Formal Models and Algorithms for XML Data Interoperability

Thomas Y. Lee and David W. Cheung
Department of Computer Science, The University of Hong Kong, Hong Kong, China
{ytlee, dcheung}@cs.hku.hk

Received 1 November 2010; Revised 6 December 2010; Accepted 16 December 2010

In this paper, we study the data interoperability problem of web services in terms of XML schema compatibility. When Web Service $A$ sends XML messages to Web Service $B$, $A$ is interoperable with $B$ if $B$ can accept all messages from $A$. That is, the XML schema $R$ for $B$ to receive XML instances must be compatible with the XML schema $S$ for $A$ to send XML instances, i.e., $A$ is a subschema of $B$. We propose a formal model called Schema Automaton (SA) to model W3C XML Schema (XSD) and develop several algorithms to perform different XML schema computations. The computations include 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 practicality of our algorithms. One experiment has refuted the claim that the xCBL 3.5 XSD is backward 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 the schema processing time.

Categories and Subject Descriptors: Theory of Computation [Computation by Abstract Devices]: Models of Computation

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Web Service data interoperability, XML schema equivalence and compatibility testing, XML subschema extraction, Schema Automaton, Data Tree

1. INTRODUCTION

Today, software applications are commonly implemented as distributed services over the Internet. New distributed computing architectures, such as service-oriented architecture, software as a service, and cloud computing, enable applications to be delivered as web services. One key consideration in implementing these architectures

\footnote{This paper extends the conference paper [Lee and Cheung 2010].}
is how to effectively manage the interoperability between interacting web services. Yet, the term “interoperability” is vaguely defined. Web service standards, such as SOAP [Mitra and Lafon 2004] with Web Services Description Language (WSDL) [Chinnici et al. 2007], and ebXML Messaging Service [Jones et al. 2010], provide the messaging protocols that make applications running on heterogeneous technology platforms interoperate. This technology interoperability is well addressed by many standards consortia, like W3C, OASIS, and WS-I. However, whether two web services are interoperable cannot be simply guaranteed by using the same web service standard.

Another dimension of interoperability is at the data level. The Extensible Markup Language (XML) [Bray et al. 2008] is typically used as the message format for web services to exchange data. This paper studies the data interoperability between two web services concerns whether one web service is able to transmit XML data that can be processed by another web service. Data interoperability is a more complicated problem than technology interoperability because data interoperability often needs to be resolved application by application. Various initiatives, such as [Bosak et al. 2010], [xCBL.org 2000a] and [Lee and Cheung 2010], have been established to standardize XML messages for business applications. Nevertheless, these data standards can only reduce the complexity of data interoperability between web services but cannot provide real plug-and-play solutions.

1.1 XML Schema and Standardized Schema Libraries

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) [Bray et al. 2008], W3C XML Schema (XSD) [Fallside and Walmsley 2004], and RelaxNG [Clark and Makoto 2001]. Table I lists the numbers of schema files in four formats, which are XSD, DTD, RelaxNG XML (RNG), and RelaxNG compact (RNC), published in the W3C and OASIS websites. 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) [xCBL.org

---

Table I. Numbers of schema files in different formats published on W3C and OASIS.

<table>
<thead>
<tr>
<th>Website</th>
<th>#XSDs</th>
<th>#DTDs</th>
<th>#RNGs+RNCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>w3.org</td>
<td>6,490</td>
<td>612</td>
<td>445+37=482</td>
</tr>
<tr>
<td>oasis-open.org</td>
<td>2,330</td>
<td>198</td>
<td>106+128=234</td>
</tr>
<tr>
<td>total</td>
<td>8,820</td>
<td>810</td>
<td>716 (7%)</td>
</tr>
</tbody>
</table>

2The numbers were reported by Google search as of August 2008.
Table II. XSD sizes of xCBL and UBL standards.

<table>
<thead>
<tr>
<th>XSD</th>
<th>xCBL 3.0</th>
<th>xCBL 3.5</th>
<th>xCBL 4.0</th>
<th>UBL 1.0</th>
<th>UBL 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>#types</td>
<td>1,290</td>
<td>1,476</td>
<td>830</td>
<td>226</td>
<td>682</td>
</tr>
<tr>
<td>#edecls</td>
<td>3,728</td>
<td>4,473</td>
<td>2,941</td>
<td>1,098</td>
<td>2,918</td>
</tr>
<tr>
<td>#doctypes</td>
<td>42</td>
<td>51</td>
<td>44</td>
<td>8</td>
<td>31</td>
</tr>
<tr>
<td>#files</td>
<td>413</td>
<td>496</td>
<td>709</td>
<td>27</td>
<td>43</td>
</tr>
<tr>
<td>size (MB)</td>
<td>1.8</td>
<td>2.0</td>
<td>6.3</td>
<td>0.9</td>
<td>2.7</td>
</tr>
</tbody>
</table>

2000a], and OASIS UBL. Table II 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 may be defined only once but can be reused by many document types, e.g., Order and Invoice.

