Efficient Schema-Based Revalidation of XML

Mukund Raghavachari[1] and Oded Shmueli[2]

[1] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA,
raghavac@us.ibm.com,
[2] Technion – Israel Institute of Technology, Haifa, Israel,
oshmu@cs.technion.ac.il

Abstract. As XML schemas evolve over time or as applications are integrated, it is sometimes necessary to validate an XML document known to conform to one schema with respect to another schema. More generally, XML documents known to conform to a schema may be modified, and then, require validation with respect to another schema. Recently, solutions have been proposed for incremental validation of XML documents. These solutions assume that the initial schema to which a document conforms and the final schema with which it must be validated after modifications are the same. Moreover, they assume that the input document may be preprocessed, which in certain situations, may be computationally and memory intensive. In this paper, we describe how knowledge of conformance to an XML Schema (or DTD) may be used to determine conformance to another XML Schema (or DTD) efficiently. We examine both the situation where an XML document is modified before it is to be revalidated and the situation where it is unmodified.

1 Introduction

The ability to validate XML documents with respect to an XML Schema [21] or DTD is central to XML's emergence as a key technology for application integration. As XML data flow between applications, the conformance of the data to either a DTD or an XML schema provides applications with a guarantee that a common vocabulary is used and that structural and integrity constraints are met. In manipulating XML data, it is sometimes necessary to validate data with respect to more than one schema. For example, as a schema evolves over time, XML data known to conform to older versions of the schema may need to be verified with respect to the new schema. An intra-company schema used by a business might differ slightly from a standard, external schema and XML data valid with respect to one may need to be checked for conformance to the other.

The validation of an XML document that conforms to one schema with respect to another schema is analogous to the cast operator in programming languages. It is useful, at times, to access data of one type as if it were associated with a different type. For example, XQuery [20] supports a validate operator which converts a value of one type into an instance of another type. The type safety of this conversion cannot always be guaranteed statically. At runtime,

XML fragments known to correspond to one type must be verified with respect to another. As another example, in XJ [9], a language that integrates XML into Java, XML variables of a type may be updated and then cast to another type. A compiler for such a language does not have access to the documents that are to be revalidated. Techniques for revalidation that rely on preprocessing the document [3, 17] are not appropriate.

The question we ask is how can one use knowledge of conformance of a document to one schema to determine whether the document is valid according to another schema? We refer to this problem as the *schema cast validation* problem. An obvious solution is to revalidate the document with respect to the new schema, but in doing so, one is disregarding useful information. The knowledge of a document's conformance to a schema can help determine its conformance to another schema more efficiently than full validation. The more general situation, which we refer to as *schema cast with modifications validation*, is where a document conforming to a schema is modified slightly, and then, verified with respect to a new schema. When the new schema is the same as the one to which the document conformed originally, schema cast with modifications validation addresses the same problem as the incremental validation problem for XML [3, 17]. Our solution to this problem has different characteristics, as will be described.

The scenario we consider is that a *source* schema A and a *target* schema B are provided and may be preprocessed statically. At runtime, documents valid according to schema A are verified with respect to schema B. In the modification case, inserts, updates, and deletes are performed to a document before it is verified with respect to B. Our approach takes advantage of similarities (and differences) between the schemas A and B to avoid validating portions of a document if possible. Consider the two XML Schema element declarations for purchaseOrder shown in Figure 1. The only difference between the two is that whereas the billTo element is optional in the schema of Figure 1a, it is required in the schema of Figure 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 would be valid. Given a document valid according to the schema of Figure 1a, an ideal validator would only check the presence of a billTo element and ignore the validation of the other components (they are guaranteed to be valid).

