XML Schema Computations: Schema Compatibility Testing and Subschema Extraction

Thomas Y. Lee
Department of Computer Science
University of Hong Kong
ytlee@cs.hku.hk

David W. Cheung
Department of Computer Science
University of Hong Kong
dcheung@cs.hku.hk

ABSTRACT

In this paper, we propose new models and algorithms to perform practical computations on W3C XML Schemas, which are schema minimization, schema equivalence testing, subschema testing and subschema extraction. We have conducted experiments on an e-commerce standard XSD called xCBL to demonstrate the effectiveness of our algorithms. One experiment has refuted the claim that the xCBL 3.5 XSD is compatible with the xCBL 3.0 XSD. Another experiment has shown that the xCBL XSDs can be effectively trimmed into small subschemas for specific applications, which has significantly reduced schema processing time.

Categories and Subject Descriptors

F.1.1 [**Theory of Computation**]: Computation by Abstract Devices—*models of computation*; D.2.12 [**Software**]: Software Engineering—*interoperability*

General Terms

Algorithms, Experimentation, Theory

Keywords

XML Schema Computations, Schema Automata

1. INTRODUCTION

Interoperability is a key consideration for implementation of web services. Web service standards (e.g., SOAP[14]) provide the messaging protocols for heterogeneous applications to interoperate. However, beyond this *technology interoperability*, *data interoperability* is a more important but more complex problem to address. *Data interoperability* concerns whether the XML data from one web service can be processed by another web service. This is more complex than technology interoperability because this has to be resolved application by application. Various initiatives, such as Universal Business Language (UBL)[1], are established to standardize XML messages for business applications. Nevertheless, these data standards can only reduce the complexity of the data interoperability between web services but cannot provide real plug-and-play solutions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'10, October 26–30, 2010, Toronto, Ontario, Canada.
Copyright 2010 ACM 978-1-4503-0099-5/10/10 ...$10.00.

Website	#XSDs	#DTDs	#RNGs+RNCs
w3.org	1,650	450	317+62=379
oasis-open.org	1,260	150	185+124=309
total:	2,910 (69%)	600 (14%)	688 (16%)

Table 1: Numbers of schema files in different formats published on W3C and OASIS

1.1 XML Schema Languages and Standards

The XML structures permitted by an application can be defined by an XML schema language. For example, a *product quotation* web service receives an RFQ (request for quote) document, and then sends a Quote document. The RFQ schema defines the set of all possible XML messages that can be *accepted* by this product quotation service while the Quote schema defines the set of all possible XML messages that can be *generated* by the service. The data interoperability between two web services depends on the schemas they use. Popular XML schema languages include *Datatype Definition (DTD)*[7], *W3C XML Schema (XSD)*[10], and *RelaxNG*[8]. Table 1 lists the numbers of schema files in four formats, which are XSD, DTD, RelaxNG XML (RNG), and RelaxNG compact (RNC), published in the W3C[3] and OASIS[2] websites.[1] This shows the majority of the schema files published in these websites are written in XSD.

Many e-business standards are defined in XSD; some of these are very large. Two popular e-business standards are *XML Common Business Library (xCBL)*[4], and OASIS *Universal Business Language (UBL)*[1]. Table 2 lists the numbers of datatypes (#types), element declarations (#edecls), document types (#doctypes), XSD files (#files), and the file size (size) of xCBL 3.0, xCBL 3.5, xCBL 4.0, UBL 1.0, and UBL 2.0. Such a standard is a schema library, which may contain thousands of datatype and element definitions. Many different document types (e.g., Quote, Order, Invoice) are usually specified in a single standard. Generally, each document type is not defined as an independent XSD. Some datatype definitions may be shared among different document types. For example, the PostalAddress datatype

[1]The numbers were reported by Google Search as of May 2010.

XSD	xCBL3.0	xCBL3.5	xCBL4.0	UBL1.0	UBL2.0
#types	1,290	1,476	830	226	682
#edecls	3,728	4,473	2,941	1,098	2,918
#doctypes	42	51	44	8	31
#files	413	496	709	27	43
size (MB)	1.8	2.0	6.3	0.9	2.7

Table 2: XSD sizes of xCBL and UBL standards

may be defined only once but can be reused by many document types, e.g., Order and Invoice.

In reality, even though two web services apply the same data standard, they need not be interoperable with each other. Usually, a specific web service handles only several XML document types. For example, a UBL-based product quotation web service only needs to process RFQ and Quote documents and can safely ignore other irrelevant document types. In other words, this service only needs to process a subset of UBL instances. It is also typical that a web Service needs to restrict a data standard to meet its specific business requirements. For example, the ⟨PostalCode⟩ element might be defined as optional in UBL because not all countries use postal codes in their addresses. However, a web service specific for the USA environment may require ⟨PostalCode⟩ as a mandatory element in all received XML documents.

Therefore, even a sender submits a UBL-compliant document to the web service, the service may still reject the document. Therefore, it is more practical to model an XML message exchange between two web services by a *sending schema* and a *receiving schema*. The sending schema specifies all possible XML messages the sender can generate while the receiving schema specifies all possible messages the receiver can accept. Then, whether two web services are able to exchange all possible messages is determined by whether the receiving schema can accept all possible instances of the sending schema. When the receiving schema can accept all instances of the sending schema, the receiving schema is said to be *compatible* with the sending schema.

1.2 Research Problems

It is often infeasible to manually verify the compatibility on large schemas like xCBL and UBL to prove the data interoperability between web services. This paper discusses the following two *schema compatibility* problems.

Schema compatibility. There are two levels of schema compatibility. First, schema A is *equivalent to* schema B when they accept the same set of instances. Second, A is a *subschema* of B when B accepts every instance of A. The schema compatibility problem is relevant to many applications. The following describes two examples: (1) web service interoperability, and (2) schema version compatibility. On web service interoperability, if web service A needs to accept all messages sent from web service B, the sending schema of B must be a subschema of the receiving schema of A. On schema version compatibility, when a data standard schema is updated to a new version, the new version must be a *superschema* of the old version in order to maintain the backward compatibility. This way, a new application using the new schema version can accept all data generated from an existing application using the previous version.

Subschema extraction. Before an application can use an XML schema to validate XML data, the application is usually required to load and parse the schema into the main memory. In run-time, processing a huge schema may create considerable memory and performance overheads. In design-time, it is very difficult for a programmer to comprehend a huge schema that defines thousands of types and elements when developing an application. In reality, an application usually processes only a few document types defined in a huge schema. For example, a quotation application which processes only Quote and RFQ documents in xCBL 3.5 (i.e., 2 out of 51 document types) only needs to use a small subschema of the huge xCBL 3.5 schema. This example provides a motivation to derive a technique for extracting a trimmed-down subschema that recognizes only a given subset of elements defined in the original schema.

1.3 Contributions

To solve these problems, we have developed two formal models namely *Data Tree* and *Schema Automaton* for modeling hierarchical data instances and schemas respectively. In particular, these models well represent XML documents and schemas. Because of the popularity of XSD, our discussion focuses on how Schema Automata represent XSDs.