In real business cases, 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. In this example, even a sender submits a URL-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 send and 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 accepts all possible instances of the sending schema. When the receiving schema can accept all instances of the sending schema (i.e., the instance set of the receiving schema is a superset of the instance set 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 schema compatibility on very 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 in many web service 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. Using an XML schema to validate XML data in runtime, an application needs to load and parse the schema into the main memory. Processing a huge schema may cause considerable memory and performance overheads. In design-time, it is 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 documents types defined in a huge schema. For example, a quotation application which processes only \( \text{Quote} \) and \( \text{RFQ} \) documents in xCBL 3.5 (i.e., 2 out of 51 document types) needs to use just a small subschema of the huge xCBL 3.5 schema. This example provides a motivation to derive a technique to extract a trimmed-down subschema that recognizes exactly a given subset of elements defined in the original schema.

1.3 Contributions
To solve the above problems, we have developed two formal models namely Data Tree and Schema Automaton for modeling XML data and schemas respectively. 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 refuted the claim of xCBL [xCBL.org 2000b] that v3.5 is compatible with v3.0. In the second experiment, xCBL XSDs were effectively trimmed down using subschema extraction.

The main contributions our research are summarized as follows:

(1) How the data interoperability between web services is affected by XML schema compatibility has been studied.
(2) New models called Data Tree and Schema Automaton have been proposed to represent XML instances and schemas.
(3) Several practical algorithms based on this formalism for schema compatibility testing and subschema extraction have been developed.
(4) How these algorithms can be used in real industry cases have been demonstrated.
through experiments. One experiment has shown that xCBL 3.5 is in fact *incompatible* with xCBL 3.0 despite the claimed backward compatibility. Another experiment has shown that the subschema extraction algorithm can effectively trim the large xCBL XSDs into small subschemas; the processing time for each subschema has been largely shortened.

### 1.4 Organization of This Paper
The rest of this paper is organized as follows. Section 2 elaborates how the data interoperability between two web services is affected by schema compatibility and gives some motivating XSD examples to illustrate the schema compatibility problems. We also review the related work on XML schema formalisms and computations. Section 4 formalizes the models of Data Tree and Schema Automaton. Section 5 provides the theorems and algorithms on schema minimization, schema equivalence testing, subschema testing, and subschema extraction. Section 7 describes the experiments and analyzes their results. Section 8 describes potential extensions of this research and concludes this paper. Some additional algorithms and the proofs of all theorems are provided as appendices.

## 2. PRELIMINARIES
In this section, we use some motivating examples to elaborate how schema compatibility is defined based on XSD. Then, we review some related work on formalisms and computations on XML schemas.

### 2.1 XML Message Exchange
**One-Way Message Transmission.** The transmission of XML messages between two web services should be modeled by one sending schema and one receiving schema instead of only one schema between them. As shown in Figure 1, the sending schema (the gray arrow on the left) defines the set of all possible XML instances the sender can generate while the receiving schema (the gray arrow on the right) defines the set of all possible instances the receive can accept. Therefore, if it is required for the receiver to accept all messages generated by the sender, the instance set of the receiving schema must be a superset of the instance set of the sending schema. In this case, it is said that the receiving schema is a superschema of the sending schema or the sending schema is a subschema of the receiving schema. In other words, the receiving schema is compatible with the sending schema.
Two-Way Message Exchange. The interaction between two web services is more often a two-way message exchange than a one-way message transmission. In this situation, when web service A invokes another web service B, A initiates the invocation by sending a request message to B. B processes the request and then replies A with a response message. A two-way message exchange is composed of two one-way message transmissions in different directions; each transmission involves one sending schema and one receiving schema. As shown in Figure 2, the interoperability of these two web services requires:

1. the sending schema of the initiator for request messages must be a subschema of the receiving schema of the responder for request messages, and
2. the sending schema of the responder for response messages must be a subschema of the receiving schema of the initiator for response messages.

