s$), $\delta(q^0, x) \in IA$, then M_{immed} accepts s. M_{immed} rejects s after evaluating a proper prefix x of s if $\delta(q^0, x) \in IR$. If M_{immed} processes all of s, it accepts s if $\delta(q^0, s) \in F$; otherwise, it rejects s. We can derive an immediate decision automaton from a DFA so that both automata accept the same language.

Definition 1. *Let $M = (Q, \Sigma, \delta, q^0, F)$ be a DFA. The derived immediate decision automaton is*

$$M_{immed} = (Q, \Sigma, \delta, q^0, F, IA, IR),$$

where

- $IA = \{q \in Q \mid L_M(q) = \Sigma^*\}$ *and*
- $IR = \{q \in Q \mid L_M(q) = \emptyset\}$.

It can be easily shown that M_{immed} and M accept the same language.

For a DFA M, we can determine all states that belong to IA and IR efficiently in time linear in the size of the

TABLE 1
Abstract XML Schema Type for XML Schema Type POType1 of Fig. 1a

Type	Σ_τ	$regexp_\tau$	$types_\tau$
POType1	shipTo billTo items	(shipTo (ϵ + billTo) items)	shipTo $\rightarrow$ USAddress billTo $\rightarrow$ USAddress items $\rightarrow$ Items

automaton. The members of IR are the dead states of M. The members of IA can be identified as follows: First, construct the complement of M, $\overline{M} = (Q, \Sigma, \delta, q^0, \overline{F})$, where $\overline{F} = Q - F$. IA is the set of dead states in $\overline{M}$.

3.3 Ordered Labeled Trees

We abstract XML documents as *ordered labeled trees*, where an ordered labeled tree over a finite alphabet Σ is a pair $T = (t, \lambda)$, where $t = (N, E)$ is an ordered tree consisting of a finite set of nodes, N, and a set of edges E, and $\lambda : N \rightarrow \Sigma \cup \{\chi\}$ is a function that associates a label with each node n of N. The label, χ, which can only be associated with leaves of t, represents XML Schema simple values. We use $root(T)$ to denote the root node of the tree t. We will abuse the notation to allow $\lambda(T)$ to denote the label of the root node of the ordered labeled tree T. We use $r(t_1, t_2, \ldots, t_k)$, $k \geq 0$, to denote an ordered tree with root node r and subtrees $t_1 \ldots t_k$, where $r()$ denotes an ordered tree with a root r that has no children. The *depth* of a node n in the tree is defined as the number of edges between the root of the tree and n. The *height* of a tree is defined as the maximum depth of a leaf in the tree. We use $\mathcal{T}_\Sigma$ to represent the set of all ordered labeled trees.

Two trees, $T = (t, \lambda)$ and $T' = (t', \lambda')$, are *equal*, denoted by $T \equiv T'$, if:

- $t = n()$ and $t' = n'()$ and $\lambda(n) = \lambda'(n')$, or
- $t = r(t_1, t_2 \ldots, t_k)$ and

$$t' = r'(t'_1, t'_2, \ldots, t'_k), \lambda(r) = \lambda'(r'),$$

  and for all i, $1 \leq i \leq k$, $t_i \equiv t'_i$.

3.4 Abstract XML Schema

Our abstraction of XML Schema, *Abstract XML Schema*, is a 4-tuple, $(\Sigma, \mathcal{T}, \rho, \mathcal{R})$, where

- Σ is the alphabet of element labels (tags).
- $\mathcal{T}$ is the set of types defined in the schema.
- ρ is a set of type declarations, one for each $\tau \in \mathcal{T}$, where $\rho(\tau)$ is either a *simple* type of the form τ: *simple*, or a *complex type* of the form $\tau : (regexp_\tau, types_\tau)$, where
  - $regexp_\tau$ is a regular expression [13] over Σ. We sometimes refer to $regexp_\tau$ as the *content model of* τ. $L(regexp_\tau)$ denotes the language associated with $regexp_\tau$.
  - Let $\Sigma_\tau \subseteq \Sigma$ be the set of element labels appearing in any string of $L(regexp_\tau)$. Then, $types_\tau : \Sigma_\tau \rightarrow \mathcal{T}$ is a function that assigns a type to each element label used in the type declaration of τ. The function, $types_\tau$, abstracts the notion of XML Schema that each child of an element can be assigned a type based on its label without considering the child's content. It also models the XML Schema constraint that if two children of an element have the same label, they must be assigned the same type.
- $\mathcal{R} : \Sigma \rightarrow \mathcal{T}$ is a partial function which states which element labels can occur as the root element of a valid tree according to the schema (that is, the domain of $\mathcal{R}$), and the type the root element is assigned (from the range of $\mathcal{R}$).

Consider the XML Schema fragment of Fig. 1a. The function $\mathcal{R}$ maps global element declarations to their appropriate types, that is, $\mathcal{R}(\text{purchaseOrder}) =$ POType1. Table 1 shows the type declaration for POType1 in our formalism.

Abstract XML Schemas do not explicitly represent simple types, such as xsd:integer. For simplicity of exposition, we have assumed that all XML Schema atomic and simple types are represented by a single simple type. Handling atomic and simple types, restrictions on these types and relationships between the values denoted by these types is a straightforward extension. We do not address the identity constraints (such as key and keyref constraints) of XML Schema; it is an area of future work. Other features of XML Schema such as substitution groups, subtyping, and namespaces can be integrated into our model. A discussion of these issues is beyond the scope of the paper.

Definition 2. *The set of ordered labeled trees that are* valid *with respect to a type τ is defined as follows:*

If τ is a simple type,

$$valid(\tau) = \{(t, \lambda) \in \mathcal{T}_\Sigma \mid t = n_1(n_2()), \lambda(n_1) \in \Sigma, \lambda(n_2) = \chi\}.$$

If τ is a complex type,

$$valid(\tau) = \bigcup_{m \geq 0} valid^m(\tau),$$

where $valid^m(\tau)$ is defined inductively.

For $m = 0$, $valid^0(\tau) = \{(t, \lambda) \in \mathcal{T}_\Sigma \mid t = n(), \lambda(n) \in \Sigma\}$ if $\epsilon \in L(regexp_\tau)$; otherwise, $valid^0(\tau) = \emptyset$.

For $m > 0$,

$$valid^m(\tau) = \{(t, \lambda) \in \mathcal{T}_\Sigma | t = n(t_1, t_2, \ldots, t_k), k > 0\},$$

such that:

- $height(t) = m$.
- $\lambda(n), \lambda(t_1), \ldots, \lambda(t_k) \in \Sigma$, *and*

$$\lambda(t_1) \cdot \lambda(t_2) \cdot \ldots \cdot \lambda(t_k) \in L(regexp_\tau).$$

- *Let* $\tau_i = types_\tau(\lambda(t_i))$, $1 \leq i \leq k$. *If* τ_i *is a simple type,* $t_i \in valid(\tau_i)$. *If* τ_i *is a complex type,* $t_i \in valid^p(\tau_i)$, *where* $p = height(t_i)$.

An ordered labeled tree, T, is *valid with respect to a schema* $S = (\Sigma, \mathcal{T}, \rho, \mathcal{R})$ if $\mathcal{R}(\lambda(T))$ is defined and $T \in valid(\mathcal{R}(\lambda(T)))$. Note that if τ is a complex type, and $L(regexp_\tau)$ contains the empty string ϵ, $valid(\tau)$ contains *all* trees of height 0, where the root node has a label from Σ, that is, τ may have an *empty content model*. The following is straightforward given the definition of validity.

Proposition 1. *If there is a tree* $T \in valid(\tau)$*, then for all* $\sigma \in \Sigma$*, there is a tree* $T' \in valid(\tau)$*, where* T' *is identical to* T*, except that the label of the root node in* T' *is* σ.

We are interested only in *productive* types, τ, where $valid(\tau) \neq \emptyset$. We assume that for $S = (\Sigma, \mathcal{T}, \rho, \mathcal{R})$, all $\tau \in \mathcal{T}$ are productive (there is a straightforward algorithm for converting a schema with types that are nonproductive into one that contains only productive types [14]).

Pseudocode for validating an ordered, labeled tree with respect to an Abstract XML Schema follows: *constructstring* is a utility method (not shown) that creates a string from the labels of the root nodes of a sequence of trees (it returns ϵ if the sequence is empty). Note that if a node has no children, the body of the **foreach** loop will not be executed. In Line 12, if $\mathcal{R}(\lambda(T))$ is undefined, $doValidate$ returns false.

```
1  boolean validate(τ : type, e : node)
2      if (τ is a simple type)
3          if (children(e) = {n()}, λ(n) = χ) return true
4          else return false
5      if (constructstring(children(e)) ∉ L(regexp_τ))
6          return false
7      foreach child e' of e
8          if (¬validate(types_τ(λ(e')), e'))
9              return false
10     return true
11 boolean doValidate(S : schema, T : tree)
12     return validate(R(λ(T)), root(T))
```

4 XML Schema Revalidation

Given two Abstract XML Schemas, $S = (\Sigma, \mathcal{T}, \rho, \mathcal{R})$ and $S' = (\Sigma, \mathcal{T}', \rho', \mathcal{R}')$, and an ordered labeled tree, T, that is valid according to S, our algorithm validates T with respect to S and S' in parallel. Suppose that during the validation of T with respect to S', one must validate a subtree of T, T', with respect to a type τ'. Let τ be the type assigned to T' during the validation of T with respect to S. If one can assert that every ordered labeled tree that is valid according to τ is also valid according to τ', then one can immediately deduce the validity of T' according to τ'. Conversely, if no ordered labeled tree that is valid according to τ is also valid according to τ', then one can stop the validation immediately since T' will not be valid according to τ'.