We have also formulated two classes of schema computation operations, namely *schema compatibility testing* and *subschema extraction*. These operations are supported by five main algorithms: *schema minimization*, *schema equivalence testing*, *subschema testing*, and *schema extraction*. We have implemented the models and algorithms, and have experimented them with xCBL datasets. The first experiment has refuted the claim of xCBL that v3.5 is compatible with v3.0. In the second experiment, xCBL XSDs have been effectively trimmed down using subschema extraction.

1.4 Organization of This Paper

The rest of this paper is organized as follows. Sect. 2 gives some motivating XSD examples to illustrate the schema compatibility problems. Sect. 3 reviews the related work on XML schema formalisms. Sect. 4 formalizes the models of Data Tree and Schema Automaton. Sect. 5 provides the theorems and algorithms on schema minimization, schema equivalence testing, subschema testing, and subschema extraction. Sect. 6 analyzes the complexity of the algorithms and proposes some techniques to improve performance. Sect. 7 describes the experiments and analyzes their results. Sect. 8 discusses the potential extensions of this research.

2. PRELIMINARIES OF XML SCHEMA

In this section, we use some motivating examples to elaborate how schema compatibility is defined based on XSD. An XSD consists of a set of element declarations and datatype definitions. The elements declared in the top level of the XSD (immediately under ⟨xs : schema⟩) can be used as the root elements of XML instances. An element is bound to some *datatype*. A datatype can be defined as an anonymous type locally within an element declaration. An anonymous datatype can only be bound to its parent element declaration but cannot be reused by other element declarations. (See Listing 1.) A datatype can also be defined globally and assigned with a name such that this named datatype can be reused by multiple element declarations. (See Listing 2.) Moreover, there are two kinds of datatypes: *complex types* and *simple types*. When a parent element contains some child elements or attributes, this parent element must be declared with a complex type. In contrast, a simple type defines the value space for an attribute an element which

has no attribute or child element. XSD has defined a set of built-in simple types for extension or restriction to user-defined simple types.

The following XSD examples help explain our research problems. Listing 1 (XSD 1) and Listing 2 (XSD 2) are two different XSDs that accept the same set of XML instances. They are considered *equivalent*. An XML instance must have the root element named either $\langle$Quote$\rangle$ or $\langle$Order$\rangle$. (The documents with two different root element names can be regarded as two different document types.) Listing 3 and Listing 4 are two instances of XSD 1 and XSD 2.

Listing 1: XSD 1 for Quote and Order documents

```
<xs:schema xmlns:xs=
           "http://www.w3.org/2001/XMLSchema">
 <xs:element name="Quote">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Line" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Desc" type="xs:string"/>
       <xs:element name="Price" type="xs:decimal"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="Order">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="Line" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Product">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="Desc" type="xs:string"/>
          <xs:element name="Price" type="xs:decimal"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="Qty" type="xs:int"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

XSD 1 is *larger* than XSD 2 despite their equivalence. XSD 1 defines 5 complex types and declares 10 elements while XSD 2 has only 4 complex type definitions and 8 element declarations. In XSD 1, each complex type is defined as an anonymous type; hence, there is no reuse of type definitions. On the contrary, XSD 2 defines each complex type as a named datatype so that multiple element declarations can reference the same type and reuse its content model. In fact, XSD 2 has maximized type reuse and represents a *minimal schema*.

Listing 2: XSD 2 for Quote and Order documents

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="Quote" type="QuoteType"/>
 <xs:element name="Order" type="OrderType"/>
 <xs:complexType name="QuoteType">
  <xs:sequence>
   <xs:element name="Line" type="ProdType"
               maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="OrderType">
  <xs:sequence>
   <xs:element name="Line" type="OrderLineType"
               maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="ProdType">
  <xs:sequence>
   <xs:element name="Desc" type="xs:string"/>
   <xs:element name="Price" type="xs:decimal"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="OrderLineType">
  <xs:sequence>
   <xs:element name="Product" type="ProdType"/>
   <xs:element name="Qty" type="xs:int"/>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Listing 3: XML Quote

```
<Quote>
 <Line>
  <Desc>hPhone</Desc>
  <Price>499.9</Price>
 </Line>
 <Line>
  <Desc>iMat</Desc>
  <Price>999.9</Price>
 </Line>
</Quote>
```

Listing 4: XML Order