This paper focuses on the validation of XML documents with respect to the structural constraints of XML schemas. We present algorithms for schema cast validation with and without modifications that avoid traversing subtrees of an XML document where possible. We also provide an optimal algorithm for revalidating strings known to conform to a deterministic finite state automaton according to another deterministic finite state automaton; this algorithm is used to revalidate content model of elements. The fact that the content models of XML Schema types are deterministic [6] can be used to show that our algorithm for XML Schema cast validation is optimal as well. We describe our algorithms in terms of an abstraction of XML Schemas, *abstract XML schemas*, which model the structural constraints of XML schema. In our experiments, our algorithms achieve 30-95% performance improvement over Xerces 2.4.

```
<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 (b) Target Schema.

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 more precisely than specialized DTDs [16] and regular type expressions [11].
2. Efficient algorithms for schema cast validation (with and without updates) of XML documents with respect to XML Schemas. We describe optimizations for the case where the schemas are DTDs. Unlike previous algorithms, our algorithms do not preprocess the documents that are to be revalidated.
3. Efficient algorithms for revalidation of strings with and without modifications according to deterministic finite state automata. These algorithms are essential for efficient revalidation of the content models of elements.
4. Experiments validating the utility of our solutions.

Structure of the Paper: We examine related work in Section 2. In Section 3, we introduce abstract XML Schemas and provide an algorithm for XML schema revalidation. The algorithm relies on an efficient solution to the problem of string revalidation according to finite state automata, which is provided in Section 4. We discuss the optimality of our algorithms in Section 5. We report on experiments in Section 6, and conclude in Section 7.

2 Related Work

Papakonstantinou and Vianu [17] treat incremental validation of XML documents (typed according to specialized DTDs). Their algorithm keeps data structures that encode validation computations with document tree nodes and utilizes these structures to revalidate a document. Barbosa *et al.* [3] present an algorithm that also encodes validation computations within tree nodes. They take advantage of the 1-unambiguity of content models of DTDs and XML Schemas [6], and structural properties of a restricted set of DTDs, to revalidate documents.

Our algorithm is designed for the case where schemas can be preprocessed, but the documents to be revalidated are not available *a priori* to be preprocessed. Examples include message brokers, programming language and query compilers, etc. In these situations, techniques that preprocess the document and store state information at each node could incur unacceptable memory and computing overhead, especially if the number of updates is small with respect to the document, or the size of the document is large. Moreover, our algorithm handles the case where the document must be revalidated with respect to a different schema.

Kane *et al.* [12] use a technique based on query modification for handling the incremental update problem. Bouchou and Halfeld-Ferrari [5] 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.

The subsumption of XML schema types used in our algorithm for schema cast validation is similar to Kuper and Siméon's notion of type subsumption [13]. Their type system is more general than our abstract XML schema. They assume that a subsumption mapping is provided between types such that if one schema is subsumed by another, and if a value conforming to the subsumed schema is 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.

One approach to handling XML and XML Schema has been to express them in terms of formal models such as tree automata. For example, Lee *et al.* describe how XML Schema may be represented in terms of deterministic tree grammars with one lookahead [15]. The formalism for XML Schema and the algorithms in these paper are a more direct solution to the problem, which obviates some of the practical problems of the tree automata approach, such as having to encode unranked XML trees as ranked trees.

Programming languages with XML types [1, 4, 10, 11] define notions of types and subtyping that are enforced statically. XDuce [10] uses tree automata as the base model for representing XML values. One difference between our work and XDuce is that we are interested in dynamic typing (revalidation) where static analysis is used to *reduce* the amount of needed work. Moreover, unlike XDuce's regular expression types and specialized DTDs [17], our model for XML values captures exactly the structural constraints of XML Schema (and is not equivalent to regular tree automata). As a result, our subtyping algorithm is polynomial rather than exponential in complexity.