We use *subsumed type* and *disjoint type* relationships to avoid traversals of subtrees of T.

Definition 3. *A type* τ *is* subsumed *by a type* τ'*, denoted* $\tau \preceq \tau'$*, if* $valid(\tau) \subseteq valid(\tau')$*. Note that* τ *and* τ' *may belong to different schemas. Two types,* τ *and* τ'*, are* disjoint*, denoted* $\tau \oslash \tau'$*, if* $valid(\tau) \cap valid(\tau') = \emptyset$*. Again,* τ *and* τ' *may belong to different schemas.*

In the following sections, we present algorithms for determining whether an Abstract XML Schema type is subsumed by another or is disjoint from another, which will be used efficient schema revalidation of an ordered labeled tree, with and without updates.

4.1 Schema Revalidation

Our algorithm relies on relations, R_{sub} and R_{dis}, that capture precisely all subsumed type and disjoint type information with respect to the types defined in $\mathcal{T}$ and $\mathcal{T}'$.

4.1.1 Computing the R_{sub} Relation

Definition 4. *Given two schemas,* $S = (\Sigma, \mathcal{T}, \rho, \mathcal{R})$ *and* $S' = (\Sigma, \mathcal{T}', \rho', \mathcal{R}')$*, a relation* $R \subseteq \mathcal{T} \times \mathcal{T}'$ *is* well-founded *with respect to the schemas* S *and* S' *if for all* $(\tau, \tau') \in R$ *one of the following two conditions hold:*

1. τ, τ' *are both simple types.*[1]
2. τ *and* τ' *are both complex types,* $L(regexp_\tau) \subseteq L(regexp_{\tau'})$ *and* $\forall \sigma \in \Sigma$*, where* $types_\tau(\sigma)$ *is defined (and, hence,* $types_{\tau'}(\sigma)$ *is defined),*

$$(types_\tau(\sigma), types_{\tau'}(\sigma)) \in R.$$

Observe that a well-founded relation R must be a finite relation since there are finitely many types. Also, observe that if R_1 and R_2 are both well-founded, so is $R_1 \cup R_2$. Therefore, there is a *largest* well-founded relation, which we will denote by R_{sub}.

Proposition 2. *If* $\tau \preceq \tau'$*, and* τ *and* τ' *are complex types, then* $L(regexp_\tau) \subseteq L(regexp_{\tau'})$.

Proof. We prove the proposition by contradiction. If $L(regexp_\tau) \not\subseteq L(regexp_{\tau'})$, there is a string s such that $s \in L(regexp_\tau)$ and $s \notin L(regexp_{\tau'})$. Since all types are assumed to be productive, we can construct a tree $t = r(t_1, t_2, \ldots, t_k)$ such that $\lambda(t_1) \cdot \lambda(t_2) \cdots \lambda(t_k) = s$, and for all t_i, $1 \leq i \leq k$, $t_i \in valid(types_\tau(\lambda(t_i)))$. By the definition of validity, $t \in valid(\tau)$, and $t \notin valid(\tau')$, contradicting the assumption that $\tau \preceq \tau'$. □

Proposition 3. *If* $\tau \preceq \tau'$*, and* τ *and* τ' *are complex types, then for all* $\sigma \in \Sigma$*, where* $types_\tau(\sigma)$ *is defined,* $types_\tau(\sigma) \preceq types_{\tau'}(\sigma)$.

Proof. Assume $\tau \preceq \tau'$, where τ and τ' are complex types. Let there be a $\sigma \in \Sigma$, where $types_\tau(\sigma) = \nu$. By Proposition 2, $L(regexp_\tau) \subseteq L(regexp_{\tau'})$ and, therefore, $types_{\tau'}(\sigma)$ must be defined. Let $types_{\tau'}(\sigma) = \nu'$. We show that a contradiction arises if $\nu \not\preceq \nu'$.

If $\nu \not\preceq \nu'$, there must be at least one tree (t, λ) such that $(t, \lambda) \in valid(\nu)$ and $(t, \lambda) \notin valid(\nu')$. Let $s \in L(regexp_\tau)$ be a string that uses the symbol σ. Since we assume that all types are productive, we can construct a tree $t' =$

1. As mentioned before, for exposition, we have merged all simple types into a common *simple type*. It is straightforward to extend the definition of subsumption so that the various XML Schema atomic and simple types, and the subtyping relationships among them are used.

$r(t_1, t_2, \ldots, t_k)$ such that $\lambda(t_1) \cdot \lambda(t_2) \cdots \lambda(t_k) = s$, and there exists t_i, $1 \le i \le k$ such that $\lambda(t_i) = \sigma$ and $t_i = t$. In other words, we construct a tree such that one of the children of the root node is a node labeled σ, which is the root of a tree that is in $valid(\nu)$, but not in $valid(\nu')$. By the definition of validity (Definition 2), $t' \in valid(\tau)$ and $t' \notin valid(\tau')$. We have obtained a contradiction of the assumption that $\tau \preceq \tau'$, thus completing the proof. □

The following theorem states that the R_{sub} relation captures precisely the notion of subsumption:

Theorem 1. $(\tau, \tau') \in R_{sub}$ *if and only if* $\tau \preceq \tau'$.

Proof. (Only if) Recall that $\tau \preceq \tau'$ denotes that $\forall T$, $T \in valid(\tau) \Rightarrow T \in valid(\tau')$. If τ and τ' are both simple types, by item 1 in the definition of $valid$ (Definition 2), both of them denote the same set of trees, and the statement is trivially true. We use induction on the height of trees T to show that for complex types, τ and τ', $(\tau, \tau') \in R_{sub}$ implies that $\forall T$, $T \in valid(\tau) \Rightarrow T \in valid(\tau')$.

- $h = 0$. If $T \in valid(\tau)$, and T is of height 0, $\epsilon \in L(regexp_\tau)$. By the definition of R_{sub}, $\epsilon \in L(regexp_{\tau'})$ and, therefore, $T \in valid(\tau')$.
- $h > 0$. Assume that for all trees, T, of height up to h, $(\tau, \tau') \in R_{sub} \Rightarrow (T \in valid(\tau) \Rightarrow T \in valid(\tau'))$. Consider a tree $t = e(t_1, t_2, \ldots, t_k)$ of height h that is in $valid(\tau)$. Since $(\tau, \tau') \in R_{sub}$, we are assured that $L(regexp_\tau) \subseteq L(regexp_{\tau'})$ and, therefore, $\lambda(t_1) \cdot \lambda(t_2) \cdots \lambda(t_k) \in L(regexp_{\tau'})$. Consider any t_i, $1 \le i \le k$, where $\omega = types_\tau(\lambda(t_i))$ and

$$\nu = types_{\tau'}(\lambda(t_i)).$$

By the definition of R_{sub}, $(\tau, \tau') \in R_{sub}$ implies $(\omega, \nu) \in R_{sub}$. If ω is a simple type, then by the definition of R_{sub}, ν is also a simple type, and $t_i \in valid(\nu)$. If ω is a complex type, by the inductive hypothesis, $t_i \in valid(\nu)$. Since for each t_i, we can show that

$$t_i \in valid(types_{\tau'}(\lambda(t_i))),$$

we can conclude that $T \in valid(\tau')$, thus completing the induction.

(If) If τ and τ' are not both simple types, or are not both complex types, according to the definition of $valid$, no tree belonging to τ can be valid according to τ'. Therefore, $\tau \not\preceq \tau'$. If τ and τ' are simple types, by definition, $(\tau, \tau') \in R_{sub}$.

If τ and τ' are complex types, $\tau \preceq \tau'$, and (τ, τ') is *not* in R_{sub}, then we derive a contradiction. We construct a sequence of sets $R_0, R_1, \ldots$, where $R_0 = \{(\tau, \tau')\}$. The construction will ensure that 1) $R_i \subseteq R_{i+1}$ and 2) $(\omega, \nu) \in R_i$ implies $\omega \preceq \nu$ (which certainly holds for $i = 0$). We construct R_i from R_{i-1} as follows: $(\tau_1, \tau_2) \in R_i$ if $(\tau_1, \tau_2) \in R_{i-1}$. Furthermore, if there exists $(\omega, \nu) \in R_{i-1}$ and $\sigma \in \Sigma$ such that $types_\omega(\sigma) = \tau_1$ is defined, then by Proposition 2, $types_\nu(\sigma) = \tau_2$ is defined. We add (τ_1, τ_2) to R_i. Observe that by Proposition 3, $\tau_1 \preceq \tau_2$, which maintains condition 2).