```
<Order>
 <Line>
   <Product>
    <Desc>hPhone</Desc>
    <Price>499.9</Price>
   </Product>
  <Qty>2</Qty>
 </Line>
</Order>
```

XSD 3 (Listing 5) is a subschema of both XSD 1 and XSD 2. XSD 3 accepts only the instances with Quote as the root element and rejects other instances. For example, XSD 3 accepts the XML document in Listing 3 but rejects the one in Listing 4. XSD 3 is even smaller than XSD 2 and contains only 2 complex types and 4 elements.

Listing 5: XSD 3 as subschema of XSD 1 and XSD 2

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="Quote" type="q1"/>
 <xs:complexType name="q1">
  <xs:sequence>
   <xs:element name="Line" type="q9" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="q9">
  <xs:sequence>
   <xs:element name="Desc" type="xs:string"/>
   <xs:element name="Price" type="xs:decimal"/>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

Regarding the above examples, we provide the formal models and algorithms to solve the following problems.

Schema compatibility testing. (1) How to verify XSD 1 and XSD 2 are equivalent. (2) How to verify XSD 3 is a subschema of XSD 1 and XSD 2.

Subschema extraction. Given XSD 2 (or XSD 1), how to extract a smaller subschema XSD 3 when XSD 3 only needs to recognize the elements in a Quote document.

3. RELATED WORK

This section describes some existing work on XML Schema formalisms. Despite wider industry adoption and more expressive power of XSD over DTD, DTD has attracted more research efforts than XSD. Martens et al. attributed this to the perceived simplicity of DTD and the alleged impenetrability of XSD.[12] Papakonstantinou and Vianu[15] proposed a specialized or *extended DTD (EDTD)* model, adding ele-

ment typing to DTD. EDTD is theoretically backed by the *tree automata* theory[9] for unranked trees.

Although EDTD has added types to DTD, it is different from XSD. Unlike XSD, an EDTD may be non-deterministic. Yet, there is a special class of EDTDs called *single-type and restrained competition* EDTDs[13], with which validation of XML trees is top-down deterministic. Also, EDTD is different from XSD. In EDTD, a type is associated with a regular expression over types, where each type is uniquely mapped to an element. In contrast, in XSD, a type is associated with a regular expression over elements and each element is mapped to a type. The type reuse in XSD is more efficient than that in EDTD. In EDTD, the same content model for two elements of different names must be defined as two different types while in XSD, this content model only requires one type definition.

In another paper, Martens et al. proposed a more accurate XSD abstraction called *XSchema*.[12] Because of the Element Declaration Consistent (EDC) constraint of XSD, Martens et al. defined a special class of XSchema called *single-type XSchema* as an abstraction of XSD. The Schema Automaton (SA) model to be proposed in Sect. 4 resembles the single-type XSchema model in representing XSDs. Yet, SA provides a richer abstraction of XSD than XSchema does for two main reasons. First, an XSchema does not validate data values inside elements. In other words, it does not model XSD simple types and built-in types. SA models simple types as *value domains*. Second, SA uses a different formalism that better facilitates schema computations, such as schema minimization, schema compatibility testing, and schema extraction. These computational problems have not been studied for XSchema. Also, some important concepts proposed in this paper, such as the usefulness of XSD types, have not been studied for XSchema too.

4. DATA TREE & SCHEMA AUTOMATON

In this section, we formalize the models of *Data Tree (DT)* and *Schema Automaton (SA)*. A DT is a tree-form data structure. An SA is a *deterministic finite automaton (DFA)* to recognize DTs. We also elaborate how DT and SA can be used to model XML documents and XSDs.

4.1 Data Tree

A DT is a generic tree-form data model. Each tree node is called *data node (d-node)*, which can store a data value. A d-node may have some child d-nodes. The parent is connected to each child by an edge called *data edge (d-edge)*. Each d-edge is labeled with a *symbol*. See Definition 1.

Definition 1. *A Data Tree (DT) is a 7-tuple* $(N, E, Y, n_0, \mathrm{CEdges}, \mathrm{Val}, \mathrm{Sym})$. *$N$ is a finite set of* data nodes *(d-nodes) connected by a finite set of* data edges *(d-edges) E. A d-edge $e \in E$ is an ordered pair (n_{parent}, n_{child}) where $n_{parent} \in N$ is the parent d-node and $n_{child} \in N$ is the child d-node.* $\mathrm{CEdges} : N \mapsto E^*$ *is a function that takes every d-node $n_{parent} \in N$ to a finite (possibly empty) sequence of child d-edges* $\mathrm{CEdges}(n_{parent}) = e_1 e_2 \ldots e_k$, *where $e_1 \ldots e_k \in E$. A DT has exactly one* root d-node *$n_0 \in N$. Except the root d-node, every other d-node has exactly one parent, and is a descendant of the root via a unique path of d-edges. Every d-node stores a data value. The function* $\mathrm{Val} : N \mapsto \mathcal{V}$ *returns the* data value $\mathrm{Val}(n)$ *of d-node n. A d-node may store the* null *value denoted ϵ, i.e., the empty string. $\mathcal{V}$ denotes the universe of all possible data values, including ϵ. Every d-edge is labeled with a* symbol *and Y is the set of these symbols.* $\mathrm{Sym} : E \mapsto Y$ *is a function that returns the symbol* $\mathrm{Sym}(e)$ *of d-edge e. (Two different d-edges can be labeled with the same symbol.)*

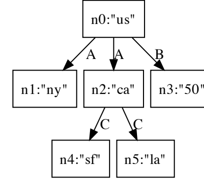


Figure 1: DT 1

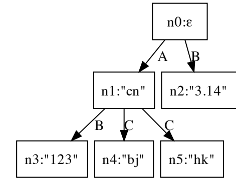


Figure 2: DT 2

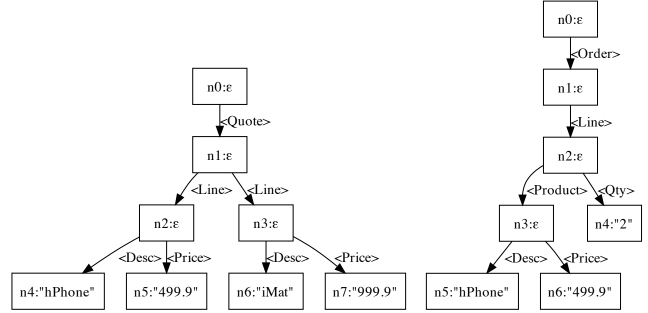


Figure 3: DT for Quote document

Figure 4: DT for Order document

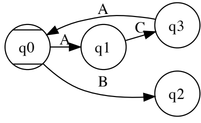
Fig. 1 and Fig. 2 show two DT examples. A box represents a d-node. A directed edge from a parent to its child represents a d-edge. Each d-node (e.g., n1) stores a value (e.g., "ny"). Each d-edge is labeled with a symbol (e.g., B between n0 and n3). n0 is the *root d-node*.

4.1.1 Modeling XML

An XML document can be modeled by a DT. An XML element is represented by a d-edge together with its child d-node. The element name is given by the symbol of the d-edge. The content of an element is given by the child d-node. The value of the element is the textual value of the d-node. If the element has some child elements, these child elements are represented by the child d-edges and d-nodes in the next level. Since an XML document has exactly one root element, the DT modeling an XML document has exactly one child d-edge from the root d-node. Fig. 3 and Fig. 4 show the DTs representing the XML documents in Listing 3 and Listing 4 respectively.

4.2 Schema Automaton