3 XML Schema and DTD Conformance

In this section, we present the algorithm for revalidation of documents according to XML Schemas. We first define our abstractions for XML documents,

ordered labeled trees, and for XML Schema, *abstract XML Schema*. Abstract XML Schema captures precisely the structural constraints of XML Schema.

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 the tree t, represents XML Schema simple values. We use $root(T)$ to denote the root of tree t. We shall abuse notation slightly to allow $\lambda(T)$ to denote the label of the root node of the ordered labeled tree T. We use $r(t_1, t_2, \ldots, t_k)$ to denote an ordered tree with root r and subtrees $t_1 \ldots t_k$, where $r()$ denotes an ordered tree with a root r that has no children. We use $\mathfrak{T}_\Sigma$ to represent the set of all ordered labeled trees.

Abstract XML Schema XML Schemas, unlike DTDs, permit the decoupling of an element tag from its type; an element may have different types depending on context. XML Schemas are not as powerful as regular tree automata. The XML Schema specification places restrictions on the decoupling of element tags and types. Specifically, in validating a document according to an XML Schema, each element of the document can be assigned a single type, based on the element's label and the type of the element's parent (without considering the content of the element). Furthermore, this type assignment is guaranteed to be unique.

We define an abstraction of XML Schema, an *abstract XML Schema*, as 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 τ 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 over Σ. $L(regexp_\tau)$ denotes the language associated with $regexp_\tau$.
  - Let $\Sigma_\tau \subseteq \Sigma$ be the set of element labels used in $regexp_\tau$. 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, and the type this root element is assigned.

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

Table 1. Abstract XML Schema type for XML Schema type POType1 of Figure 1a.

Type	Σ_τ	$regexp_\tau$	$types_\tau$
POType1	shipTo billTo items	(shipTo billTo? items)	shipTo → USAddress billTo → USAddress items → Items

Abstract XML Schemas do not explicitly represent atomic 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 in this paper. This 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.

We define the validity of an ordered, labeled tree with respect to an abstract XML Schema as follows:

Definition 1. *The set of ordered labeled trees that are valid with respect to a type is defined in terms of the least solution to a set of equations, one for each* $\tau \in \rho$*, of the form (*n, n_1, n_2 *are nodes):*

$$valid(\tau) = \begin{cases} \{(t,\lambda) \in \mathfrak{T}_\Sigma | t = n_1(n_2()), \lambda(n_1) \in \Sigma, \lambda(n_2) = \chi\} & \text{if } \tau \text{ is simple} \\ \\ \{(t,\lambda) \in \mathfrak{T}_\Sigma | t = n(), \lambda(n) \in \Sigma, \epsilon \in L(regexp_\tau)\} \cup & \text{otherwise} \\ \{(t,\lambda) \in \mathfrak{T}_\Sigma | t = n(t_1, t_2, \ldots, t_k) & \\ \quad \lambda(n), \lambda(t_1), \ldots, \lambda(t_k) \in \Sigma, k > 0 & \\ \quad \lambda(t_1) \cdot \lambda(t_2) \cdot \ldots \cdot \lambda(t_k) \in L(regexp_\tau) & \\ \quad t_i \in valid(types_\tau(\lambda(t_i))), 1 \leq i \leq k\} & \end{cases}$$

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 $T \in valid(\mathcal{R}(\lambda(T)))$. 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.*

We are interested only in *productive* types, that is types, τ, where $valid(\tau) \neq \emptyset$. We assume that for a schema $S = (\Sigma, \mathcal{T}, \rho, \mathcal{R})$, all $\tau \in \mathcal{T}$ are productive. Whether a type is productive can be verified easily as follows:

1. Mark all simple types as productive since by the definition of *valid*, they contain trees of height 1 with labels from Σ.
2. For complex types, τ, compute the set $ProdLabels_\tau \subseteq \Sigma$ defined as $\{\sigma \in \Sigma \mid types_\tau(\sigma)$ is productive$\}$.

3. Mark τ as productive if $ProdLabels_\tau^* \cap L(regexp_\tau) \neq \emptyset$. In other words, a type τ is productive if $\epsilon \in L(regexp_\tau)$ or there is a string in $L(regexp_\tau)$ that uses only labels from $ProdLabels_\tau$.
4. Repeat Steps 2 and 3 until no more types can be marked as productive.

This procedure identifies all productive types defined in a schema. There is a straightforward algorithm for converting a schema with types that are non-productive into one that contains only productive types. The basic idea is to modify $regexp_\tau$ for each productive τ so that the language of the new regular expression is $L(regexp_\tau) \cap ProdLabels_\tau^*$.

Pseudocode for validating an ordered, labeled tree with respect to an abstract XML Schema is provided below. *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.