Since there is a finite number of types in S and S', at some point, k, R_k will equal R_{k+1}. By construction of R_k, for all $(\tau_i, \tau_j) \in R_k$, $\tau_i \preceq \tau_j$. For all $(\tau_i, \tau_j) \in R_k$, either τ_i and τ_j are both simple types, or by Proposition 2, $L(regexp_{\tau_i}) \subseteq L(regexp_{\tau_j})$ and the construction ensures that for all $\sigma \in \Sigma$, where $types_{\tau_i}(\sigma)$ is defined, $(types_{\tau_i}(\sigma), types_{\tau_j}(\sigma)) \in R_k$. Therefore, by construction, R_k is well-founded.

As R_{sub} is well-founded, the set $R_{sub} \cup R_k$ is also well-founded. Since $R_{sub} \cup R_k$ contains (τ, τ'), which by assumption is not in R_{sub}, R_{sub} is not the *largest* well-founded relation. We have a contradiction and, therefore, (τ, τ') must belong to R_{sub}. □

We now present an algorithm for computing the R_{sub} relation. The algorithm starts with a subset of $\mathcal{T} \times \mathcal{T}'$ and refines it successively until R_{sub} is obtained.

1. Let $R_{sub} \subseteq \mathcal{T} \times \mathcal{T}'$ be the maximal relation such that $(\tau, \tau') \in R_{sub}$ implies that either both τ and τ' are simple types, or both of them are complex types.
2. For $(\tau, \tau') \in R_{sub}$, if $L(regexp_\tau) \not\subseteq L(regexp_{\tau'})$, remove (τ, τ') from R_{sub}.
3. For each (τ, τ'), if there is $\sigma \in \Sigma$, $types_\tau(\sigma) = \omega$ and $types_{\tau'}(\sigma) = \nu$ (after Step 2, $L(regexp_\tau) \subseteq L(regexp_{\tau'})$ must be true and, therefore, $types_{\tau'}(\sigma)$ must be defined), and $(\omega, \nu) \notin R_{sub}$, remove (τ, τ') from R_{sub}.
4. Repeat Step 3 until no more tuples can be removed from the relation R_{sub}.

4.1.2 Computing the R_{dis} Relation

Rather than computing R_{dis} directly, we compute its complement.

Definition 5. *Given two schemas,* $S = (\Sigma, \mathcal{T}, \rho, \mathcal{R})$ *and* $S' = (\Sigma, \mathcal{T}', \rho', \mathcal{R}')$, *let* $R_{nondis} \subseteq \mathcal{T} \times \mathcal{T}'$ *be defined by the following procedure. The procedure begins with an empty relation and adds tuples until* R_{nondis} *is obtained.*

1. *Let* $R_{nondis} = \emptyset$.
2. *Add all* (τ, τ') *to* R_{nondis} *such that* $\tau : simple \in \rho$, $\tau' : simple \in \rho'$.
3. *For each* $(\tau, \tau') \in \mathcal{T} \times \mathcal{T}'$, *let*

   $$P = \{\sigma \in \Sigma \mid (types_\tau(\sigma), types_{\tau'}(\sigma)) \in R_{nondis}\}.$$

   If $L(regexp_\tau) \cap L(regexp_{\tau'}) \cap P^* \neq \emptyset$ *add* (τ, τ') *to* R_{nondis}.
4. *Repeat Step 3 until no more tuples can be added to* R_{nondis}.

Theorem 2. $\tau \oslash \tau'$ *if and only if* $(\tau, \tau') \notin R_{nondis}$ *(recall that* $\tau \oslash \tau'$ *denotes that* $valid(\tau) \cap valid(\tau') = \emptyset$*).*

Proof. (Only if) If $\tau \oslash \tau'$, but $(\tau, \tau') \in R_{nondis}$, we derive a contradiction. We prove by induction on the stage, k, in which (τ, τ') is added to R_{nondis} that

$$valid(\tau) \cap valid(\tau') \neq \emptyset.$$

- $k = 1$. Initially, R_{nondis} is empty, so the addition must be due to the fact that both τ and τ' are simple types. Obviously, $valid(\tau) \cap valid(\tau') \neq \emptyset$.

- $k > 1$. The addition is due to Step 3. This means that the intersection of the content models of τ and τ' is nonempty. If both τ and τ' contain the empty string ϵ by the definition of $valid$, both τ and τ' contain all trees of height 0, where the root node has a label from Σ, and therefore, $valid(\tau) \cap valid(\tau') \neq \emptyset$. Otherwise, there is at least one string s in the content model of both τ and τ' such that for each σ used in s, $(types_\tau(\sigma), types_{\tau'}(\sigma))$ is already in R_{nondis}. By the induction hypothesis, this implies that each type pair is such that

$$valid(types_\tau(\sigma)) \cap valid(types_{\tau'}(\sigma)) \neq \emptyset.$$

We can, therefore, construct a tree in $valid(\tau) \cap valid(\tau')$ using the trees in

$$valid(types_\tau(\sigma)) \cap valid(types_{\tau'}(\sigma)),$$

for each σ used in s.

(If) Suppose $(\tau, \tau') \notin R_{nondis}$, but there is a tree $T \in valid(\tau) \cap valid(\tau')$, we derive a contradiction. By the definition of R_{nondis}, τ and τ' must be complex types. We prove by induction on the height h of subtrees T' of T that the complex types ω and ν assigned to a node at height h, when validating T according to τ and τ', respectively, are such that $(\omega, \nu) \in R_{nondis}$. This will imply that $(\tau, \tau') \in R_{nondis}$, which is a contradiction.

- $h = 0$. Since T' is valid with respect to both τ and τ', the types assigned to the root of T', ω, and ν, must support an empty content model: $\epsilon \in L(regexp_\omega)$ and $\epsilon \in L(regexp_\nu)$. By Definition 5, $(\omega, \nu) \in R_{nondis}$.
- $h > 0$. We assume that the induction hypothesis holds for all subtrees of T, of height up to h, that are marked with a complex type. Let n, the root of T', be of type ω (respectively, ν) when typed according to τ (respectively, τ'). There exists a string s that is in $L(regexp_\omega) \cap L(regexp_\nu)$ because the labels of the children nodes of n are valid according to both ω and ν. Moreover, the types assigned to the symbols of s according to ω and ν are pairwise in R_{nondis}, either because they are both simple types, or if they are both complex types, as a consequence of the induction hypothesis. Therefore, $L(regexp_\omega) \cap L(regexp_\nu) \cap P^*$ (as defined in Definition 5) is not empty, and $(\omega, \nu) \in R_{nondis}$. This completes the induction.

As a result of the induction, when $h = height(T)$, $(\tau, \tau') \in R_{nondis}$, which is a contradiction. □

4.1.3 Algorithm for Schema Revalidation

Given the relation R_{sub} between types defined in Abstract XML Schemas S and S', let the schema S be *subsumed* by S' if for all $\sigma \in \Sigma$, where $\mathcal{R}(\sigma)$ is defined, $\mathcal{R}'(\sigma)$ is defined and $\mathcal{R}(\sigma) \preceq \mathcal{R}'(\sigma)$. Similarly, let S be *disjoint* from S' if for all $\sigma \in \Sigma$, where $\mathcal{R}(\sigma)$ is defined, $\mathcal{R}'(\sigma)$ is either not defined or $\mathcal{R}(\sigma) \oslash \mathcal{R}'(\sigma)$.

1 **boolean** $revalidate(\tau : \text{type}, \tau' : \text{type}, e : \text{node})$
2   if $\tau \preceq \tau'$ **return** true
3   if $\tau \oslash \tau'$ **return** false
4   **if** $(constructstring(children(e)) \notin L(regexp_{\tau'}))$
5   **return** false
6   **foreach** child e' of e, in order,
7   **if** $(\neg revalidate(types_\tau(\lambda(e')), types_{\tau'}(\lambda(e')), e'))$
8   **return** false
9   **return** true
10 **boolean** $doRevalidate(S : \text{schema}, S' : \text{schema}, T : \text{tree})$
11   **if** (S is subsumed by S') **return** true
12   **if** (S is disjoint from S') **return** false
13   **return** $revalidate(\mathcal{R}(\lambda(T)), \mathcal{R}'(\lambda(T)), root(T))$

Fig. 2. Pseudocode for schema revalidation.

If S is subsumed by S', we can certify that a tree T known to be valid according to S is valid according to S' without inspecting the tree. Similarly, if S is disjoint from S', we can reject all trees known to be valid according to S immediately. Otherwise, if at any time, a subtree of the document that is valid with respect to τ from schema S is being validated with respect to τ' from schema S', and $\tau \preceq \tau'$, then the subtree need not be examined (since by definition, the subtree belongs to $valid(\tau')$). On the other hand, if $\tau \oslash \tau'$, the document can be determined to be invalid with respect to S' immediately. Pseudocode for incremental revalidation of the document is provided in Fig. 2. Again, *constructstring* is a utility method (not shown) that creates a string from the labels of the root nodes of a sequence of trees (returning ϵ if the sequence is empty). We can verify the content of e with respect to $regexp_{\tau'}$ (Line 4, Fig. 2) using techniques for finite automata-based revalidation, as described in Section 5.