A Schema Automaton (SA) defines the permissible structures and contents of DTs. Essentially, an SA uses a set of *regular languages* to define how d-edges can be sequenced and uses a set of *value domains* (VDoms) to constrain the data values of d-nodes. (Each VDom is a set of values.) First, an SA uses a regular language called *vertical language* (VLang) to define the permissible sequences of the symbols on the d-edges along all paths from the root to the leaves in a DT. For example, in DT 1, these vertical symbol sequences are A, AC, AC, B.) Second, the SA uses a set of



q	HLang(q)	VDom(q)
q0	A{2,5}B	STRS
q1	C*	STRS
q2	{ϵ}	INTS
q3	A*	STRS

Figure 5: SA example

regular languages called *horizontal languages* (HLangs) to define the permissible symbol sequences of the child d-edges under a d-node. For example, in DT 1, the symbol sequence of the child d-edges under the root n0 is AAB; the symbol sequence of the child d-edges under the leaf n4 is the null string ϵ because n4 has no child. The VLang is specified as a *deterministic finite automaton (DFA)* while the HLangs are specified in *regular expression (RE)*. See Definition 2.

Definition 2. *A Schema Automaton (SA) is a 6-tuple* $A = (Q, X, q_0, \delta, \text{HLang}, \text{VDom})$. *$Q$ is a finite set of states. $q_0 \in Q$ is the initial state. There is one implicit dead state $\perp \notin Q$. X is a finite set of symbols. $\delta : Q \times X \mapsto Q \cup \{\perp\}$ is a function called* transition function *that takes each state $q \in Q$ and each symbol $a \in X$ to the next state $\delta(q, a)$ (possibly $\perp$).* HLang : $Q \mapsto \mathscr{P}(X^*) - \{\emptyset\}$ *is a function that takes every state in Q to a non-empty* regular language *over X, called* horizontal language *(HLang). For any state $q \in Q$, if some symbol a does not occur in any string in* HLang(q) *then $\delta(q, a)$ must be set to $\perp$; otherwise, $\delta(q, a)$ must be set to some state in Q.* VDom : $Q \mapsto \mathscr{P}(\mathcal{V}) - \{\emptyset\}$ *is a function that takes every state $q \in Q$ to a finite and non-empty set of values* VDom(q)*, called* value domain *(VDom). Note that an SA does not explicitly define the set of final states. A state is final when its HLang accepts ϵ.*

Fig. 5 shows an SA example. The set Q of states is $\{\mathsf{q0}, \mathsf{q1}, \mathsf{q2}, \mathsf{q3}\}$. The set X of symbols is $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$. The initial state is q0. The transition function δ is defined with the arrows. For example, the SA transits from q0 to q1 on symbol A, i.e., $\delta(\mathsf{q0}, \mathsf{A}) = \mathsf{q1}$ or $\mathsf{q0} \xrightarrow{\mathsf{A}} \mathsf{q1}$; q0 also goes to the dead state $\perp$ on symbol C. Cyclic transitions are possible, e.g., $\mathsf{q0} \xrightarrow{\mathsf{A}} \mathsf{q1} \xrightarrow{\mathsf{C}} \mathsf{q3} \xrightarrow{\mathsf{A}} \mathsf{q0}$. The table in the figure defines the HLang and VDom for each state. For example, the HLang for q0 is the regular language specified by RE $\mathsf{A}\{2, 5\}\mathsf{B}$, which accepts only the strings with 2 to 5 As followed by exactly one B; the HLang for q2 accepts only the null string. The VDom for q0 are all possible strings (STRS) while the VDom for q1 is the set of all possible integers (INTS).

4.2.1 Schema Automaton Validating Data Tree

An SA validates a DT as follows. The SA first uses the initial state to validate the root d-node of the DT. Suppose the SA is currently validating some d-node n of the DT with some state q. If the value of n is outside the VDom of q or the symbol sequence of the child d-edges of n is outside the HLang of q, then the SA immediately rejects the DT. Otherwise, the SA proceeds to validate every child d-node n_{child} (if any) of n against the next state q' of the transition from q on the symbol of the d-edge (n, n_{child}). If none of the descendant d-nodes in the DT subtree rooted at d-node n is rejected, then it is said that the DT subtree at n is *accepted by* q, or simply n is *accepted by* q. Ultimately, the entire DT *is accepted* by the SA if n_0 is accepted by q_0. In this case, it is also said that the DT is an *instance* of the SA. If an SA accepts a DT then each d-node n in the DT is bound to exactly one state q of the SA, where n is accepted by q. The set of all possible instances of the SA are collectively called the *language of the SA*. See Definition 3. Th SA in Fig. 5 accepts DT 1 (Fig. 1) but rejects DT 2 (Fig. 2).

Definition 3. *Let* $A = (Q, X, q_0, \delta, \text{HLang}, \text{VDom})$ *be an SA,* $T = (N, E, Y, n_0, \text{CEdges}, \text{Val}, \text{Sym})$ *be a DT. T is* accepted by *A when there exists a unique* binding map*,* Bind : $N \mapsto Q$*, that binds every d-node $n \in N$ to exactly one accepting state $q \in Q$ such that all of the following conditions hold.*

1. $\text{Bind}(n_0) = q_0$.
2. *For any $n \in N$,* $\text{Val}(n) \in \text{VDom}(\text{Bind}(n))$.
3. *For any $n \in N$, let* $\text{CEdges}(n) = e_1 \dots e_k$*, and $e_i = (n, n_i)$ for $i = 1, \dots, k$. Define* CSeq : $N \mapsto X^*$ *that takes a d-node n to the string* $\text{CSeq}(n) = \text{Sym}(e_1) \dots \text{Sym}(e_k)$*, which specifies the symbol sequence of the child d-edges of n.*
   (a) $\text{CSeq}(n) \in \text{HLang}(\text{Bind}(n))$.
   (b) $\text{Bind}(n_i) = \delta(\text{Bind}(n), \text{Sym}(e_i))$*, for $i = 1, \dots, k$.*

The set of all instance DTs accepted by A is called the language of A, denoted $\mathscr{L}(A)$.

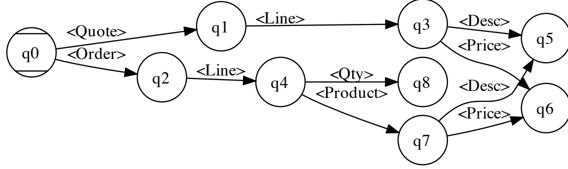
4.2.2 Modeling W3C XML Schema

SA can model the core features of XSD. For example, SA 1 (Fig. 6) and SA 2 (Fig. 7) model XSD 1 (Listing 1) and XSD 2 (Listing 2) respectively. A state in an SA represents an XSD data type, i.e., complex type, simple type, or built-in data type (e.g., xs : string). A symbol represents an element name. A transition from an originating state represents a child element declaration under the complex type represented by this originating state. The destination state of a transition represents the type used by the element declaration. In XSD 2, complex type OrderLineType declares two child elements ⟨Product⟩ and ⟨Qty⟩. ⟨Product⟩ uses complex type ProdType, and ⟨Qty⟩ uses built-in type xs : decimal. States q4, q9, and q8 in SA 2 represent data types OrderLineType, ProdType and xs : int respectively. q4 has two transitions (1) to the next state q9 on symbol ⟨Product⟩, and (2) to q8 on ⟨Qty⟩. Besides, the xs : sequence statement in complex type OrderLineType requires that exactly one ⟨Product⟩ followed by exactly one ⟨Qty⟩ must occur as the children of element ⟨Line⟩. Thus, the HLang of q4 is specified by RE ⟨Product⟩⟨Qty⟩.

Nevertheless, the SA formalized in this paper cannot model some XSD features. For example, it cannot express the xs : any content model, which permits a free structure of any descendant elements. Yet, SA can be further extended to cover these features. Despite some limitation, SA can model most commonly-used XSD features used by industry XSDs, such as xCBL and UBL. We implemented a program to convert XSD to SA, and SA to XSD, which accurately handled the xCBL XSDs for the experiments.

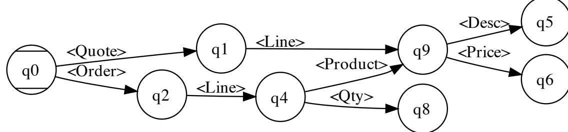
5. XML SCHEMA COMPUTATIONS