boolean $validate(\tau : \text{type}, e : \text{node})$
  if (τ is a simple type)
    if ($children(e) = \{n()\}$, $\lambda(n) = \chi$) **return** true
    else return false
  if ($\neg constructstring(children(e)) \in L(regexp_\tau)$)
      return false
  foreach child e' of e
    if ($\neg validate(types_\tau(\lambda(e')), e')$)
      return false
  return true

boolean $doValidate(S : \text{schema}, T : \text{tree})$
  return $validate(\mathcal{R}(\lambda(T)), root(T))$

A *DTD* can be viewed as an abstract XML Schema, $D = (\Sigma, \mathcal{T}, \rho, \mathcal{R})$, where each $\sigma \in \Sigma$ is assigned a unique type irrespective of the context in which it is used. In other words, for all $\sigma \in \Sigma$, there exists $\tau' \in \mathcal{T}$ such that for all $\tau : (regexp_\tau, types_\tau) \in \rho$, $types_\tau(\sigma)$ is either not defined or $types_\tau(\sigma) = \tau'$. If $\mathcal{R}(\sigma)$ is defined, then $\mathcal{R}(\sigma) = \tau'$ as well.

3.1 Algorithm Overview

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' we wish to 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 where possible:

Definition 2. *A type τ is subsumed by a type τ', denoted $\tau \preceq \tau'$ if $valid(\tau) \subseteq valid(\tau')$. Note that τ and τ' can belong to different schemas.*

Definition 3. *Two types τ and τ' are disjoint, denoted $\tau \oslash \tau'$, if $valid(\tau) \cap valid(\tau') = \emptyset$. Again, note that τ and τ' can 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. We present an algorithm for efficient schema cast validation of an ordered labeled tree, with and without updates. Finally, in the case where the abstract XML Schemas represent DTDs, we describe optimizations that are possible if additional indexing information is available on ordered labeled trees.

3.2 Schema Cast Validation

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 $\mathcal{T}'$. We first describe how these relations are computed, and then, present our algorithm for schema cast validation.

Computing the R_{sub} relation

Definition 4. *Given two schemas, $S = (\Sigma, \mathcal{T}, \rho, \mathcal{R})$ and $S' = (\Sigma, \mathcal{T}', \rho', \mathcal{R}')$, let $R_{sub} \subseteq \mathcal{T} \times \mathcal{T}'$ be the* largest *relation such that for all $(\tau, \tau') \in R_{sub}$ one of the following two conditions hold:*

i. τ, τ' *are both simple types.*
ii. τ, τ' *are both complex types, $L(regexp_{\tau}) \subseteq L(regexp_{\tau'})$, and $\forall \sigma \in \Sigma$, where $types_{\tau}(\sigma)$ is defined, $(types_{\tau}(\sigma), types_{\tau'}(\sigma)) \in R_{sub}$.*

As mentioned before, for exposition reasons, we have chosen to merge all simple types into one common *simple type*. It is straightforward to extend the definition above so that the various XML Schema atomic and simple types, and their derivations are used to bootstrap the definition of the subsumption relationship. Also, observe that R_{sub} is a finite relation since there are finitely many types.

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

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

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 $R_{sub} \subseteq \mathcal{T} \times \mathcal{T}'$, such that $(\tau, \tau') \in R_{sub}$ implies that 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 exists $\sigma \in \Sigma, types_\tau(\sigma) = \omega$ and $types_{\tau'}(\sigma) = \nu$ 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}.

Computing the R_{dis} relation Rather than computing R_{dis} directly, we compute its complement. Formally:

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 as the smallest relation (least fixpoint) such that $(\tau, \tau') \in R_{nondis}$ if:*

i. *τ and τ' are both simple types.*
ii. *τ and τ' are both complex types, $P = \{\sigma \in \Sigma \mid (types_\tau(\sigma), types_{\tau'}(\sigma)) \in R_{nondis}\}$, $L(regexp_\tau) \cap L(regexp_{\tau'}) \cap P^* \neq \emptyset$.*

To compute the R_{nondis} relation, the algorithm 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 $(\tau, \tau') \notin R_{nondis}$.* □

Algorithm for Schema Cast Validation Given the relations R_{sub} and R_{dis}, if at any time, a subtree of the document that is valid with respect to τ from S is being validated with respect to τ' from 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 validation of the document is provided below. 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 efficiently verify the content model of e with respect to $regexp_{\tau'}$ by using techniques for finite automata schema cast validation, as will be described in the Section 4.

boolean *validate*(τ : type, τ' : type, e : node)
  if $\tau \preceq \tau'$ **return** true
  if $\tau \oslash \tau'$ **return** false
  if (τ is a simple type)
    if ($children(e) = \{n()\}, \lambda(n) = \chi$) **return** true
    else return false