Note that in $revalidate$, we do not consider the case where τ' is a simple type explicitly. If τ' is a simple type and τ is a simple type, $\tau \preceq \tau'$, and the procedure would return $true$ from Line 2. Otherwise, if τ is not a simple type, $\tau \oslash \tau'$, and the procedure would return $false$ from Line 3.

4.2 Schema Revalidation with Modifications

Given an ordered, labeled tree, T, that is valid with respect to an Abstract XML Schema S, and a sequence of insertions and deletions of nodes, and renaming of element labels, we discuss how the tree may be validated efficiently with respect to a new Abstract XML Schema S'. The updates permitted are the following:

1. Change the label of a specified node to a new label.
2. Insert a new leaf node before, after, or as the first child of a node.
3. Delete a specified leaf node.

Given a sequence of updates to T, we encode the modifications on T that result in a tree T' by extending Σ with special element tags of the form Δ^a_b, where $a, b \in \Sigma \cup \{\epsilon, \chi\}$. A node in T' with label Δ^a_b represents the modification of the element tag a in T with the element tag b in T'. Similarly, a node in T' with label Δ^ϵ_b represents a newly inserted node with tag b, and a label Δ^a_ϵ denotes a node deleted from T. The labels of unmodified nodes remain unchanged. By discarding all nodes with label Δ^a_ϵ and converting the labels of all other nodes labeled Δ^*_b into b, one obtains the tree that is the result of performing the modifications on T (where $*$ represents any symbol).

We assume the availability of a function *modified* on the nodes of T' that returns for each node whether any part of the subtree rooted at that node has been modified. The function *modified* can be implemented efficiently as follows: We generate the Dewey decimal number of each node dynamically as we process. Whenever an original tree node is updated, we keep it in a trie [15] according to its Dewey decimal number. A child insertion or deletion is considered an update. To determine whether a descendant of an original tree node v was modified, the trie is searched according to the Dewey decimal number of v. There is a modification in the subtree rooted at a node if there is a path in the trie corresponding to the Dewey decimal number of the node. Note that we can navigate the trie in parallel to navigating the XML tree.

The algorithm for efficient schema revalidation with modifications validates $T' = (t', \lambda')$ with respect to S and S' in parallel. While processing a subtree of T', t'', with respect to types τ from S and τ' from S', one of the following cases apply:

1. If $modified(t'')$ is false, the subtree t'' is unchanged. Since $t'' \in valid(\tau)$ when checked with respect to S, we can treat the validation of t'' as an instance of the schema revalidation problem (without modifications) described in Section 4.1.
2. Otherwise, if $\lambda'(t'') = \Delta^a_\epsilon$, we do not need to validate the subtree (with respect to τ') since that subtree has been deleted.
3. Otherwise, if $\lambda'(t'') = \Delta^\epsilon_b$, since the label denotes that t'' is a newly inserted subtree, we have no knowledge of its validity with respect to any other schema. Therefore, we validate the whole subtree explicitly from scratch.
4. Otherwise, if $\lambda'(t'') = \Delta^a_b, a, b \in \Sigma \cup \{\chi\}$, or

$$\lambda'(t'') = \sigma, \sigma \in \Sigma \cup \{\chi\},$$

since elements may have been added or deleted from the original content of the node, we must ensure that the content of t'' is valid with respect to τ'. If τ' is a simple type, the content is validated to ensure that it satisfies Definition 2. Otherwise, if $t'' = n(t_1, \ldots, t_k)$, we check that $t_1, \ldots, t_k$ fit into the content model of τ' as specified by $regexp_{\tau'}$. In verifying the content model, we check whether

$$Proj_{new}(t_1) \cdot \ldots \cdot Proj_{new}(t_k) \in L(regexp_{\tau'}),$$

where $Proj_{new}(t_i)$ is $\lambda'(t_i)$ if $\lambda'(t_i) \in \Sigma \cup \{\chi\}$, and b, if $\lambda'(t_i) = \Delta^a_b, a, b \in \Sigma \cup \{\epsilon, \chi\}$. $Proj_{old}$ is defined analogously. If the content model check succeeds, and τ is also a complex type, we recursively validate t_i, $1 \leq i \leq k$ with respect to $types_\tau(Proj_{old}(t_i))$ from S and $types_{\tau'}(Proj_{new}(t_i))$ from S' (note that if $Proj_{new}(t_i)$ is ϵ, we do not have to validate t_i since it has been deleted in T'). If τ is not a complex type, we validate each t_i according to $types_{\tau'}(Proj_{new}(t_i))$ explicitly.

5 Finite Automata Revalidation

We now examine the revalidation problem (with and without modifications) for strings verified with respect to DFAs. The algorithms described in this section support efficient content model checking for XML Schemas. Since XML Schema content models correspond directly to DFAs, we only address that case (similar techniques can be applied to nondeterministic finite state automata).

5.1 Revalidation

The problem that we address is the following: Given two DFAs, $M_1 = (Q_1, \Sigma_1, \delta_1, q_1^0, F_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, q_2^0, F_2)$, and a string $s \in L(M_1)$, does $s \in L(M_2)$? One could, of course, scan s using M_2 to determine acceptance by M_2. When many strings that belong to $L(M_1)$ are to be validated with respect to $L(M_2)$, it can be more efficient to preprocess M_1 and M_2 so that the knowledge of s's acceptance by M_1 can be used to determine its membership in $L(M_2)$.

Our method for the efficient validation of a string $s = s_1 \cdot s_2 \cdot \ldots \cdot s_n$ in $L(M_1)$ with respect to M_2 relies on evaluating M_1 and M_2 on s in parallel. Assume that after processing a prefix $s_1 \cdot \ldots \cdot s_i$ of s, we are in a state $q_1 \in Q_1$ in M_1, and a state $q_2 \in Q_2$ in M_2. Then, we can:

1. Accept s immediately if $L(q_1) \subseteq L(q_2)$, because $s_{i+1} \cdot \ldots \cdot s_n$ is guaranteed to be in $L(q_1)$ (since M_1 accepts s), which implies that $s_{i+1} \cdot \ldots \cdot s_n$ will be in $L(q_2)$. By definition of $L(q)$, M_2 will accept s.
2. Reject s immediately if $L(q_1) \cap L(q_2) = \emptyset$. Then, $s_{i+1} \cdot \ldots \cdot s_n$ is guaranteed not to be in $L(M_2)$ and, therefore, M_2 will not accept s.

We construct an immediate decision automaton, M_{immed} from the intersection automaton M of M_1 and M_2, with IR and IA based on the two conditions above.

Definition 6. *Let $M = (Q, \Sigma, \delta, q^0, F)$ be the intersection automaton derived from two DFAs M_1 and M_2. The derived immediate decision automaton is*

$$M_{immed} = (Q, \Sigma, \delta, q^0, F, IA, IR),$$

where

- $IA = \{(q_1, q_2) \in Q \mid L(q_1) \subseteq L(q_2)\}$ *and*
- $IR = \{(q_1, q_2) \in Q \mid (q_1, q_2) \text{ is a dead state}\}$.

The proof of the following theorem is straightforward.

Theorem 3. *For all $s \in L(M_1)$, M_{immed} accepts s if and only if $s \in L(M_2)$.*

The following proposition is useful for efficient computation of the members of IA.

Proposition 4. *For any state, $(q_1, q_2) \in Q$, $L(q_1) \subseteq L(q_2)$ if and only if $\forall s \in \Sigma^*$, for states q_a and q_b such that $\delta((q_1, q_2), s) \rightarrow (q_a, q_b)$, if $q_a \in F_1$ then $q_b \in F_2$.*

Proof. (Only if) Consider $s \in \Sigma^*$, either $s \in L(q_1)$ or $s \notin L(q_1)$. If $s \in \Sigma^*$ is in $L(q_1)$, then by definition of $L(q)$, $\delta_1(q_1, s) \in F_1$. Since $L(q_1) \subseteq L(q_2)$, then similarly, $\delta_2(q_2, s) \in F_2$. By definition of δ, $\delta((q_1, q_2), s) \rightarrow (q_a, q_b)$, $q_a \in F_1$, $q_b \in F_2$, thus proving the assertion. If $s \notin L(q_1)$, then $\delta_1(q_1, s) \notin F_1$, and $\delta((q_1, q_2), s) \rightarrow (q_a, q_b)$, $q_a \notin F_1$. The assertion holds trivially since $q_a \notin F_1$.