This section presents several schema operations using SA. These operations include schema minimization, schema equivalence testing, subschema testing, and subschema extraction. We also analyze the complexity of these operations



q	HLang(q)	VDom(q)	q	HLang(q)	VDom(q)
q0	⟨Quote⟩|⟨Order⟩	$\{\epsilon\}$	q5	$\{\epsilon\}$	STRS
q1	⟨Line⟩+	$\{\epsilon\}$	q6	$\{\epsilon\}$	DECS
q2	⟨Line⟩+	$\{\epsilon\}$	q7	⟨Desc⟩⟨Price⟩	$\{\epsilon\}$
q3	⟨Desc⟩⟨Price⟩	$\{\epsilon\}$	q8	$\{\epsilon\}$	INTS
q4	⟨Product⟩⟨Qty⟩	$\{\epsilon\}$			

Figure 6: SA 1 modeling XSD 1



q	HLang(q)	VDom(q)	q	HLang(q)	VDom(q)
q0	⟨Quote⟩|⟨Order⟩	$\{\epsilon\}$	q4	⟨Product⟩⟨Qty⟩	$\{\epsilon\}$
q1	⟨Line⟩+	$\{\epsilon\}$	q5	$\{\epsilon\}$	STRS
q2	⟨Line⟩+	$\{\epsilon\}$	q6	$\{\epsilon\}$	DECS
q9	⟨Desc⟩⟨Price⟩	$\{\epsilon\}$	q8	$\{\epsilon\}$	INTS

Figure 7: SA 2 modeling XSD 2

and propose some techniques to improve the performance of these operations.

5.1 Schema Minimization

A key operation used in schema compatibility testing and subschema extraction is *schema minimization*. Given an SA, schema minimization computes the equivalent SA that has the fewest states among all equivalent SAs. This minimized SA can be regarded as the *canonical SA* for all SAs recognizing the same language. See Definition 4 on schema equivalence and schema size.

Definition 4. *Let A and A' be two SAs. If $\mathscr{L}(A) = \mathscr{L}(A')$, A and A' are said to be equivalent, denoted $A \equiv A'$.*
Let A be an SA. The size of A, denoted $|A|$, is the number of states in A.

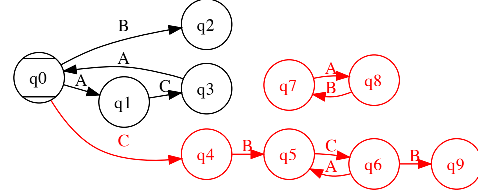
5.2 Usefulness of States (XSD Types)

The first step of minimizing an SA is to remove all *useless states*, each representing a *useless XSD type*. Useless types can be safely discarded from the XSD while its instance set of the resultant XSD is unchanged. Given some SA A, we call a state of A *useful* if some d-node in some instance of A is bound to this state. See Definition 5.

Definition 5. *Let A be an SA and q be a state of A. q is said to be* useful *if there exists some instance T of A and some d-node n in T such that* $\text{Bind}(n) = q$*, where* Bind *is the binding map for A to accept T. A is said to be a* useful SA *if all of its states are useful.*

First, if a state is not *accessible* (Definition 6), then it is never used to recognize any instance; hence, it is useless.

Definition 6. *Let q be a state of an SA. q is said to be accessible if there exists some path of transitions from the initial state to q. Otherwise, q is said to be inaccessible.*



q	HLang(q)	VDom(q)	q	HLang(q)	VDom(q)
q0	A{2,5}BC?	STRS	q5	C	STRS
q1	C*	STRS	q6	A+B*	INTS
q2	$\{\epsilon\}$	INTS	q7	A?	STRS
q3	A*	STRS	q8	B*	STRS
q4	B+	STRS	q9	$\{\epsilon\}$	DECS

Figure 8: Example of SA that contains useless states

Second, if a state is *irrational* then it is useless too. A state is considered irrational if it is on a cycle of *mandatory transitions*. Intuitively, when an SA reaches an irrational state q on a cycle of mandatory transitions while validating some d-node n of a DT, q would require n to have *infinite* descendants. Since a DT is finite, an irrational state never accepts any DT subtree. Therefore, an irrational state is useless.

Definition 7. *Let $A = (Q, X, q_0, \delta, \text{HLang}, \text{VDom})$ be an SA. Some symbol $a \in X$ is a* mandatory symbol *of some state $q \in Q$ if a occurs in every string of the HLang of q. The transition $q \xrightarrow{a} q'$, where $q' \in Q$ (i.e., $q' \neq \perp$), is called a* mandatory transition*. $q_1, \ldots, q_k \in Q$ are said to be* irrational *if there exists a cycle of* mandatory *transitions such that $q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \cdots \xrightarrow{a_k} q_1$ for some symbols $a_1, \ldots, a_k \in X$.*

It is possible for some useful and rational states to be useless too. Each of such states (1) has a path of mandatory transitions to some irrational state or (2) can be reached *only* via useless states. Theorem 1 formalizes the conditions for a state to be useful.

Theorem 1. *Let A be an SA and q be a state of A. q is useless if and only if* any *of the following conditions hold. (1) q is inaccessible. (2) q is irrational. (3) There is a path of mandatory transitions from q to some irrational state. (4) Every transition path from the initial state to q passes through some useless state.*

Fig. 8 shows an example of SA with some useless states. q7 and q8 are inaccessible. q5 and q6 are irrational states because they form a cycle of mandatory transitions. q4 is also useless because it has a mandatory transition to the irrational state q5. However, q0 is not useless because its transition to q4 on symbol C is not mandatory. q9 is also useless because its only transition path from q0 is blocked by useless states q4, q5, and q6. Algorithm 1 (MakeUsefulSA) removes all useless states from a given SA and produces a useful and equivalent SA. Running MakeUsefulSA on Fig. 8 produces the useful SA in Fig. 5.

Algorithm 1. MakeUsefulSA
Input: *SA $A = (Q, X, q_0, \delta, \text{HLang}, \text{VDom})$*
Output: *A is modified so that A is useful*
1: *create an empty list L to store all useless states*
2: *find all mandatory transitions in A*

3: *add all states on any cycles of mandatory transitions, i.e., irrational states, to L*
4: ***while** L is not empty **do***
5: *pick a state q in L and remove q from L*
6: ***if** $q = q_0$ **then***
7: *report no useful SA equivalent to A exists and halt*
8: ***end if***
9: ***for all** $q' \in Q - L$ where there exists $a \in X$ such that $\delta(q', a) = q$ is a mandatory transition **do***
10: *add q' to L*
11: ***end for***
12: *remove all transitions to q /* makes q inaccessible */*
13: ***end while***
14: *traverse A from q_0 and add all inaccessible states to L*
15: ***for all** $q \in Q - L$ where there exist $a_1, \ldots, a_n \in X$ such that $\delta(q, a_1), \ldots, \delta(q, a_n) \in L$ **do***
16: *modify $\mathrm{HLang}(q)$ to a new regular language that is equivalent to the original regular language yet excluding all strings containing any symbol in $\{a_1, \ldots, a_n\}$*
17: ***end for***
18: *remove all states in L together with their incoming and outgoing transitions*

5.2.1 Schema Automaton Minimization

A *minimal* SA of a language is an SA with the fewest states among all SAs accepting the same language. See Definition 8. In fact, this *minimal* SA is *the minimum* (canonical) SA because it is unique up to isomorphism as stated in Theorem 4. Schema minimization involves merging of equivalent states. (See Theorem 2)

THEOREM 2. *Let A be a useful SA and q_1, q_2 be two states of A. q_1 and q_2 are said to be* equivalent *if q_1 and q_2 accept the same set of DT subtrees in all instances of A. q and q' are equivalent if and only if* all *of the following conditions hold. (1) $\mathrm{HLang}(q) = \mathrm{HLang}(q')$. (2) $\mathrm{VDom}(q) = \mathrm{VDom}(q')$. (3) For each $a \in X$, $\delta(q, a) = \delta(q', a) = \perp$ or $\delta(q, a)$ and $\delta(q', a)$ are equivalent.*