Now, we discuss what conditions two web services using two different schema versions to interoperate with each other, supposing the new version is backward compatible with the old version. If the responder is required to interoperate with any initiators using any of the two schema versions, the responder must:

1. receive request messages based on the new version, but
2. send response messages based on the old version.

On the other hand, if the initiator is required to interoperate with any responders using any of the two schema versions, the initiator must:

1. send request messages based on the old version, but
2. receive response messages based on the new version.

2.2 W3C XML Schema (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 element or attribute, this parent element must be declared with a complex type. In contrast, a simple type defines the value space for an element or an attribute. XSD has defined a set of built-in simple types for extension or restriction to user-defined simple types.

The following examples of XSDs help explain the 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 to each other. An XML instance must have the root element named either `<Quote>` or `<Order>`. (The documents with two different root element names can be regarded as two different document types.) Inside a Quote...
document are one or more <Line> elements. Under each <Line>, there are one <Desc> element and one <Price> element. A <Desc> contains a product description (string) while a <Price> contains a product price (decimal). In an Order document, there are one or more <Line> elements. Each <Line> contains one <Product> and one <Qty> (integer). A <Product> has one <Desc> (string) and one <Price> (decimal) as children. Listing 3 and Listing 4 are two instances of XSD 1 and XSD 2.

**Listing 1. XSD 1 for Quote and Order documents**

```xml
<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:complexType>
</xs:schema>
```
Listing 2. XSD 2 for Quote and Order documents

```xml
<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>iPhone</Desc>
    <Price>499.99</Price>
  </Line>
  <Line>
    <Desc>iPad</Desc>
    <Price>999.99</Price>
  </Line>
</Quote>
```

Listing 4. XML Order

```
<Order>
  <Line>
    <Desc>iPhone</Desc>
    <Price>499.99</Price>
  </Line>
  <Line>
    <Desc>iPad</Desc>
    <Qty>2</Qty>
  </Line>
</Order>
```

However, 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 definition 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. For example, elements <Quote> and <Product> have reused complex type ProdType. In fact, XSD 2 has maximized type
reuse and represents a *minimal schema*.

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. Note that since the datatype names are only used for referencing within an XSD, these names (e.g., q0 and q1) do not affect the instance set of the XSD.

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

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

Various research projects have proposed different formalisms on XML schema languages. Despite the higher popularity in industry adoption and more power in expressiveness of XSD over DTD, DTD has attracted more research efforts than XSD. [Martens et al. 2007] attributed this to the perceived simplicity of DTD and the alleged impenetrability of XSD.

The primary difference between DTD and XSD in expressiveness is due to the lack of element typing in DTD. XSD allows a content model to be defined as a type, which can be reused by multiple element declarations. For example in the XSD in Listing 2, complex type ProdType is reused in complex types QuoteType and OrderType. However, in DTD, every element must be declared with its own content model. Moreover, DTD only permits all elements with the same name to use the same content model regardless of their contexts. In contrast, XSD allows two elements declared the same name but under different complex types (contexts) to use different content models (types). For example in Listing 2, the elements named hLine under QuoteType and

---

OrderType use different types ProdType and OrderLineType respectively. This XSD example cannot be expressed in DTD.

3.1 Schema Matching

Many research projects have studied the XML schema matching problem. They were motivated by the interoperability of two applications using two schemas with different structures and elements to exchange data. [Milo and Zohar 1998] introduced a system that aimed to automate matching of elements in two schemas by their names and structures. [Madhavan et al. 2001] proposed a similar approach to perform schema matching in three phases. In the first phase, called linguistic matching, the similarity coefficients of elements are calculated based on their names. In the second phase, called structural matching, the similarity coefficients of elements are calculated based on their structures. In the third phase, called mapping generation, mappings of elements in the two schemas are produced based on these similarity coefficients. Other examples of schema matching projects include [Lakshmanan and Sadri 2003], [Algergawy et al. 2009], and [Reynold Cheng 2010].

However, the schema models proposed by many of these projects are over-simplified because they are either proprietary or DTD-like, and cannot represent core XSD constructs. Moreover, these schema matching approaches usually share a common problem that they use “artificial” matching rules. They can only generate a meaningful element mapping when two schemas use very similar element names and very simple structures. For example, the elements <author> and <Author> in two schemas may be matched by these algorithms just because two elements share the same name case-insensitively.