(If) Suppose $\forall s \in \Sigma^*$, the states q_a and q_b are such that

$$\delta((q_1, q_2), s) \rightarrow (q_a, q_b), q_a \in F_1 \Rightarrow q_b \in F_2.$$

Let $s \in L(q_1)$. If $\delta((q_1, q_2), s) \to (q_a, q_b)$, $q_a \in F_1$, then by the assumption, $q_b \in F_2$. This implies that $\delta_2(q_2, s) \in F_2$ and $s \in L(q_2)$. Therefore, $L(q_1) \subseteq L(q_2)$. □

Given two DFAs M_1 and M_2, we can preprocess M_1 and M_2 to construct the immediate automaton M_{immed} efficiently. The dead states of the intersection automaton of M_1 and M_2 are the members of IR. The set of states, IA is derived using Proposition 4. A state $(q_1, q_2) \in IA$ if for all states (q_1', q_2') reachable from (q_1, q_2), if q_1' is a final state of M_1, then q_2' is a final state of M_2. We can determine all such states in time linear in the size of the intersection automaton. An efficient algorithm for revalidation without modifications is to construct M_{immed} and use it to process strings s known to be in $L(M_1)$ to determine membership in $L(M_2)$.

5.2 Optimizing Revalidation

The performance of revalidation with respect to DFAs can be improved by storing auxiliary state with strings to assist with revalidation. Unlike the incremental validation problem, we do not know the target schema with which one must revalidate a priori. Therefore, the auxiliary state must be useful irrespective of the target schema. Given a string s known to be in $L(M)$ for a DFA M, we use a *trimmed automaton*, $\mathrm{TRIM}_{M,s}$, constructed from M and s.

Definition 7. *A* trimmed automaton

$$\mathrm{TRIM}_{M,s} = (Q_T, \Sigma, \delta_T, q^0, F_T)$$

constructed from a DFA $M = (Q, \Sigma, \delta, q^0, F)$ and a string $s \in L(M)$ is as follows:

- *Let $Q' \subseteq Q$ consist of all states of M that are traversed in the accepting computation of M on s. $Q_T = Q' \cup \{dead\}$, where dead is a special dead state.*
- *Let $\delta' \subseteq \delta$ be the partial function whose domain consists of the elements of $Q \times \Sigma$ used in the accepting computation of M on s; $\delta_T(q, \sigma)$, $q \in Q_T$, $\sigma \in \Sigma$, is defined as q' in case $\delta'(q, \sigma) \to q'$, and dead otherwise.*
- *$F_T = \{q\}$, where $\delta(q^0, s) \to q$. In other words, F_T is the singleton set containing the final state in the accepting computation of M on s.*

The size of $\mathrm{TRIM}_{M,s}$ is bounded by the size of M. $\mathrm{TRIM}_{M,s}$ represents the equivalence class of strings recognized by M that use the same set of transitions in an accepting computation as the accepting computation of M on s.

Instead of constructing an immediate automaton from M in revalidation, one can use $\mathrm{TRIM}_{M,s}$. $\mathrm{TRIM}_{M,s}$ is typically smaller than M and has fewer accepting states. The resulting computation of the immediate acceptance automaton constructed using $\mathrm{TRIM}_{M,s}$ can be expected to determine acceptance or rejection earlier than the immediate acceptance automaton constructed using M. Intuitively, the fact that on s, M will only use the transitions of δ' is used to optimize the immediate automaton.

5.3 Revalidation with Modifications

Consider the following variation of the revalidation problem. Given two DFAs, M_1 and M_2, a string $s \in L(M_1)$, $s = s_1 \cdots s_n$, is modified through insertions, deletions, and the renaming of symbols to obtain a string $s' = s_1' \cdots s_m'$. The question is whether $s' \in L(M_2)$?

As the updates are performed, it is straightforward to keep track of the leftmost location at which, and beyond to the right, no updates have been performed, that is, the *least* i, $1 \leq i \leq m$ such that $s_i' \cdot \ldots \cdot s_m' = s_{n-m+i} \cdot \ldots \cdot s_n$. The knowledge that $s \in L(M_1)$ is generally of no utility in evaluating $s_0' \cdot \ldots \cdot s_{i-1}'$ since the string might have changed drastically. The validation of the substring, $s_i' \cdot \ldots \cdot s_m'$, however, reduces to the revalidation problem without modifications.

To determine the validity of s' according to M_2, we first process M_2 to construct an immediate decision automaton, (as described in Section 3.2) $M_{2,immed}$. We also process M_1 and M_2 to generate an immediate decision automaton, M_{immed}, as described in Section 5.1. Given a string s' where the leftmost unmodified position is i, we:

1. Evaluate $s_1' \cdot \ldots s_{i-1}'$ using $M_{2,immed}$. That is, determine $q_2 = \delta_{2,immed}(q^0_{2,immed}, s_1' \cdot \ldots \cdot s_{i-1}')$. While scanning, $M_{2,immed}$ may immediately accept or reject, at which time, we stop scanning and return the appropriate answer. Otherwise, $M_{2,immed}$ scans $i-1$ symbols of s' and does not immediately accept or reject, and we proceed as follows.
2. Evaluate $s_1 \cdot \ldots \cdot s_{n-m+i-1}$ using M_1. That is, determine $q_1 = \delta_1(q_1^0, s_1 \cdot \ldots \cdot s_{n-m+i-1})$.
3. Scan $s_i' \cdot \ldots \cdot s_m'$ using M_{immed} starting in state $q' = (q_1, q_2)$.
4. If M_{immed} accepts, either immediately or by scanning all of s', then $s' \in L(M_2)$; otherwise, the string is rejected, possibly by entering an immediate reject state.

6 Discussion

We now examine the complexity of our schema revalidation algorithm, describe the way in which our algorithm may be considered optimal, and discuss extensions to our approach.

6.1 Complexity

We assume that for an XML tree, the label of each node, the first child of each node, and the next sibling of each node can be retrieved in constant time. The R_{sub} and R_{dis} relations (as defined in Section 4) can be recorded as matrices where the rows range over the types declared in one schema and the columns range over the types in the other schema. Assuming that the relations are precomputed, it is straightforward to see that the execution of the schema revalidation algorithm takes time linear in the size of the tree being revalidated.

The computation of the R_{sub} relation depends on detecting whether the language denoted by one regular expression is contained within the language denoted by another regular expression. In general, this containment problem is PSPACE-COMPLETE [13]. If we were, however, to restrict our attention to 1-unambiguous expressions [16]—all XML Schema content models are 1-unambiguous—containment between two 1-unambiguous regular expressions is in PTIME [17]. Since the computation of R_{sub} starts with a finite relation of cardinality polynomial in the size of the input schemas, and in each step, it removes at least one pair from the relation, the number of steps executed to compute R_{sub} is also polynomial in the size of the inputs. For 1-unambiguous regular expressions, therefore, the computation of R_{sub} is in PTIME.

The computation of the R_{dis} relation is in PTIME for arbitrary regular expressions since it mostly depends on checking whether the intersection of the languages denoted by two regular expressions is nonempty. One can construct the intersection automaton from the automata corresponding to the two regular expressions and verify that there is at least one string in the language accepted by the intersection automaton in polynomial time [13].

6.2 Optimality

We first consider the optimality of our algorithm for revalidation with respect to a DFA and, subsequently, consider the optimality of our schema revalidation algorithm.

6.2.1 DFA Revalidation

An immediate decision automaton M_{immed} derived from DFAs M_1 and M_2, with IA and IR as defined in Definition 6, is optimal in the sense that there can be no other deterministic Turing machine (DTM) D that scans a string s in a left-to-right order, that can determine whether s belongs to $L(M_2)$ earlier than M_{immed}. Observe that no limitation is imposed on the DTM and it can perform arbitrarily complex computations between scanning two adjacent string symbols. We say that a DTM D *determines* that s is in $L(M_2)$ (respectively, not) if it enters a special accept (respectively, reject) state, in which case it is said to accept (respectively, reject) s, and halts.

Proposition 5. *Let D be an arbitrary DTM that recognizes, by scanning strings left-to-right, for all strings $s \in L(M_1)$, whether $s \in L(M_2)$. For every string $s = s_1 \cdot s_2 \cdot \ldots \cdot s_n$ in $L(M_1)$, if D accepts or rejects s after scanning i symbols of s, $1 \leq i \leq n$, then M_{immed} scans at the most i symbols to make the same determination.*

Proof. Consider the case where D accepts s. Suppose M_{immed} scans more than i symbols to make the same determination. Then,

$$\delta(q^0, s_1 \cdot s_2 \cdot \ldots \cdot s_i) \rightarrow (q_1, q_2),$$

where $(q_1, q_2) \notin IA$. By the definition of IA in Definition 6, this implies that $L(q_1) \not\subseteq L(q_2)$. In other words, there exists a string $x \in L(q_1)$ that is not in $L(q_2)$. Since $x \notin L(q_2)$, $\delta(q^0, s_1 \cdot s_2 \cdot \ldots \cdot s_i \cdot x) \notin F$. Therefore, $s_1 \cdot s_2 \cdot \ldots \cdot s_i \cdot x$ will not be accepted by M_{immed} (F is the set of accepting states of M_{immed}) and is not in $L(M_2)$. DTM D would however accept $s_1 \cdot s_2 \cdot \ldots \cdot s_i \cdot x$ since it accepts after scanning s_i (without examining x). Therefore, D would accept a string not in $L(M_2)$, contradicting the assumption of the theorem.