DEFINITION 8. *Let A be an SA. If there does not exist another SA A' such that $\mathscr{L}(A') = \mathscr{L}(A)$ and $|A'| < |A|$ then A is called a* minimal SA *of its language.*

THEOREM 3. *Given a useful SA A, for any SA A' equivalent to A, there cannot be fewer states in A' than the equivalence classes of states in A.*

Theorem 3 states that the number of equivalence classes of states in an SA of a language is the lower bound of the size of all SAs accepting the same language. Given any SA, Algorithm 2 computes an SA that is equivalent to the given SA and has as many states as the equivalence classes of states in the given SA. Therefore, the computed SA is a minimal SA of the given SA's language. Essentially, the algorithm combines each class of equivalent states in an input *useful* SA into a new state in the output SA. First, all states in the input SA are partitioned into blocks of the states sharing the same HLang and VDom. Then, each block is examined. When a block contains two states that have transitions on the same symbol to the states in different blocks, the block is split into new blocks, so that all states in each new block have transitions on the same symbol to the states in the same block. The partition is refined iteratively until no new block needs to be split. At that time, every block contains an equivalence classes of states. Finally, all transitions in the input SA from the states in equivalence class B_1 to the states in equivalence class B_2 on the same symbol are combined into a single transition in the minimized SA from new state B_1 to new state B_2 on that symbol.

ALGORITHM 2. MinimizeSA
Input: *useful SA $A = (Q, X, q_0, \delta, \mathrm{HLang}, \mathrm{VDom})$*
Output: *minimum SA $A' = (Q', X', q'_0, \delta', \mathrm{HLang}', \mathrm{VDom}')$ equivalent to A*
1: *create a partition $P = \{B_1, \ldots, B_k\}$ of Q such that for any two states $q_1, q_2 \in Q$, $\mathrm{HLang}(q_1) = \mathrm{HLang}(q_2)$ and $\mathrm{VDom}(q_1) = \mathrm{VDom}(q_2)$ if and only if q_1 are q_2 are in the same B_i, where $1 \leq i \leq k$*
2: *create an empty list L*
3: *add each block $B \in P$ to L if $|B| > 1$*
4: ***while** L is not empty **do***
5: *pick a block B from L and remove B from L*
6: ***if** there exist two states q_1, q_2 in B and some symbol $a \in X$ such that $\delta(q_1, a)$ and $\delta(q_2, a)$ are in different blocks in P **then***
7: *partition B into $R = \{C_1, \ldots, C_m\}$ such that for any two states $q_1, q_2 \in B$, q_1 and q_2 are in the same C_i if and only if $\delta(q_1, a)$ and $\delta(q_2, a)$ are in the same $B' \in P$ for all $a \in X$*
8: *remove B from P and add each $C \in R$ to P*
9: *add $C \in R$ to L for any $|C| > 1$*
10: ***end if***
11: ***end while***
12: *set X' to X; set Q' to P*
13: *set q'_0 to $B \in Q'$ where $q_0 \in B$*
14: ***for all** $B \in Q'$ **do***
15: *set $\mathrm{HLang}'(B)$ to $\mathrm{HLang}(q)$ where $q \in B$*
16: *set $\mathrm{VDom}'(B)$ to $\mathrm{VDom}(q)$ where $q \in B$*
17: *for any $a \in X$, set $\delta'(B, a)$ to B' where $\delta(q, a) = q'$, $q \in B$, and $q' \in B'$*
18: ***end for***

5.3 Schema Equivalence Testing

If two schemas are equivalent, they are compatible with each other. Theorem 4 states that the minimum SA is unique up to isomorphism. Hence, we can test whether two SAs are equivalent by testing whether their minimized forms are isomorphic. (Two SAs are isomorphic when they are "structurally identical" although their states may share different sets of labels.) Algorithm 3 checks the equivalence of two SAs by first minimizing them and then traversing them in parallel from their initial states to check whether they transit in the same way with all HLangs and VDoms matched.

THEOREM 4. *Let A and A' be two equivalent SAs where A and A' are minimal. A and A' are isomorphic, i.e., the* minimum SA *of a language is unique up to isomorphism.*

ALGORITHM 3. EquivalentSA
Input: *SA $A = (Q, X, q_0, \delta, \mathrm{HLang}, \mathrm{VDom})$*
Input: *SA $A' = (Q', X', q'_0, \delta', \mathrm{HLang}', \mathrm{VDom}')$*
Output: true *is returned if $A \equiv A'$;* false *is returned otherwise*
1: $\mathrm{MakeUsefulSA}(A)$; $\mathrm{MakeUsefulSA}(A')$
2: $\mathrm{MinimizeSA}(A)$; $\mathrm{MinimizeSA}(A')$
3: *create a list L that contains one tuple (q_0, q'_0)*

4: *mark* q_0, q'_0 visited
5: ***while*** L *is not empty* ***do***
6: *pick* (q, q') *from* L *and remove* (q, q') *from* L
7: ***if*** $\text{VDom}(q) \neq \text{VDom}'(q')$ *or* $\text{HLang}(q) \neq \text{HLang}'(q)$ ***then***
8: *return* false
9: ***end if***
10: ***for all*** $a \in X$ ***do***
11: $q_1 \leftarrow \delta(q, a)$; $q'_1 \leftarrow \delta(q', a)$
12: ***if*** *exactly one of* q_1, q'_1 *is* $\perp$ ***then***
13: *return* false
14: ***else if*** *both* q_1, q'_1 *are not* $\perp$ ***then***
15: ***if*** *exactly one of* q_1, q'_1 *is* visited ***then***
16: *return* false
17: ***else if*** *both* q_1, q'_1 *are not* visited ***then***
18: *put* (q_1, q'_1) *to* L
19: *mark* q_1, q'_1 visited
20: ***end if***
21: ***end if***
22: ***end for***
23: ***end while***
24: *return* true

SA 1 (Fig. 6) can be minimized to SA 2 (Fig. 7) where states q3 and q7 in SA 1 are combined into q9 in SA 2. Thus, SA 1 and SA 2 are equivalent, which implies the equivalence of their modeled XSD 1 (Listing 1) and XSD 2 (Listing 2).

5.3.1 Subschema Testing

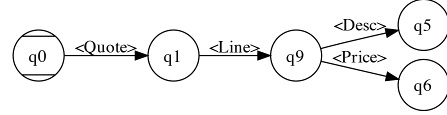
If one schema is a subschema of the other schema, then the latter accepts all instances of the former and thus the latter is compatible with the former one. The subschema notion is formally defined as follows.

DEFINITION 9. *Let A and A' be two SAs. If $\mathscr{L}(A) \subseteq \mathscr{L}(A')$, It is said that A is a* subschema *of A', and A' is* compatible *with A.*

The overall idea of testing whether SA $A = (Q, X, q_0, \delta, \text{HLang}, \text{VDom})$ is a subschema of $A' = (Q', X', q'_0, \delta', \text{HLang}', \text{VDom}')$ is to test whether each possible path of transitions in A can be found in A'. Let $q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_i} q_{i+1} \cdots$ be any transition path in A, where all $q_i \in Q$ and all $a_i \in X$. In order for A' to be a *superschema* of A, the corresponding transition path $q'_0 \xrightarrow{a_0} q'_1 \xrightarrow{a_1} \cdots \xrightarrow{a_i} q'_{i+1} \cdots$ must exist in A' where all $q'_i \in Q'$ and $a_i \in X'$. In addition, the HLang of each q_i must be a subset of the HLang of the corresponding q'_i and the VDom of each q_i must be a subset of the VDom of q'_i too. Otherwise, some values and child sequences of d-nodes that can be accepted by A cannot be accepted by A'. Algorithm 4 (SubschemaSA) performs this subschema testing.

ALGORITHM 4. SubschemaSA

Input: *SA* $A = (Q, X, q_0, \delta, \text{HLang}, \text{VDom})$
Input: *SA* $A' = (Q', X', q'_0, \delta', \text{HLang}', \text{VDom}')$
Output: true *is returned if A is a subschema of A';* false *is returned otherwise*
1: $\text{MakeUsefulSA}(A)$