A schema mapping automatically generated by these algorithms without manual adjustment is often unreliable because schemas are not semantically annotated [Rahm and Bernstein 2001]. While schema matching is done at the syntax level rather than the semantic level, two semantically different elements with an identical element name in two schemas may mislead the algorithms to link these unrelated elements together. For example, a <line> element may represent a purchase order line item in one schema but another <line> element may represent an address line in the other schema; it is not correct to match these two unrelated elements just because they have the same name. Also, these schema matching projects have not considered the schema compatibility and subschema extraction problems addressed by our research.

3.2 DTD and Extended DTD

A DTD is commonly abstracted as a set of production rules in the form of \( a \rightarrow r \), where \( a \) is an element and \( r \) is a regular expression over the element set. [Neven et al. 2006] formalized DTD as a tuple \((\Sigma, d, s)\), where \( \Sigma \) is a finite alphabet called element names, \( d \) is a function that maps \( \Sigma \) to regular expressions over \( \Sigma \), and \( s \in \Sigma \) is the start element. For example, a DTD that recognizes the XML document in Listing 3 can be written as follows:

\[
\text{(Quote)} \rightarrow \text{(Line)}^* \\
\text{(Line)} \rightarrow \text{(Desc)/(Price)}
\]
While DTD has weaker expressive power than XSD does, [Papakonstantinou and Vianu 2000] proposed a specialized or Extended DTD (EDTD) model, adding element typing to DTD. EDTD is theoretically backed by the tree automata theory [Comon et al. 2007] for unranked trees. [Martens and Niehren 2007] studied the schema minimization problems for different types of tree automata (UTA). An EDTD is formalized as a tuple \((\Sigma, \Delta, d, s, \mu)\), where:

(1) \(\Delta\) is a finite alphabet called \textit{types},
(2) \(\Sigma\) is a finite alphabet called \textit{element names},
(3) \((\Delta, d, s)\) is a DTD, where \(d\) is a function that maps \(\Delta\) to regular expressions over \(\Delta\) and \(s \in \Delta\) is a start type, and
(4) \(\mu: \Delta \to \Sigma\) is a function that maps each type to some element name.

An EDTD embeds a DTD and adds an additional alphabet and a mapping function. The alphabet used in the embedded DTD is called types, instead of element names. A new alphabet called element names is introduced to the EDTD, with a mapping function \(\mu\) that maps each type to some element name. Intuitively, the mapping function transforms each XML instance \(X\) of the embedded DTD to another XML instance \(Y\) of the EDTD by renaming each type (in \(\Delta\)) to some new element name (in \(\Sigma\)) through \(\mu\). [Neven et al. 2006] studied EDTD and its derivatives in depth. For example, an EDTD that recognizes both documents in Listing 3 and Listing 4 can be written as follows:

<table>
<thead>
<tr>
<th>EDTD Rule</th>
<th>EDTD Rule</th>
<th>EDTD Rule</th>
<th>EDTD Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{QuoteType} \to \text{QuoteLineType}^*)</td>
<td>(\mu(\text{QuoteType}) = \langle\text{Quote}\rangle)</td>
<td>(\mu(\text{QuoteLineType}) = \langle\text{Line}\rangle)</td>
<td>(\mu(\text{OrderType}) = \langle\text{Order}\rangle)</td>
</tr>
<tr>
<td>(\text{QuoteLineType} \to \text{DescTypePriceType})</td>
<td>(\mu(\text{DescType}) = \langle\text{Desc}\rangle)</td>
<td>(\mu(\text{PriceType}) = \langle\text{Price}\rangle)</td>
<td>(\mu(\text{ProductType}) = \langle\text{Product}\rangle)</td>
</tr>
<tr>
<td>(\text{OrderType} \to \text{OrderLineType}^*)</td>
<td>(\mu(\text{OrderLineType}) = \langle\text{Line}\rangle)</td>
<td>(\mu(\text{OrderType}) = \langle\text{Order}\rangle)</td>
<td>(\mu(\text{OrderLineType}) = \langle\text{Order}\rangle)</td>
</tr>
<tr>
<td>(\text{OrderLineType} \to \text{ProductType}^*)</td>
<td>(\mu(\text{ProductType}) = \langle\text{Product}\rangle)</td>
<td>(\mu(\text{OrderType}) = \langle\text{Order}\rangle)</td>
<td>(\mu(\text{ProductType}) = \langle\text{Product}\rangle)</td>
</tr>
<tr>
<td>(\text{ProductType} \to \text{DescTypePriceType})</td>
<td>(\mu(\text{ProductType}) = \langle\text{Product}\rangle)</td>
<td>(\mu(\text{OrderType}) = \langle\text{Order}\rangle)</td>
<td>(\mu(\text{ProductType}) = \langle\text{Product}\rangle)</td>
</tr>
</tbody>
</table>