```
  if (¬constructstring(children(e)) ∈ L(regexp_τ'))
      return false
  foreach child e' of e, in order,
    if (¬validate(types_τ(λ(e')), types_τ'(λ(e')), e'))
      return false
  return true

boolean doValidate(S : schema, S' : schema, T : tree)
  return validate(R(λ(T)), R'(λ(T)), root(T))
```

3.3 Schema Cast Validation 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 modifications of element tags, 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. Modify the label of a specified node with a new label.
2. Insert a new leaf node before, or after, or as the first child of a node.
3. Delete a specified leaf node.

Given a sequence of updates, we perform the updates on T, and at each step, we encode the modifications on T to obtain 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. Nodes that have not been modified have their labels 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.

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 assume we have the Dewey decimal number of the node (generated dynamically as we process). Whenever a node is updated we keep it in a trie [7] according to its Dewey decimal number. To determine whether a descendant of a node v was modified, the trie is searched according to the Dewey decimal number of v. Note that we can navigate the trie in parallel to navigating the XML tree.

The algorithm for efficient validation of schema casts 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*, we can run the algorithm described in the previous subsection on this subtree. Since the subtree t'' is unchanged and we know that $t'' \in valid(\tau)$ when checked with respect to S, we can treat the validation of t'' as an instance of the schema cast validation problem (without modifications) described in Section 3.2.

2. Otherwise, if $\lambda'(t'') = \Delta^a_\epsilon$, we do not need to validate the subtree with respect to any 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 must validate the whole subtree explicitly.
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 model 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 model must satisfy (1) of Definition 1. Otherwise, if $t'' = n(t_1, \ldots, t_k)$, one must 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 $Proj_{new}(t_1) \ldots Proj_{new}(t_k) \in L(regexp_{\tau'})$, where $Proj_{new}(t_i)$ is defined as:

$$\sigma \text{ , if } \lambda'(t_i) = \sigma, \sigma \in \Sigma \cup \{\chi\} \quad (1)$$

$$b \text{ , if } \lambda'(t_i) = \Delta^a_b, a, b \in \Sigma \cup \{\epsilon, \chi\} \quad (2)$$

   $Proj_{old}$ is defined analogously. If the content model check succeeds, and τ is also a complex type, then we continue recursively validating $t_i, 1 \leq i \leq k$ 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 that t_i since it has been deleted in T'). If τ is not a complex type, we must validate each t_i explicitly.

3.4 DTDs

Since the type of an element in an XML Schema may depend on the context in which it appears, in general, it is necessary to process the document in a top-down manner to determine the type with which one must validate an element (and its subtree). For DTDs, however, an element label determines uniquely the element's type. As a result, there are optimizations that apply to the DTD case that cannot be applied to the general XML Schema case. If one can access all instances of an element label in an ordered labeled tree directly, one need only visit those elements e where the types of e in S and S' are neither subsumed nor disjoint from each other and verify their immediate content model.

4 Finite Automata Conformance

In this section, we examine the schema cast validation problem (with and without modifications) for strings verified with respect to finite automata. The algorithms described in this section support efficient content model checking for DTDs and XML Schemas (for example, in the statement of the method *validate* of Section 3.2: **if** $(\neg constructstring(children(e)) \in L(regexp_{\tau'})))$. Since XML Schema content models correspond directly to deterministic finite state automata, we only address that case. Similar techniques can be applied to non-deterministic finite state automata, though the optimality results do not hold. For reasons of space, we omit details regarding non-deterministic finite state automata.

4.1 Definitions

A deterministic finite automaton 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 relation. δ is a map from $Q \times \Sigma$ to Q. Without loss of generality, we assume that for all $q \in Q, \sigma \in \Sigma, \delta(q, \sigma)$ is defined. We use $\delta(q, \sigma) \rightarrow q'$, where $q, q' \in Q, \sigma \in \Sigma$ to denote that δ maps (q, σ) to q'. For string s and state q, $\delta(q, s) \rightarrow q'$ denotes the state q' reached by operating on s one symbol at a time. A string s is *accepted* by a finite state automaton if $\delta(q^0, s) \rightarrow q', q' \in F$; s is *rejected* by the automaton if s is not accepted by it.

The language accepted (or recognized) by a finite automaton a, denoted $L(a)$, is the set of strings accepted by a. We also define $L_a(q), q \in Q$, as $\{s \mid \delta(q, s) \rightarrow q', q' \in F\}$. Note that for a finite state automaton a, if a string $s = s_0 \cdot s_1 \cdot \ldots \cdot s_n$ is in $L(a)$, and $\delta(q^0, s_0 \cdot s_1 \cdot \ldots \cdot s_i) = q', 1 \leq i < n$, then $s_{i+1} \cdot \ldots \cdot s_n$ is in $L_a(q')$. We shall drop the subscript a from L_a when the automaton is clear from the context.

A state $q \in Q$ is a *dead state* if either:

1. $\forall s \in \Sigma^*$, if $\delta(q^0, s) \rightarrow q'$ then $q \neq q'$, or
2. $\forall s \in \Sigma^*$, if $\delta(q, s) \rightarrow q'$ then $q' \notin F$.

In other words, either the state is not reachable from the start state or no final state is reachable from it. We can identify all dead states in a finite state automaton in time linear in the size of the automaton via a simple graph search.