2: *create a list L that contains one tuple* (q_0, q'_0)
3: *mark the tuple* (q_0, q'_0) visited
4: ***while*** L *is not empty* ***do***



q	HLang(q)	VDom(q)	q	HLang(q)	VDom(q)
q0	⟨Quote⟩	$\{\epsilon\}$	q5	$\{\epsilon\}$	STRS
q1	⟨Line⟩+	$\{\epsilon\}$	q6	$\{\epsilon\}$	DECS
q9	⟨Desc⟩ ⟨Price⟩	$\{\epsilon\}$			

Figure 9: SA 3 modeling XSD 3

5: *pick* (q, q') *from* L *and remove* (q, q') *from* L
6: ***if*** $\text{VDom}(q) \nsubseteq \text{VDom}'(q')$ ***then***
7: *report VDom incompatibility*
8: ***end if***
9: ***if*** $\text{HLang}(q) \nsubseteq \text{HLang}'(q)$ ***then***
10: *report HLang incompatibility*
11: ***end if***
12: ***for all*** $a \in X$ ***do***
13: $q_1 \leftarrow \delta(q, a)$; $q'_1 \leftarrow \delta(q', a)$
14: ***if*** $q_1 \neq \perp$ ***then***
15: ***if*** $q'_1 = \perp$ ***then***
16: *report transition incompatibility*
17: ***else if*** (q_1, q'_1) *is not* visited ***then***
18: *put* (q_1, q'_1) *into* L
19: *mark* (q_1, q'_1) visited
20: ***end if***
21: ***end if***
22: ***end for***
23: ***end while***
24: *return* true

For example, SA 3 (Fig. 9) models the XSD in Listing 5. SubschemaSA can verify that SA 3 is a subschema of SA 1 as well as SA 2.

5.4 Subschema Extraction

Given a large XSD, if an application only needs to recognize a subset of elements, we can *trim* the original schema by extracting a smaller subschema that contains only the needed elements, to save the schema processing time. Given some SA A and a set of *permissible symbols* X', Algorithm 5 (ExtractSubschema) computes another SA A' such that A' accepts all instances of A containing only the symbols in X', and rejects any other DTs. First, all "unwanted" transitions on any symbols outside X' are found and put into a list L pending for deletion. Then, a loop iterates through list L and deletes each unwanted transition. If an unwanted transition $q \xrightarrow{a} q'$ is mandatory, state q should be removed from the extracted schema. This is because the HLang of q does not permit any d-node with no child carrying symbol a. In that case, all transitions going to q also need to be deleted. If $q \xrightarrow{a} q'$ is not mandatory, q need not be deleted. Yet, the HLang of q needs to be modified to a new HLang equivalent to the original HLang minus any strings containing a. After all transitions in L are removed, the resultant schema is minimized into the required subschema.

ALGORITHM 5. ExtractSubschema

Input: *SA* $A = (Q, X, q_0, \delta, \text{HLang}, \text{VDom})$
Input: *a set of permissible symbols* $X' \subseteq X$

Output: *SA A is modified so that the modified A is a subschema of original A and accepts every instance T where T uses only the symbols from X' to label d-edges.*

1: *create a list L that contains all tuples (q,a) where $q \in Q$ and $a \in X - X'$ and $\delta(q,a) \neq \perp$*
2: ***while*** *L is not empty* ***do***
3: *pick (q,a) from L and remove (q,a) from L*
4: *set $\delta(q,a)$ to $\perp$*
5: ***if*** *(q,a) is a mandatory transition* ***then***
6: ***if*** *$q = q_0$* ***then***
7: *report no valid subschema can be extracted and halt*
8: ***end if***
9: ***for all*** *$(q',a') \in Q \times X$ where $\delta(q',a') = q$* ***do***
10: *put (q',a') to L if (q',a') is not in L*
11: ***end for***
12: ***end if***
13: *modify* HLang(q) *to a new regular language such that the new language accepts the same set of strings except those containing symbol a*
14: ***end while***
15: MakeUsefulSA(A)
16: MinimizeSA(A)

Suppose SA 2 (Fig. 7) is given and the permissible symbol set is the whole symbol set of SA 2 excluding ⟨Product⟩, i.e., {⟨Quote⟩, ⟨Order⟩, ⟨Line⟩, ⟨Qty⟩, ⟨Desc⟩, ⟨Price⟩}. The extracted subschema SA is shown in Fig. 9, which corresponds to XSD 3 (Listing 5).

6. COMPLEXITY ANALYSIS

This section analyzes the complexity of the algorithms MakeUsefulSA, MinimizeSA, EquivalentSA, SubschemaSA, and ExtractSubschema. Also, we propose some techniques to speed up their execution. Each algorithm has a while-loop, where the maximum number of iterations is in polynomial order of the number of states. All operations in these algorithms are PTIME except the following two. They are (1) testing whether two REs are equivalent (i.e., $\mathscr{L}(r_1) = \mathscr{L}(r_2)$) and (2) testing whether one RE includes the other (i.e., $\mathscr{L}(r_1) \subseteq \mathscr{L}(r_2)$), which are PSPACE-complete[11].

6.1 Speeding Up Regular Expression Tests

When processing large XSDs, EquivalentSA or SubschemaSA needs to execute a large number of RE tests, which can be very time-consuming. To tackle this issue, we have developed a filtering technique by leveraging some common XSD usage patterns. First, most industry XSDs express xs : complexType content models (i.e., HLangs) in simple combinations of xs : sequence and xs : choice (i.e, REs). Bex et al.[6] suggested that 97% of XSDs expressed the content models in some simple forms of REs. Also, Martens et al.[11] showed that the equivalence and inclusion of some types of these simple REs could be done in PTIME. We have implemented a weak RE test to handle the content models where the occurrence of each xs : sequence or xs : choice must be one yet the occurrence of each xs : element is not restricted. This weak test runs very fast in PTIME. Second, the *equality test* can be used to conclude most positive cases of RE equivalence and inclusion. In reality, developers seldom express two equivalent content models differently, i.e., most equivalent HLangs are *literally equal*. (For example, A+ and AA∗ are equivalent but literally unequal.) Also, in an XSD version update, most complex types in the updated XSD version are the same as those in the old version. While the RE equality test is a sub-linear string matching problem, we can use it to efficiently filter many positive RE equivalence cases. Because of the above properties, we may speed up the RE equivalence / inclusion test as follows. If two REs are literally equal then conclude two REs are equivalent. Otherwise, if the forms of REs are supported by the *weak* test then the weak test on the REs is done. Otherwise, the full test is required. An experiment has showed that the algorithm SubschemaSA using our technique runs 13 times faster than that using only the full test.

7. EXPERIMENTS

This section analyzes the results of two experiments: (1) schema compatibility testing and (2) subschema extraction. The experiments were run on a PC with Quad Core Q6600@2.40GHz, 4GB RAM, and Ubuntu 8.04 (x86) OS. We have implemented the algorithms in Java and have programmed a converter to transform XSD into SA, and SA to XSD. We selected two real datasets, xCBL 3.0 and xCBL 3.5 XSDs, to conduct the above experiments for two reasons. (1) These two datasets are good representatives of very large industry XSDs. (2) xCBL 3.5 is claimed to be compatible with xCBL 3.0, which can be verified by SubschemaSA.

7.1 xCBL Compatibility Testing