Consider the case where D rejects s after scanning i symbols. Suppose M_{immed} takes more than i symbols to make the same determination. Then,

$$\delta(q^0, s_1 \cdot s_2 \cdot \ldots \cdot s_i) \rightarrow q',$$

where $q' \notin IR$. By the definition of IR, this implies that q' is not a dead state; there is at least one string $x \in L(q')$ and, therefore, $y = s_1 \cdot s_2 \cdot \ldots \cdot s_i \cdot x \in L(M_{immed})$. By Theorem 3, this string is in $L(M_2)$, but this string would not be accepted by D (which rejects after i steps). Therefore, D does not properly recognize whether $y \in L(M_2)$ for some $y \in L(M_1)$, contradicting the assumption of the theorem. □

1 **boolean** $revalidate(\tau : \text{type}, \tau' : \text{type}, e : \text{node})$
2   if $\tau \preceq \tau'$ **return** true
3   **if** $(constructstring(children(e)) \notin L(regexp_{\tau'}))$
4   **return** false
5   **if** $(\exists \sigma$ in $constructstring(children(e)), types_{\tau}(\sigma) \oslash types_{\tau'}(\sigma))$
6   **return** false
7   **foreach** child e' of e, in order,
8   **if** $(\neg revalidate(types_{\tau}(\lambda(e')), types_{\tau'}(\lambda(e')), e'))$
9   **return** false
10   **return** true
11 **boolean** $doRevalidate(S : \text{schema}, S' : \text{schema}, T : \text{tree})$
12   **if** (S is subsumed by S') **return** true
13   **if** (S is disjoint from S') **return** false
14   **return** $revalidate(\mathcal{R}(\lambda(T)), \mathcal{R}'(\lambda(T)), root(T))$

Fig. 3. Pseudocode for optimal schema revalidation.

Since we can efficiently construct IA as defined in Definition 6, our algorithm is optimal in the sense of Proposition 5.

6.2.2 Abstract XML Schema Revalidation

The algorithm provided in Fig. 2 is nearly optimal, but not optimal. Consider a call to $revalidate$ where a node n known to be valid according to some type τ is validated according to a type τ'. After the content of n is checked with respect to $regexp_{\tau'}$, $revalidate$ is invoked on each of the children of n. If the label of one of the children of n (say, the last child) is σ, and $types_{\tau}(\sigma)$ and $types_{\tau'}(\sigma)$ are disjoint, the validation of any of the children of n is unnecessary—validation will fail when the last child of n is processed. We, therefore, have to modify $revalidate$ so that this situation is taken into account.

Fig. 3 depicts the new $revalidate$ algorithm; the algorithm checks whether it can reject a tree because the types assigned to a child of a node (according to S and S', respectively) are disjoint (Line 5). This test can be merged with the content model check of Line 3. Let $\Sigma_d \subseteq \Sigma_{\tau'}$ be the set of labels used in $regexp_{\tau'}$ such that for each $\sigma \in \Sigma_d$, $types_{\tau}(\sigma)$ and $types_{\tau'}(\sigma)$ are disjoint. In validating a subtree rooted at n known to be valid according to τ with respect to τ', we can reject all trees where a child of n has a label from Σ_d. Let M_1 and M_2 be DFAs corresponding to $regexp_{\tau}$ and $regexp_{\tau'}$. We modify M_2 so that all transitions on symbols from Σ_d lead to a dead state. We then use M_1 and the modified M_2 automaton to construct the immediate decision automaton for checking the content model.

The algorithm of Fig. 3 is optimal in that there can be no other algorithm, which preprocesses only the XML Schemas, that validates a tree by marking (definition follows) fewer nodes than our algorithm, assuming that the document is not preprocessed. We formalize deterministic validation algorithms with the definition of an *abstract validator*.

Definition 8. *An* abstract validator *V with respect to an Abstract XML Schema S is a deterministic algorithm that traverses a tree T in a depth-first manner. The traversal classifies each node as either* marked *or* unmarked, *where a node is* marked *if:*

- *V verifies the content of the node according to some type τ, or*
- *V marks a child of the node in the tree.*

If V does not mark a node, then the node is unmarked. *At the end of its possibly partial traversal of a tree T, the abstract validator either accepts or rejects a tree T as belonging to $valid(S)$.*

The algorithm in Fig. 3 can be viewed as an abstract validator. If $revalidate$ is never invoked (due to one of the conditions on Line 12 or Line 13 being true), then no nodes are marked. If a call to $revalidate$ on a node n returns by the conditional on Line 2, n and all nodes in its subtree are unmarked. If the content of a node is checked (Line 3), then the node is marked. If any of the children of a node is marked, then the node is marked as well.

Definition 9. *A node n* precedes *a node n' in a tree $T = (t, \lambda)$, if n is visited before n' in the preorder traversal of t.*

Definition 10. *Let $T = ((N, E), \lambda)$ be a tree and $n \in N$ be a node. Let $T' = ((N', E'), \lambda')$ be a tree derived from T such that N' contains n and all nodes in N that precede n, E' contains all edges $(u, v) \in E$, where $u, v \in N'$, and $\lambda'(m) = \lambda(m)$, $m \in N'$. T' is said to be a* prefix tree of T with respect to n *and is denoted $P_T(n)$.*

Definition 11. *A tree $T' = ((N', E'), \lambda')$ is an* extension *of a tree $T = ((N, E), \lambda)$ with respect to a node $n \in N$ if there exists $n' \in N'$, $P_{T'}(n') \equiv P_T(n)$.*

In a preorder traversal of trees T and an extension T', the structure of T and T' are identical until n and n' are reached in T and T', respectively.

Proposition 6. *Let T be a tree known to be valid according to a schema S. Let n be a node in T marked by the algorithm of Fig. 3 during the validation of T with respect to a schema S'. There exists an extension of T with respect to n, T', such that $T' \in valid(S) \cap valid(S')$.*

Proof. We prove this by induction on the depth of n.

$depth = 0$. n is the root of the tree. If $revalidate$ was invoked on the root of the tree, the schemas S and S' are not disjoint. Therefore, there must be at least one tree T' in $valid(\mathcal{R}(\lambda(n))) \cap valid(\mathcal{R}'(\lambda(n)))$. By Proposition 1, there exists such a tree T', where the root of T' has label $\lambda(n)$. Clearly, T' is an extension of T with respect to n and is in $valid(S) \cap valid(S')$.

$depth = d > 0$. Assume that we can construct such a T' for all marked nodes of depth up to $d - 1$. Since n is marked, n's parent n' must be marked as well. Let T'' be an extension tree of T with respect to n' that is in $valid(S) \cap valid(S')$. The induction hypothesis guarantees the existence of T''. We construct a T' from T'' that is in $valid(S) \cap valid(S')$ such that T' is an extension of T with respect to n.

Let $t_0, t_1, \ldots, t_k$ be the children of n' in T, where n is the root of t_i, $0 \leq i \leq k$. By the definition of extension, there is a node n'' in T'' such that $P_{T'}(n'') \equiv P_T(n')$. We construct T' by removing all children of n'' in T'' and replacing them with $t_0, t_1, \ldots, t_{i-1}, t'_i, \ldots t'_k$ as follows: Since $revalidate$ was invoked on n by our algorithm (n is marked), the content of n' is valid according to the type in S' with respect to which it was validated. Furthermore, the invocation of $revalidate$ on all siblings of n that precede n must have returned true. For each sibling t_j, $0 \leq j < i$ of n that precedes n, in order, we add t_j as a child of n''. The t'_i tree is computed as follows: The type τ assigned to n during validation of T according to S and the type τ' with respect to which n is validated in S' are not disjoint. Otherwise, the execution of Line 5 on the parent of n (n') would have returned false. Therefore, by Proposition 1, there is at least one tree t'_i whose root has label $\lambda(n)$ such that $t'_i \in valid(\tau) \cap valid(\tau')$. The remaining $t'_{i+1} \ldots t'_k$ are constructed similarly.