Although EDTD has added types to DTD, it does not model XSD. In general, an EDTD can be non-deterministic. Yet, there is a special class of EDTDs called single-type and restrained competition EDTDs [Martens and Niehren 2007], with which validation of XML trees is top-down deterministic. This class of EDTD has been shown to have the same expressive power as XSD does. However, 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. For example, in the above EDTD, QuoteLineType and ProductType must be defined as two separate types so that the two types can be mapped to two different element names although the two types share the same content model. As illustrated in the corresponding XSD in Listing 2, this content model only needs to be defined once in ProdType even though it is shared by two elements with different names, i.e., \(<\text{Line}>\) and \(<\text{Product}>\).
3.3 XSchema

[Martens et al. 2007] proposed a more accurate XSD abstraction called \textit{xschema}. An \textit{xschema} is a tuple \((ENames, Types, \rho, t_0)\), where:

1. \(ENames\) is a set of element names,
2. \(Types\) is a set of types,
3. \(\rho : Types \rightarrow ENames \times Types\) is a function that maps a type to some regular expression over an alphabet called \(Elems\ (ENames, Types)\), where \(Elems\ (ENames, Types) = a[t] : a \in EName, t \in Types\), and
4. \(t_0 \in Types\) is the start type.

For example, the XSD in Listing 2 can be modeled as follows:

\[
\begin{align*}
t_0 & \rightarrow \text{Quote}\{\text{QuoteType}\}\langle\text{Order}\{\text{OrderType}\}\rangle \\
\text{QuoteType} & \rightarrow \langle\text{Line}\{\text{ProdType}\}\rangle^* \\
\text{OrderType} & \rightarrow \langle\text{Line}\{\text{OrderLineType}\}\rangle^* \\
\text{ProdType} & \rightarrow \langle\text{Desc}\{\text{Empty}\}\langle\text{Price}\{\text{Empty}\}\rangle\rangle \\
\text{OrderLineType} & \rightarrow \langle\text{Product}\{\text{ProdType}\}\{\text{Qty}\}{\text{Empty}}\rangle \\
\text{Empty} & \rightarrow \epsilon
\end{align*}
\]

Because of the Element Declarations Consistent (EDC) constraint of XSD [Thompson et al. 2004], Martens et al. defined a special class of XSchema called single-type XSchema as an abstraction of XSD. An XSchema is single-type when there is no element \(a[t_1]\) and \(a[t_2]\) in \(\rho(t)\) with \(t_1 \neq t_2\).

The Schema Automaton (SA) model to be proposed in Section 4 resembles the single-type XSchema model in representing XSDs. Yet, SA provides a richer abstraction of XSD than XSchema does because of the following:

1. 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.
2. A formal model for XML documents is not defined with respect to the XSchema model to illustrate how an XSchema can be operated on XML documents. With respect to the SA model, the Data Tree model is defined for XML instances.
3. 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 by this research, such as the usefulness of XSD types, have not been studied for XSchema.

4. DATA TREE AND 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 a 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 a data edge (d-edge). Each d-edge is labeled with a symbol. See Definition 4.1.

Definition 4.1. A Data Tree (DT) is a 7-tuple \((N, E, Y, n_0, \text{CEdges}, \text{Val}, \text{Sym})\). 
\(N\) 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 \((n_{\text{parent}}, n_{\text{child}})\) where \(n_{\text{parent}} \in N\) is the parent d-node and \(n_{\text{child}} \in N\) is the child d-node. \(\text{CEdges} : N \mapsto E^*\) is a function that takes every d-node \(n_{\text{parent}} \in N\) to a finite (possibly empty) sequence of child d-edges \(\text{CEdges}(n_{\text{parent}}) = e_1 e_2 ... e_k\), where \(e_1 ... e_k \in E\). 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 \(\text{Val} : N \mapsto V\) returns the data value \(\text{Val}(n)\) of d-node \(n\). A d-node may store the null value denoted \(\varepsilon\), i.e., the empty string. \(V\) denotes the universe of all possible data values, including \(\varepsilon\). Every d-edge is labeled with a symbol and \(Y\) is the set of these symbols. \(\text{Sym} : E \mapsto Y\) is a function that returns the symbol \(\text{Sym}(e)\) of d-edge \(e\). Note: two different d-edges can be labeled with the same symbol.