The xCBL 3.5 website *claims* its backward-compatibility with xCBL 3.0 as follows: *"The only modifications allowed to xCBL 3.0 documents were the additions of new optional elements and additions to code lists; to maintain interoperability between the two versions. An xCBL 3.0 instance of a document is also a valid instance in xCBL 3.5."*

The above claim implies xCBL 3.0 XSD should be a subschema of xCBL 3.5 XSD. This experiment aimed to verify this claim. The result has surprisingly shown that xCBL 3.0 is in fact *not* a subschema of xCBL 3.5, and has refuted this compatibility claim. The experiment has detected four incompatibility errors. (1) xCBL 3.0 declares a root element Carrier, which does not exist in xCBL 3.5. (2) Under complex type CatalogSchema, element SchemaSource is declared before element ValidateAttributes in xCBL 3.0 but after ValidateAttributes in xCBL 3.5. (3) Under complex type CatalogHeader, element CatalogProvider is declared with minOccurs = "0" in xCBL 3.0 but with minOccurs = "1" in xCBL 3.5. (4) Under complex type SchemaCategory, element CategoryID is declared with minOccurs = "1" in xCBL 3.5 but not declared in xCBL 3.0. If the above errors are fixed, the XSDs can pass the subschema test. We believe these were human editing errors taking place when xCBL 3.0 was manually updated to xCBL 3.5. It is very difficult to manually detect these few errors (0.3%) among thousands of XSD types and elements. Yet, this has caused that a substantial number of xCBL 3.0 instances do not conform to xCBL 3.5.

The experiment also applied the following three filtering strategies to execute the RE inclusion test in SubschemaSA.

1. **Full-only:** It did not use any filtering technique and performed only the full test on every RE comparison.
2. **Weak+full:** It first used the weak inclusion test for simple REs and then used the full test for the REs not supported by the weak test.

strategy	equality tests	weak tests	full tests	time (ms)
full-only	0	0	1,258	3,869
weak+full	0	596	662	536
equality+ weak+full	1,258 (1,196 passed)	59	3	272

Table 3: Performance of different filtering techniques for HLang RE tests

XSD (docs)	enames	types	edecls	ctime (s)	rtime (s)
Original (42)	1,905	1,290	3,728	29.1	N/A
Invoice (8)	904	412	1,154	14.1	3.11
Order (6)	722	352	910	13.2	3.17
Quote (2)	621	299	721	12.9	3.01
Auction (4)	555	266	646	12.6	3.01
Catalog (1)	156	81	190	9.6	2.74

Table 4: Subschema extraction on xCBL 3.0

3. **Equality+weak+full:** Firstly, it used the equality test. Secondly, it used the weak inclusion test for the unequal and simple REs. Lastly, it used the full test if the REs were not supported by the weak test.

Table 3 lists the numbers of three different tests done and the speeds of running SubschemaSA for different strategies. There were 1,258 RE inclusion tests to execute in total. With the weak+full strategy, 596 (47%) weak tests were executed on simple REs; 662 full tests were needed. With the equality+weak+full strategy, 1,196 (95%) RE pairs passed the equality tests; 59 weak tests were executed on simple REs; only 3 full tests were needed. The speedup of the equality+weak+full strategy is over 14 times relative to the full-only strategy.

7.2 xCBL Subschema Extraction

This experiment extracted various subschemas from xCBL 3.0 and 3.5, and examined the reduction of the XSD size and processing time. The XSDs of xCBL 3.0 and 3.5 comprise 42 and 51 business document types respectively (e.g., Quote, Order, and Invoice). These document types are grouped into different domains. For example, the quotation domain consists of RFQ and Quote. ExtractSubschema was first run to extract subschema XSDs from the xCBL 3.0 and xCBL 3.5 XSDs for five domains, namely, invoice, order, quote, auction, and catalog. Then, XMLBeans v2.3.0[5] schema compiler was run to compile each subschema XSD into a Java XML binding library. The number of document types (docs), the number of element names (enames), the number of data types (types) with the percentage of the original number of types, the number of element declarations (edecls), the XMLBeans compilation time (ctime) with the percentage of the original compilation time, and the ExtractSchema running time (rtime) are compared in Table 4 and Table 5. The number of document types (docs) in each domain is indicated in the first column.

For example, the original xCBL 3.0 XSD comprises 1,905 different element names (i.e., symbols), 1,290 data types (i.e., states), and 3,726 element declarations (i.e., transitions) while the subschema for 8 invoice-related document types includes only 904 element names, 412 data types, and 1,154 element declarations. ExtractSubschema can reduce the schema size to a fraction of 6–32%. The time required for XMLBeans to compile each subschema was significantly reduced to a fraction of 34–50%.

XSD (docs)	enames	types	edecls	ctime (s)	rtime (s)
Original (51)	2,263	1,476	4,473	30.5	N/A
Invoice (9)	1,018	460	1,305	15.3	3.28
Order (7)	820	384	1,052	13.7	3.18
Quote (2)	621	319	786	12.7	3.32
Auction (4)	612	291	711	12.4	3.25
Catalog (1)	189	91	231	10.7	2.95

Table 5: Subschema extraction on xCBL 3.5

8. CONCLUSIONS

We anticipate other schema computation techniques can be derived based on SA. Possible extensions of this research are XML schema inferencer and XML transducer. The XML schema inferencer takes a collection XML documents of unknown schema, learns their structures, and re-engineers a "good" XSD to describe the documents. The XML transducer transforms a variety of formats (e.g., structured text and database table formats) into XML documents by annotating the SA that defines the output XML format with the logic to extract data from the input data format. We believe these schema computation techniques can be applied to develop new web services design tools and runtime engines.

9. REFERENCES

[1] OASIS UBL Website. http://www.oasis-open.org/committees/ubl.

[2] OASIS Website. http://www.oasis-open.org.

[3] W3C Website. http://www.w3.org.

[4] xCBL Website. http://www.xcbl.org.

[5] XMLBeans Website. http://xmlbeans.apache.org.

[6] G. J. Bex, F. Neven, and J. V. den Bussche. DTDs Versus XML Schema: a Practical Study. *WebDB*, 2004.

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008.

[8] J. Clark and M. Makoto. RELAX NG Specification, 3 December 2001, 2001.

[9] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. LÃűding, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2007.

[10] D. C. Fallside and P. Walmsley. XML Schema Part 0: Primer (Second Edition), 2004.

[11] W. Martens, F. Neven, and T. Schwentick. Complexity of Decision Problems for Simple Regular Expressions. *MFCS*, 2004.

[12] W. Martens, F. Neven, and T. Schwentick. Simple off the Shelf Abstractions for XML Schema. *SIGMOD RECORD*, 36(3), 2007.

[13] W. Martens and J. Niehren. On the Minimization of XML Schemas and Tree Automata for Unranked Trees. *JCSS*, 73(4), 2007.

[14] N. Mitra and Y. Lafon. SOAP Version 1.2 Part 0: Primer (Second Edition) , 2004.

[15] Y. Papakonstantinou and V. Vianuy. DTD Inference for Views of XML Data. *PODS*, 2000.