T' is an extension of T with respect to n because n and the nodes that precede it in the tree have the same structure as in T. It is straightforward to show that because the tree under n'' is valid according to the type assigned to n'', T' is in $valid(\tau) \cap valid(\tau')$ (details omitted). □

Theorem 4. *Let D be an abstract validator according to an Abstract XML Schema S' that given a tree known to be in $valid(S)$ decides whether it belongs to $valid(S')$. For a tree $T \in valid(S)$, let $\mathcal{M}(T)$ be the set of nodes in T that are marked by D, and let $\mathcal{M}'(T)$ be the set of nodes in T marked by the schema revalidation algorithm of Fig. 3. For all $T \in valid(S)$, $\mathcal{M}'(T) \subseteq \mathcal{M}(T)$.*

Proof. Suppose for some tree T in $valid(S)$, there is a node n that is in $\mathcal{M}'(T)$ but not in $\mathcal{M}(T)$. We consider the two cases:

1. *n is the root of the tree T.* If D can decide whether T belongs to $valid(S')$ without marking any node in T, then either S is disjoint from S' or S' subsumes S. In these cases, our schema revalidation algorithm would also accept or reject without examining any nodes in T.
2. *n is a nonroot node in T.* n is marked by the schema revalidation algorithm. By Proposition 6, there exists some tree T' that is an extension of T with respect to n that is in $valid(S) \cap valid(S')$. Consider the processing of T' by D. By the definition of an abstract validator, D must accept T'. Since D is deterministic and the prefix tree of T' with respect to n is the same as that of T, D will not mark n in the validation of T'; D would not check the content model of n (or of any nodes in its subtree). Since the schema revalidation algorithm marks n, the types assigned to n, τ when validating according to S and τ' when validating according to S', must satisfy the relationship $valid(\tau) \not\subseteq valid(\tau')$ (otherwise, $revalidate$ would have returned on Line 2 before processing n). Therefore, there must be at least one tree, $t \in valid(\tau)$, whose root has label $\lambda(n)$, that is not in $valid(\tau')$. Let T'' be a tree that is exactly the same as T' except n (and the subtree under n if it exists) is replaced by t. Since D does not examine n (and the subtree under it), it will accept T''. T'' is not in $valid(S')$, however, because t is not in $valid(\tau')$. We, therefore, have a contradiction—D does not correctly recognize trees that belong to $valid(S')$. □

6.3 Out-of-Order Revalidation

Our algorithm for revalidation with respect to strings scans the string in the left-to-right order that is natural in string processing. In certain cases, however, an alternate order of scanning might be more efficient. For example, it is plausible that for some pair of DFAs, M_1 and M_2, examining the last character of a string known to be valid according to M_1 is sufficient to determine whether the string belongs to $L(M_2)$. The optimal order in which to revalidate strings is an open question. One can devise good heuristics for computing a good scanning order, but for reasons of space, we focus on the question of how a string may be revalidated in an out-of-order fashion under the assumption that the scanning order is given as an input.

Consider the following problem: For a string $s = s_1 \cdot \ldots \cdot s_n$ in $L(M_1)$, let Π be some permutation of $1 \ldots n$. Assume that we are given the length of s and that the characters of s are scanned in the order specified by Π. The *out-of-order* revalidation problem is to decide, while scanning as few symbols as possible, whether $s \in L(M_2)$. We provide an algorithm for this problem that is optimal in the sense that no DTM can decide whether s belongs to $L(M_2)$ by scanning fewer symbols than our algorithm. Consider the immediate decision automaton, $M_{immed} = (Q, \Sigma, \delta, q^0, F, IA, IR)$ derived from M_1 and M_2. Our algorithm relies on a Boolean function, $Skip(p, q, h)$ that answers *true* if and only if there is some string w of length h such that when processed by M_{immed}, $\delta(p, w) = q$. $Skip(q, q, 0)$ is always true.

The computation of $Skip$ is based on the *ultimate periodic* property of regular sets over a one-letter alphabet [18]. A set X of natural numbers is ultimately periodic if it is either finite or there are natural numbers N and P such that for all $x \geq N$, $x \in X$ if and only if $x + P \in X$. A set $L \subseteq \{a\}^*$ is regular if and only if $X = \{i | a^i \in L\}$ is ultimately periodic [18]. Clearly, the set $S_{p,q} = \{a^i | \exists w \in \Sigma^*, |w| = i \wedge \delta(p, w) \rightarrow q\}$ is regular (by making all transitions in M_{immed} to be on a single symbol a). Since it is regular, we can construct a minimal DFA M' recognizing it. From M' it is straightforward to obtain $N_{p,q}$ and $P_{p,q}$ that define the ultimately periodic set $U_{p,q} = \{i | a^i \in S_{p,q}\}$. Given $N_{p,q}$ and $P_{p,q}$, we can determine whether $u \in U_{p,q}$ for any $u > N_{p,q} + P_{p,q}$. This is done by first classifying membership in $U_{p,q}$ for each $j = N_{p,q}, N_{p,q} + 1, \ldots, N_{p,q} + P_{p,q}$, and then, verifying that u is in the arithmetic progression with parameter $P_{p,q}$ starting at one of these j members.

Given that the symbols are scanned according to the permutation Π, we record for each i, $0 \leq i \leq n$, $|s| = n$, and $Exit(i)$, which is a conservative approximation of the set of states M_{immed} might be in after processing s_i, $1 \leq i \leq n$. Initially, $Exit(0) = q^0$, $Exit(n) = Q$, and $Exit(i)$ are undefined for $1 \leq i < n$. When a new symbol s_i, $1 \leq i \leq n$ is scanned, we perform the following steps:

1. Let j be the rightmost symbol such that s_j has been scanned, where $1 \leq j < i$. If no such symbol exists let $j = 0$. Let
$$Q' = \{q' | Skip(q, q', i - j - 1) = true, q \in Exit(j)\}.$$
In other words, Q' is the set of states that M_{immed} might be in starting in a state in $Exit(j)$ and processing a string of length $i - j - 1$, that is, those states that are potentially reachable through some string that "fills in blanks," of the unexplored portions of the string.
2. Set $Exit(i) = \{q' \mid \delta(q, s_i) = q', q \in Q'\}$.
3. If $i = n$ and $Exit(i)$ contains only states in F, then accept s as being in $L(M_2)$. If $i = n$ and $Exit(i)$ does not contain any states in F, then reject s. If $Exit(i) \subseteq IA$, that is, all states in $Exit(i)$ are immediate acceptance states, accept since $s \in L(M_2)$. If $Exit(i) \subseteq IR$, that is, all states in $Exit(i)$ are immediate rejection states, reject s. If s has been accepted or rejected, halt the processing of the string.
4. If $i \neq n$, let k be the least position greater than i such that s_k has been scanned and none of the positions between s_i and s_k have been scanned. If no such k exists, let $k = n$. Repeat Steps 1-4 with $i = k$.

We continue the process described above until the string s is accepted or rejected (it is straightforward to show that one or the other will occur eventually). Similarly to the case of left-to-right scanning, we assert that if any DTM D reaches any decision by scanning portions of s, so can our algorithm while scanning exactly the same portion of s. The proof technique is similar to that of the left-to-right scan and is omitted.

7 Experiments

We demonstrate the performance benefits of our schema revalidation algorithm by comparing its performance to that of Xerces [19]. We have modified Xerces 2.4 to perform schema revalidation as described in Section 4.1. We first consider the amount of time necessary to calculate the subsumption relation between the schemas (the disjoint relation can be constructed in parallel with the subsumption relation).

Rather than implementing the algorithm of Section 4.1 directly, our algorithm computes the subtyping relation in a goal-directed manner. We begin by trying to verify the subsumption relation among the top-level elements in the two schemas. Given a goal of proving that τ is subsumed by τ', we check whether the content model of τ is a subset of the content model of τ'. If it is, then a new set of goals is introduced, where for each σ used in the content model of τ, we attempt to find if $types_\tau(\sigma)$ is subsumed by $types_{\tau'}(\sigma)$. To avoid redundant computation, the algorithm caches the results of resolution of goals. If in the process of proving $\tau \preceq \tau'$, one arrives at a stage where $\tau \preceq \tau'$ must be introduced as a subgoal, one can discard that subgoal. A proof of correctness for this procedure is omitted for space, but it is similar to that of other coinductive algorithms [20].

We ran the schema analysis algorithm on several schemas, including those in Fig. 1a. Observe that the worst-case complexity for the algorithm occurs when the two schemas are identical—the maximum number of elements must be compared to each other. For a schema based on Fig. 1a (2K bytes), the algorithm for computing the subsumption relationship took 5 milliseconds (ms); this cost can be amortized across several revalidations over documents satisfying the schema. The time taken to compute the subsumption relations between the schemas of Fig. 1a and Fig. 1b, as well as a schema based on the XMark data set [21] (14K bytes), is similar.