Intersection Automata Given two automata, $a = (Q_a, \Sigma_a, \delta_a, q_a^0, F_a)$ and $b = (Q_b, \Sigma_b, \delta_b, q_b^0, F_b)$, one can derive an intersection automaton c, such that c accepts exactly the language $L(a) \cap L(b)$. The intersection automaton evaluates a string on both a and b in parallel and accepts only if both would. Formally, $c = (Q_c, \Sigma, \delta_c, q_c^0, F_c)$, where $q_c^0 = (q_a^0, q_b^0), Q_c = Q_a \times Q_b, F_c = F_a \times F_b$, and $\delta_c((q_1, q_2), \sigma) \rightarrow q'$, where $q' = (\delta_a(q_1, \sigma), \delta_b(q_2, \sigma))$. Note that if a and b are deterministic, c is deterministic as well.

Immediate Decision Automata We introduce *immediate decision automata* as modified finite state automata that accept or reject strings as early as possible. Immediate decision automata can accept or reject a string when certain conditions are met, without scanning the entire string. Formally, an immediate decision automaton d_{immed} is a 7-tuple, $(Q, \Sigma, \delta, q^0, F, IA, IR)$, where IA, IR are disjoint sets and $IA, IR \subseteq Q$ (each member of IA and IR is a state). As with ordinary finite state automata, a string s is accepted by the automaton if $\delta(q^0, s) \rightarrow q', q' \in F$. Furthermore, d_{immed} also accepts s after evaluating a strict prefix x of s (that is $x \neq s$) if $\delta(q^0, x) \rightarrow q', q' \in IA$. d_{immed} rejects s after evaluating a strict prefix x of s if $\delta(q^0, x) \rightarrow q', q' \in IR$. We can derive an immediate decision automaton from a finite state automaton so that both automata accept the same language.

Definition 6. *Let* $d = (Q_d, \Sigma, \delta_d, q_d^0, F_d)$ *be a finite state automaton. The derived immediate decision automaton is* $d_{immed} = (Q_d, \Sigma, \delta_d, q_d^0, F_d, IA_d, IR_d)$*, where:*

- $IA_d = \{q' \mid q' \in Q_d \wedge L_d(q') = \Sigma^*\}$*, and*
- $IR_d = \{q' \mid q' \in Q_d \wedge L_d(q') = \emptyset\}$*.*

It can be easily shown that d_{immed} and d accept the same language.

For deterministic automata, we can determine all states that belong to IA_d and IR_d efficiently in time linear in the number of states of the automaton. The members of IR_d can be derived easily from the dead states of d.

4.2 Schema Cast Validation

The problem that we address is the following: Given two deterministic finite-state automata, $a = (Q_a, \Sigma_a, \delta_a, q_a^0, F_a)$, and $b = (Q_b, \Sigma_b, \delta_b, q_b^0, F_b)$, and a string $s \in L(a)$, does $s \in L(b)$? One could, of course, scan s using b to determine acceptance by b. When many strings that belong to $L(a)$ are to be validated with respect to $L(b)$, it can be more efficient to prepreprocess a and b so that the knowledge of s's acceptance by a can be used to determine its membership in $L(b)$. Without loss of generality, we assume that $\Sigma_a = \Sigma_b = \Sigma$.

Our method for the efficient validation of a string $s = s_1 \cdot s_2 \cdot \ldots \cdot s_n$ in $L(a)$ with respect to b relies on evaluating s on a and b in parallel. Assume that after parsing a prefix $s_1 \cdot \ldots \cdot s_i$ of s, we are in a state $q_1 \in Q_a$ in a, and a state $q_2 \in Q_b$ in b. 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 a accepts s), which implies that $s_{i+1} \cdot \ldots \cdot s_n$ will be in $L(q_2)$. By definition of $L(q)$, b 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 $L(b)$, and therefore, b will not accept s.

We construct an immediate decision automaton, c_{immed} from the intersection automaton c of a and b, with IR_c and IA_c based on the two conditions above:

Definition 7. *Let* $c = (Q_c, \Sigma, \delta_c, q_c^0, F_c)$ *be the intersection automaton derived from two finite state automata* a *and* b*. The derived immediate decision automaton is* $c_{immed} = (Q_c, \Sigma, \delta_c, q_c^0, F_c, IA_c, IR_c)$*, where:*

- $IA_c = \{ (q_a, q_b) \mid ((q_a, q_b) \in Q_c) \wedge L(q_a) \subseteq L(q_b)\}$.
- $IR_c = \{ (q_a, q_b) \mid ((q_a, q_b) \in Q_c) \wedge (q_a, q_b) \text{ is a dead state } \}$.

Theorem 3. *For all* $s \in L(a)$*,* c_{immed} *accepts* s *if and only if* $s \in L(b)$*.* □

The determination of the members of IA_c can be done efficiently for deterministic finite state automata. The following proposition is useful to this end.

Proposition 1. *For any state,* $(q_a, q_b) \in Q_c, L(q_a) \subseteq L(q_b)$ *if and only if* $\forall s \in \Sigma^*$*, there exist states* q_1 *and* q_2 *such that* $\delta_c((q_a, q_b), s) \rightarrow (q_1, q_2)$ *and if* $q_1 \in F_a$ *then* $q_2 \in F_b$*.* □

We now present an alternative, equivalent definition of IA_c.

Definition 8. *For all $q' = (q_a, q_b)$, $q' \in IA_c$ if $\forall s \in \Sigma^*$, there exist states (q_1, q_2), such that $\delta_c((q_a, q_b), s) \rightarrow (q_1, q_2)$ and if $q_1 \in F_a$ then $q_2 \in F_b$.*

In other words, a state $(q_a, q_b) \in IA_c$ if for all states (q'_a, q'_b) reachable from (q_a, q_b), if q'_a is a final state of a, then q'_b is a final state of b. It can be shown that the two definitions, 7 and 8, of IA_c are equivalent.

Theorem 4. *For deterministic immediate decision automata, Definition 7 and Definition 8 of IA_c are equivalent, that is, they produce the same set IA_c.* □

Given two automata a and b, we can preprocess a and b to efficiently construct the immediate automaton c_{immed}, as defined by Definition 7, by finding all dead states in the intersection automaton of a and b to determine IR_c. The set of states, IA_c, as defined by Definition 8, can also be determined, in linear time, using an algorithm similar to that for the identification of dead states. At runtime, an efficient algorithm for schema cast validation without modifications is to process each string $s \in L(a)$ for membership in $L(b)$ using c_{immed}.

4.3 Schema Casts with Modifications

Consider the following variation of the schema cast problem. Given two automata, a and b, a string $s \in L(a)$, $s = s_1 \cdot s_1 \cdot \ldots \cdot s_n$, is modified through insertions, deletions, and the renaming of symbols to obtain a string $s' = s'_1 \cdot s'_1 \cdot \ldots \cdot s'_m$. The question is does $s' \in L(b)$? We also consider the special case of this problem where $b = a$. This is the single schema update problem, that is, verifying whether a string is still in the language of an automaton after a sequence of updates.