Figure 3 and Figure 4 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., \(n_1\)) stores a value (e.g., \("ny"\)). Each d-edge is labeled with a symbol (e.g., B between \(n_0\) and \(n_3\)). \(n_0\) 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. Figure 5 and Figure 6 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 one 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, ACB, B. Second, the SA uses a set of 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 because n4 has no child. The VLang is specified as a deterministic finite automaton (DFA) while the HLangs are specified in regular expression (RE). Definition 4.2 formally defines SA.

Definition 4.2. A Schema Automaton (SA) is a 6-tuple $A=(Q, X, q_0, \delta, \text{HLang}; \text{VDom})$, where:

- $Q$ is a finite set of states. $q_0 \in Q$ is the initial state. There is one implicit dead state $\bot \in Q$. $X$ is a finite set of symbols.
- $\delta$: $Q \times X \rightarrow Q \cup \{\bot\}$ is a function called the transition function that takes each state $q \in Q$ and each symbol $a \in X$ to the next state $\delta(q, a)$ (possibly $\bot$).
- HLang: $Q \rightarrow \mathcal{P}(X^*) - \{\emptyset\}$ 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 $\delta(d, a)$ must be set to $\bot$; otherwise, $\delta(d, a)$ must be set to some state in $Q$. VDom: $Q \rightarrow \mathcal{P}(V) - \{\emptyset\}$ 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 $\varepsilon$.

Figure 7 shows an SA example. The set $Q$ of states is $\{q_0, q_1, q_2, q_3\}$. The set $X$
of symbols is \{A, B, C\}. The initial state is \(q_0\). The transition function \(\delta\) is defined with the arrows. For example, the SA transits from \(q_0\) to \(q_1\) on symbol \(A\), i.e., \(\delta(q_0, A) = q_1\), or \(q_0 \xrightarrow{\text{A}} q_1\); \(q_0\) also goes to the dead state \(\perp\) on symbol \(C\). Cyclic transitions are possible, e.g., \(\xrightarrow{\text{C}}\). The table in the figure defines the HLang and VDom for each state. For example, the HLang for \(q_0\) is the regular language specified by \(RE\ \{2, 5\}B\), which accepts only the strings with 2 to 5 As followed by exactly one \(B\); the HLang for \(q_2\) accepts only the null string. The VDom for \(q_0\) are all possible STRS while the VDom for \(q_1\) 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 \(n_{\text{child}}\) of \(n\) against the next state \(q'\) of the transition from \(q\) on the symbol of the d-edge \((n, n_{\text{child}})\). 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 \(q\). See Definition 4.3. (See Appendix A.1 on the algorithm for an SA to validate a DT.) The set of all possible instances of the SA are collectively called the language of the SA. See Definition 4.4.

Definition 4.3. 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, \(\text{Bind} : N \rightarrow Q\), that binds every d-node \(n \in N\) to exactly one 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 \(\text{CEdges}(n) = e_1 \ldots e_k\), and \(e_i = (n, n_i)\) for \(i = 1, \ldots, k\).
   Define \(\text{CSeq} : N \rightarrow X^*\) that takes a d-node \(n\) to the string \(\text{CSeq}(n) = \text{Sym}(e_1)\ldots\text{Sym}(e_k)\), 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 \(i = 1, \ldots, k\).

It is said that \(n\) is accepted by \(q\) or the DT subtree at \(n\) is accepted by \(q\) when \(n\)
Definition 4.4. Let $A$ be an SA. The set of all instance DTs accepted by $A$ is called the language of $A$, denoted $L(A)$.

The SA in Figure 7 accepts DT 1 (Figure 3) but rejects DT 2 (Figure 4). Table III shows the binding map of DT 1 against the SA. DT 2 is rejected because:

1. $CSeq(n_0) = AB$ is not in $HLang(q_0) = \Sigma(A[2,5]B)$,
2. $CSeq(n_1) = BCC$ is not in $HLang(q_1) = \Sigma(C^*)$, and
3. $Val(n_2) = “3.14”$ is not in $VDom(q_2) = INTS$.