For revalidation, our modified Xerces validator receives a DOM [22] representation of an XML document that conforms to a schema S_1. At each stage of the validation process, while validating a subtree of the DOM tree with

```
<xsd:schema xmlns:xsd=". . . ">
<xsd:element name="purchaseOrder" type="POType2"/>
<xsd:element name="comment" type="xsd:string"/>

<xsd:complexType name="POType2">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
    <xsd:element name="items" type="Items"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="USAddress">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
    <xsd:element name="country" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="item" type="Item"
      minOccurs="0" maxOccurs="unbounded">
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="Item">
  <xsd:sequence>
    <xsd:element name="productName"
      type="xsd:string"/>
    <xsd:element name="quantity">
      <xsd:simpleType>
      <xsd:restriction base="xsd:positiveInteger">
      <xsd:maxExclusive value="100"/>
      </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
    <xsd:element name="USPrice"
      type="xsd:decimal"/>
    <xsd:element name="shipDate"
      type="xsd:date" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Fig. 4. Target XML Schema.

respect to a schema S_2, the validator consults hash tables to determine if it may skip validation of that subtree. There is a hash table that stores pairs of types that are in the subsumed relationship, and another that stores the disjoint types. The unmodified Xerces validates the entire document.

Due to the complexity of modifying the Xerces code base and to perform a fair comparison with Xerces, we do not use the algorithms mentioned in Section 5 to optimize the checking of whether the labels of the children of a node fit the appropriate content model. In both the modified Xerces and the original Xerces implementation, the content of a node is checked by executing a finite state automaton on the labels of the node's children.

We provide results for two experiments. In the first experiment, a document known to be valid with respect to the schema of Fig. 1a is validated with respect to the schema of Fig. 1b. The complete schema of Fig. 1b is provided in Fig. 4. In the second experiment, we modify the quantity element declaration (in items) in the schema of Fig. 4 to set xsd:maxExclusive to "200" (instead of "100"). Given a document conforming to this modified schema, we check whether it belongs to the schema of Fig. 4. In the first experiment, with our algorithm, the time complexity of validation does not depend on the size of the input document—the document is valid if it contains a billTo element. In the second experiment, the quantity element in *every* item element must be checked to ensure that it is less than "100." Therefore, our algorithm scales linearly with the number of item elements in the document. All experiments were executed on a 3.0Ghz IBM Intellistation running Linux 2.4, with 512MB of memory.

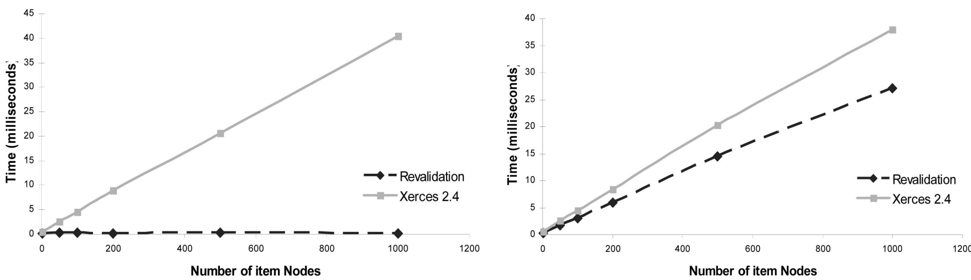
We provide results for input documents that conform to the schema of Fig. 4. We vary the number of item elements from 2 to 1,000. Table 2 lists the file size of each document. Fig. 5a plots the time taken to validate the document versus the number of item elements in the document for both the modified and the unmodified Xerces validators for the first experiment. As expected, our implementation has constant processing time, irrespective of the size of the document, whereas Xerces has a linear cost curve. Fig. 5b shows the results of the second experiment. The schema revalidation algorithm is about 30 percent faster than the unmodified Xerces algorithm. Table 2 lists the number of nodes visited by both algorithms. By only traversing the quantity child of item and not the other children of item, our algorithm visits about 20 percent fewer nodes than the unmodified Xerces validator. For larger files, especially when the data are out-of-core, the performance benefits of our algorithms would be even more significant.

8 Conclusions

We have presented efficient solutions to the problem of enforcing the validity of a document with respect to a schema given the knowledge that it conforms to another schema. We examine both the case where the document is not modified before revalidation, and the case where modifications are applied to the document before revalidation. We have provided an algorithm for the case where validation is defined in terms of abstract XML Schemas. The algorithm relies on a subalgorithm that addresses the problem of revalidation with respect to deterministic finite state automata to revalidate content models efficiently. We have considered optimizations that take advantage of auxiliary state stored with elements. The solution to this schema revalidation problem is useful in many contexts ranging from the compilation of query and programming

TABLE 2
Number of Nodes Traversed during Validation in Experiment 2

# Item Nodes	Size (Bytes)	Revalidation	Xerces 2.4
2	990	35	74
50	11,358	611	794
100	22,158	1,211	1,544
200	43,758	2,411	3,044
500	108,558	6,011	7,544
1000	216,558	12,011	15,044



(a) (b)

Fig. 5. (a) Validation times from first experiment. (b) Validation times from second experiment.

languages with XML types, to handling XML messages and Web Services interactions.

The efficiency of our algorithms has been demonstrated through experiments. Unlike schemes that preprocess documents (that handle a subset of our schema revalidation problem), the memory requirement of our algorithm does not vary with the size of the document, but depends solely on the sizes of the schemas. We are exploring how to correct a document valid according to one schema so that it conforms to a new schema.

ACKNOWLEDGMENTS

Oded Shmueli was supported by the P. and E. Nathan Research Fund and in part by ISF grant 890015.

REFERENCES

[1] "XML Schema, Parts 0, 1, and 2," *W3C Recommendation,* World Wide Web Consortium, May 2001.
[2] *Extensible Markup Language (XML) 1.0 (Second Edition).* World Wide Web Consortium, Oct. 2000.
[3] "XQuery 1.0: An XML Query Language," W3C Working Draft, World Wide Web Consortium, Nov. 2000.
[4] A. Balmin, Y. Papakonstantinou, and V. Vianu, "Incremental Validation of XML Documents," *ACM Trans. Database Systems (TODS),* vol. 29, no. 4, pp. 710-751, 2004.
[5] M.Y. Levin and B.C. Pierce, "Type-Based Optimization for Regular Patterns," *Proc. 10th Int'l Symp. Database Programming Languages,* pp. 184-198, 2005.
[6] A. Frisch, "Regular Tree Language Recognition with Static Information," *Proc. Third IFIP Int'l Conf. Theoretical Computer Science,* 2004.
[7] Y. Papakonstantinou and V. Vianu, "Incremental Validation of XML Documents," *Proc Int'l Conf. Database Theory,* pp. 47-63, Jan. 2003.
[8] D. Barbosa, A. Mendelzon, L. Libkin, L. Mignet, and M. Arenas, "Efficent Incremental Validation of XML Documents," *Proc. 20th Int'l Conf. Data Eng. (ICDE),* pp. 671-682, Mar. 2004.
[9] B. Kane, H. Su, and E.A. Rundensteiner, "Consistently Updating XML Documents Using Incremental Constraint Check Queries," *Proc. Workshop Web Information and Data Management (WIDM '02),* pp. 1-8, Nov. 2002.
[10] B. Bouchou and M.H.F. Alves, "Updates and Incremental Validation of XML Documents," *Proc. Eighth Int'l Symp. Database Programming Languages,* pp. 216-232, 2003.
[11] F. Neven, "Automata Theory for XML Researchers," *SIGMOD Record,* vol. 31, no. 3, 2002.
[12] G. Kuper and J. Siméon, "Subsumption for XML Types," *Proc. Eighth Int'l Conf. Database Theory (ICDT),* vol. 1973, pp. 331-345, Jan. 2001.
[13] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.
[14] M. Raghavachari and O. Shmueli, "Efficient Schema-Based Revalidation of XML," *Proc. Extending Database Technology (EDBT),* Mar. 2004.
[15] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* The MIT Press, 1989.
[16] A. Bruggemann-Klein and D. Wood, "One-Unambiguous Regular Languages," *Information and Computation,* vol. 142, no. 2, pp. 182-206, May 1998.
[17] W. Martens, F. Neven, and T. Schwentick, "Complexity of Decision Problems for Simple Regular Expressions," *Proc. 29th Symp. Math. Foundations of Computer Science,* pp. 889-900, 2004.
[18] M. Harrison, *Introduction to Formal Language Theory.* Addison-Wesley, 1978.
[19] Xerces2 Java Parser, Apache Software Foundation, http://xml.apache.org/, 2005.
[20] D. Kozen, J. Palsberg, and M. Scwartzbach, "Efficient Recursive Subtyping," *Math. Structures in Computer Science,* vol. 5, no. 1, pp. 113-125, 1995.
[21] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse, "Xmark: A Benchmark for XML Data Management," *Proc. 28th Int'l Conf. Very Large Databases (VLDB),* pp. 974-985, 2002.
[22] "Document Object Model Level 2 Core," *W3C Recommendation,* World Wide Web Consortium, Nov. 2000.
[23] M. Murata, D. Lee, and M. Mani, "Taxonomy of XML Schema Languages Using Formal Language Theory," *Proc. 2001 Extreme Markup Languages Conf.,* 2001.

Mukund Raghavachari graduated from the University of Pennsylvania (1991) and received the PhD degree in computer science from Princeton University (1998). He is a research staff member at the IBM T.J. Watson Research Center, Hawthorne, New York. He currently manages the Data Access Language Integration (DALI) group at IBM Research. His research interests include theoretical and practical aspects of programming language support for accessing data, including XML and relational database access.

Oded Shmueli graduated from Brandeis University (1977) and received the PhD degree in applied mathematics from Harvard University (1981). He is a professor of computer science at the Technion, Haifa, Israel. He has held positions at several leading industrial research laboratories including IBM Research, AT&T Bell Labs, and MCC. He was also a consultant for HP Laboratories over a five-year period. He was a cofounder and CTO of Dealigence Inc., a start-up company in the electronic commerce domain. His research interests include theoretical and practical aspects of database systems, XML technology, and electronic commerce, topics on which he has published widely. He also holds four US patents.