As the updates are performed, it is straightforward to keep track of the leftmost location at which, and beyond, 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 $s \in L(a)$ 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 schema cast problem without modifications.

Specifically, to determine the validity of s' according to b, we first process b to generate an immediate decision automaton, b_{immed}. We also process a and b to generate an immediate decision automata, c_{immed} as described in the previous section. Now, given a string s' where the leftmost unmodified position is i, we:

1. Evaluate $s'_1 \cdot \ldots s'_{i-1}$ using b_{immed}. That is, determine $q_b = \delta_b(q^0_a, s'_1 \cdot \ldots \cdot s'_{i-1})$. While scanning, b_{immed} may immediately accept or reject, at which time, we stop scanning and return the appropriate answer.
2. Evaluate $s_1 \cdot \ldots \cdot s_{n-m+i-1}$ using a. That is, determine $q_a = \delta_a(q^0_a, s_1 \cdot \ldots \cdot s_{n-m+i-1})$.
3. If b_{immed} scans $i-1$ symbols of s' and does not immediately accept or reject, we proceed scanning $s'_i \cdot \ldots \cdot s'_m$ using c_{immed} starting in state $q' = (q_a, q_b)$.
4. If c_{immed} accepts, either immediately or by scanning all of s', then $s' \in L(b)$, otherwise the string is rejected, possibly by entering an immediate reject state.

Proposition 2. *Given automata a and b, an immediate decision automaton constructed from the intersection automaton of a and b, and strings $s = s_1 \cdot \ldots \cdot s_n \in L(a)$ and $s' = s'_1 \cdot \ldots \cdot s'_m$ such that $s'_i \cdot \ldots \cdot s'_m = s_{n-m+i} \cdot \ldots \cdot s_n$. If $\delta_a(q^0_a, s_1 \cdot \ldots \cdot s_{n-m+i-1}) = q_a$ and $\delta_b(q^0_b, s'_1 \cdot \ldots \cdot s'_{i-1}) = q_b$, then $s' \in L(b)$ if and only if c_{immed}, starting in the state (q_a, q_b) recognizes $s'_i \cdot \ldots \cdot s'_m$.*

The algorithm presented above functions well when most of the updates are in the beginning of the string, since all portions of the string up to the start of the unmodified portion must be processed by b_{immed}. In situations where appends are the most likely update operation, the algorithm as stated will not have any performance benefit. One can, however, apply a similar algorithm to the reverse automata[3] of a and b by noting the fact that a string belongs to $L(b)$ if and only if the reversed string belongs to the language that is recognized by the reverse automaton of b. Depending on where the modifications are located in the provided input string, one can choose to process it in the forward direction or in the reverse direction using an immediate decision automaton derived from the reverse automata for a and b. In case there is no advantage in scanning forward or backward, the string should simply be scanned with b_{immed}.

5 Optimality

An immediate decision automaton c_{immed} derived from deterministic finite state automata a and b as described previously, and with IA_c and IR_c as defined in Definition 7 is optimal in the sense that there can be no other deterministic immediate decision automaton d_{immed} that can determine whether a string s belongs to $L(b)$ earlier than c_{immed}.

Proposition 3. *Let d_{immed} be an arbitrary immediate decision automaton that recognizes exactly the set $L(a) \cap L(b)$. For every string $s = s_1 \cdot s_2 \cdot \ldots \cdot s_n$ in Σ^*, if d_{immed} accepts or rejects s after scanning i symbols of $s, 1 \leq i \leq n$, then c_{immed} will scan at the most i symbols to make the same determination.* □

Since we can efficiently construct IA_c as defined in Definition 7, our algorithm is optimal. For the case with modifications, our mechanism is optimal in that there exists no immediate decision automaton that can accept, or reject, s' while scanning fewer symbols than our mechanism.

For XML Schema, as with finite state automata, our solution is optimal in that there can be no other algorithm, which preprocesses only the XML Schemas, that validates a tree faster than the algorithm we have provided. Note that this optimality result assumes that the document is not preprocessed.

Proposition 4. *Let $T = (t, \lambda)$ be an ordered, labeled tree valid with respect to an abstract XML Schema S. If the schema cast validation algorithm accepts or rejects T after processing node n, then no other deterministic algorithm that:*

- *Accepts precisely $valid(S) \cap valid(S')$.*

[3] The reverse automata of a deterministic automata may be non-deterministic.

– *Traverses T in a depth-first fashion.*
– *Uses an immediate decision automaton to validate content models.*

can accept or reject T before visiting node n. □

6 Experiments

We demonstrate the performance benefits of our schema cast validation algorithm by comparing our algorithm's performance to that of Xerces [2]. We have modified Xerces 2.4 to perform schema cast validation as described in Section 3.2. The modified Xerces validator receives a DOM [19] 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 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 4 to optimize the checking of whether the labels of the children of a node fit the node's content model. In both the modified Xerces and the original Xerces implementation, the content model 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 Figure 1a is validated with respect to the schema of Figure 1b. The complete schema of Figure 1b is provided in Figure 2. In the second experiment, we modify the quantity element declaration (in items) in the schema of Figure 2 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 Figure 2. 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 Figure 2. We vary the number of item elements from 2 to 1000. Table 2 lists the file size of each document. Figure 3a 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. Figure 3b shows the results of the second experiment. The schema cast validation algorithm is about 30% faster than the unmodified Xerces algorithm. Table 3 lists the number of nodes visited by both algorithms. By only traversing the quantity child of item and not the other