4.2.2 Modeling W3C XML Schema. SA can model the core features of XSD. For example, SA 1 (Figure 8) and SA 2 (Figure 9) 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 $<Price>$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$q = \text{Bind}(n)$</th>
<th>$Val(n)$</th>
<th>$VDom(q)$</th>
<th>$CSeq(n)$</th>
<th>$HLang(q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_0$</td>
<td>$q_0$</td>
<td>“us”</td>
<td>STRS</td>
<td>$AB$</td>
<td>$A[2,5]B$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>$q_1$</td>
<td>“ny”</td>
<td>STRS</td>
<td>$\varepsilon$</td>
<td>$C^*$</td>
</tr>
<tr>
<td>$n_2$</td>
<td>$q_1$</td>
<td>“ca”</td>
<td>STRS</td>
<td>$C$</td>
<td>$C^*$</td>
</tr>
<tr>
<td>$n_3$</td>
<td>$q_2$</td>
<td>“50”</td>
<td>INTS</td>
<td>$\varepsilon$</td>
<td>$[c]$</td>
</tr>
<tr>
<td>$n_4$</td>
<td>$q_3$</td>
<td>“sf”</td>
<td>STRS</td>
<td>$\varepsilon$</td>
<td>$[c]$</td>
</tr>
<tr>
<td>$n_5$</td>
<td>$q_3$</td>
<td>“la”</td>
<td>STRS</td>
<td>$\varepsilon$</td>
<td>$[c]$</td>
</tr>
</tbody>
</table>

Table III. Binding map of DT 1 against SA in Figure 7.
<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>.

4.3 Expressiveness of Schema Automata in Representing W3C XML Schemas

HLangs can represent more sophisticated content models. For example, the content model shown in Listing 6 defines that complex type ct1 has two to three X child elements, followed by either a series of three or more Ys, or a series of one optional X. This model can be specified as an HLang RE <X>{2,3}(<Y>{3,}|<X>?) . Note that XSD does not permit multiple child element declarations with the same element name to be assigned with different types because of the EDC constraint. In Listing 6, the first child element <X> uses built-in type xs : string; if the last element <X> uses a different type, e.g., xs : decimal, then an EDC violation error would occur. The EDC constraint makes the transition of an SA always deterministic, i.e., a state would not move to more than one next state on one symbol.

A built-in data type (e.g., xs : string) or a simple type is modeled by a state that has no child. That means the HLang of this state accepts only the empty string ε. For example, states q5, q6, and q7 in SA 2 represent xs : string, xs : decimal, and xs : integer in XSD 2 respectively. The VDom{s} for q5, q6, and q7 are therefore the set of all possible strings, the set of all decimal numbers, and the set of all integers respectively. A simple type is modeled similarly. For example, in Listing 7, st1 and st2 are both defined based on xs : decimal but the former accepts all decimals greater than zero while the latter accepts all decimals inclusively between −1 and 1. Therefore, st1 and st2 should be represented by two different states with two different VDom{s} described above.
Listing 6. Example of XSD complex type definition
<xsd:complexType name="ct1">
  <xsd:sequence>
    <xsd:element name="X" type="xsd:string" minOccurs="2" maxOccurs="3"/>
    <xsd:choice>
      <xsd:element name="Y" type="xsd:integer" minOccurs="3" maxOccurs="unbounded"/>
      <xsd:element name="Z" type="xsd:string" minOccurs="0"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>

Listing 7. Example of XSD simple type definition
<xsd:simpleType name="st1">
  <xsd:restriction base="xsd:decimal">
    <xsd:minExclusive value="0"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="st2">
  <xsd:restriction base="xsd:decimal">
    <xsd:minInclusive value="-1"/>
    <xsd:maxInclusive value="1"/>
  </xsd:restriction>
</xsd:simpleType>

Nevertheless, it is also known that SA cannot model some XSD features. For example, SA cannot express the xs : any content model, which permits a free structure of any descendant elements. Moreover, the xs : all construct cannot be represented by conventional regular expressions (HLang). Yet, [Tozawa 2005] proposed an extension to the regular expression and algorithms to check the containment between two extended regular expressions. This can be applied to extend SA to support the xs : all construct. Despite some limitation, SA can model most commonly-used XSD features used by industry XSDs, such as xCBL and UBL. We have written a program to convert XSD to SA, and SA to XSD, which could accurately handle the xCBL XSDs in our experiments.

5. XML SCHEMA COMPUTATIONS
This section presents several schema computational operations using SA. These operations include schema minimization, schema equivalence testing, subschema testing, and subschema extraction. We also analyze the complexity of these operations 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 5.1 on schema equivalence and schema size.