```
<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. 2. Target XML Schema.

children of item, our algorithm visits about 20% 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.

Table 2. File sizes for input documents.

# Item Nodes	Size (Bytes)
2	990
50	11,358
100	22,158
200	43,758
500	108,558
1000	216,558

7 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

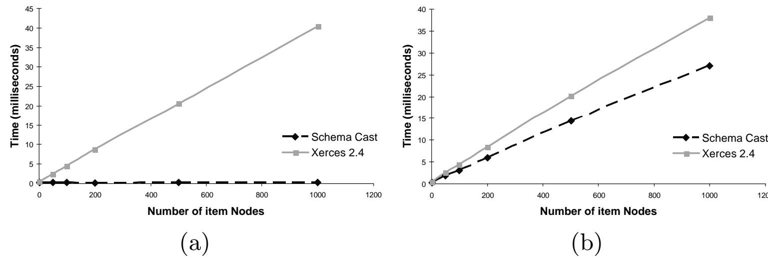

Fig. 3. (a) Validation times from first experiment. (b) Validation times from second experiment.

Table 3. Number of nodes traversed during validation in Experiment 2.

# Item Nodes	Schema Cast	Xerces 2.4
2	35	74
50	611	794
100	1,211	1,544
200	2,411	3,044
500	6,011	7,544
1000	12,011	15,044

before revalidation, and the case where insertions, updates and deletions are applied to the document before revalidation. We have provided an algorithm for the case where validation is defined in terms of XML Schemas (with DTDs as a special case). The algorithm relies on a subalgorithm to revalidate content models efficiently, which addresses the problem of revalidation with respect to deterministic finite state automata. The solution to this schema cast problem is useful in many contexts ranging from the compilation of programming languages with XML types, to handling XML messages and Web Services interactions.

The practicality and the efficiency of our algorithms has been demonstrated through experiments. Unlike schemes that preprocess documents (that handle a subset of our schema cast validation 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 currently extending our algorithms to handle key constraints, and exploring how a system may automatically correct a document valid according to one schema so that it conforms to a new schema.

Acknowledgments

We thank the anonymous referees for their careful reading and precise comments. We also thank John Field, Ganesan Ramalingam, and Vivek Sarkar for comments on earlier drafts.

References

1. S. Alagic and D. Briggs. Semantics of objectified XML. In *Proceedings of DBPL*, September 2003.
2. Apache Software Foundation. *Xerces2 Java Parser*. `http://xml.apache.org/`.
3. D. Barbosa, A. Mendelzon, L. Libkin, L. Mignet, and M. Arenas. Efficent incremental validation of XML documents. In *Proceedings of ICDE*, 2004. To Appear.
4. V. Benzaken, G. Castagna, and A. Frisch. Cduce: an XML-centric general-purpose language. In *Proceedings of ICFP*, pages 51–63, 2003.
5. B. Bouchou and M. Halfeld-Ferrari. Updates and incremental validation of XML documents. In *Proceedings of DBPL*, September 2003.
6. A. Bruggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 142(2):182–206, May 1998.
7. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1989.
8. *Galax: An implementation of XQuery*. `http://db.bell-labs.com/galax`.
9. M. Harren, M. Raghavachari, O. Shmueli, M. Burke, V. Sarkar, and R. Bordawekar. XJ: Integration of XML processing into Java. Technical Report RC23007, IBM T.J. Watson Research Center, 2003. Submitted for Publication.
10. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 2002.
11. H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of ICFP*, 2000.
12. B. Kane, H. Su, and E. A. Rundensteiner. Consistently updating XML documents using incremental constraint check queries. In *Proceedings of the Workshop on Web Information and Data Management (WIDM'02)*, pages 1–8, November 2002.
13. G. Kuper and J. Siméon. Subsumption for XML types. In *Proceedings of ICDT*, January 2001.
14. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proceedings of PODS*, pages 11–22. ACM, 2000.
15. M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.
16. Y. Papakonstantinou and V. Vianu. DTD inference for views of XML data. In *Proceedings of PODS*, pages 35–46. ACM, 2000.
17. Y. Papakonstantinou and V. Vianu. Incremental validation of XML documents. In *Proceedings of ICDT*, pages 47–63, January 2003.
18. J. Siméon and P. Wadler. The essence of XML. In *Proceedings of POPL*, pages 1–13. ACM Press, January 2003.
19. World Wide Web Consortium. *Document Object Model Level 2 Core*, November 2000.
20. World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, November 2000.
21. World Wide Web Consortium. *XML Schema, Parts 0,1, and 2*, May 2001.