Definition 5.1. Let $A$ and $A'$ be two SAs. If $L(A) = L(A')$, $A$ and $A'$ are said to be equivalent, denoted $A \equiv A'$.

Definition 5.2. Let $A$ be an SA. The size of $A$, denoted $|A|$, is the number of states in $A$.

5.2 Usefulness of States (XML Schema 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 5.3.

Definition 5.3. 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 $\text{Bind}$ 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 5.4), then it does not contribute to recognizing any instance; hence, it is useless.

Definition 5.4. 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.

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 5.5. 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., $q' \neq \bot$), 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 $a_1, \ldots, a_k \in X$. (See Appendix A.2 for the algorithm to check whether a symbol is mandatory.)

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 5.6 formalizes the conditions for a state to be useful.

Theorem 5.6. 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.

Figure 10 shows an example of SA with some useless states. \( q_7 \) and \( q_8 \) are inaccessible. \( q_5 \) and \( q_6 \) are irrational states because they form a cycle of mandatory transitions. \( q_4 \) is also useless because it has a mandatory transition to the irrational state \( q_5 \). However, \( q_0 \) is not useless because its transition to \( q_4 \) on symbol C is not mandatory. \( q_9 \) is also useless because its only transition path from \( q_0 \) is blocked by useless states \( q_4, q_5, \) and \( q_6 \). Algorithm 5.7 (MakeUsefulSA) removes all useless states from a given SA and produces a useful and equivalent SA. Running MakeUsefulSA on Figure 10 produces the useful SA in Figure 7.

**Algorithm 5.7. 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 \( L \)
end for

Figure 10. Example of SA that contains useless states.
12: remove all transitions to \( q \neq q \) becomes 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 exists \( a_1, \ldots, a_n \in X \) such that \( \delta(q, a_1), \ldots, \delta(q, a_n) \in L \) do
16: modify \( H\text{Lang}(q) \) 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. The SA minimization process involves:
(1) transforming a given SA to a useful SA,
(2) partitioning the set of states into a set of equivalence classes of states,
(3) combining all states in each equivalence class into one state in the minimized SA,
and
(4) re-mapping the transitions accordingly

See Definition 5.8 and Theorem 5.9 on state equivalence.

Definition 5.8. 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 \( A \).

Theorem 5.9. Let \( A = (Q, X, q_0, \delta, H\text{Lang}; V\text{Dom}) \) be a useful SA and \( q, q' \in Q \) be two states of \( A \). \( q \) and \( q' \) are equivalent if and only if all of the following conditions hold.
(1) \( H\text{Lang}(q) = H\text{Lang}(q') \).
(2) \( V\text{Dom}(q) = V\text{Dom}(q') \).
(3) For each \( a \in X \), \( \delta(q, a) = \delta(q', a) = \bot \) or \( \delta(q, a) \) and \( \delta(q', a) \) are equivalent.

A minimal SA of a language is an SA with the fewest states among all SAs accepting the same language. See Definition 5.10. In fact, this minimal SA is the minimum (canonical) SA because it is unique up to isomorphism as stated in Theorem 5.13.

Definition 5.10. Let \( A \) be an SA. If there does not exist another SA \( A' \) such that \( L(A') = L(A) \) and \( |A'| < |A| \) then \( A \) is called a minimal SA of its language.

Theorem 5.11. 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 5.11 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 5.12 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 5.12. MinimizeSA

**Input:** useful SA $A = (Q, X, q_0, \delta, \text{HLang}; \text{VDom})$

**Output:** minimum SA $A' = (Q', X', q_0', \delta', \text{HLang}', \text{VDom}')$ equivalent to $A$

1: create a partition $P = \{B_1, \ldots, B_k\}$ of $Q$ such that for any two states $q_1, q_2 \in Q$, $\text{HLang}(q_1) = \text{HLang}(q_2)$ and $\text{VDom}(q_1) = \text{VDom}(q_2)$ if and only if $q_1$ and $q_2$ 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 $B \in Q_0$ do

15: set $\text{HLang}'(B)$ to $\text{HLang}(q)$ where $q \in B$

16: set $\text{VDom}'(B)$ to $\text{VDom}(q)$ 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 5.13 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 5.14 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 5.13. Let A and A’ be two equivalent SAs where A and A’ are minimal. A and A’ are isomorphic. In other words, the minimum SA of a language is unique up to isomorphism.

Algorithm 5.14. EquivalentSA

Input: SA A = (Q, X, q₀, δ, HLang, VDom)

Input: SA A’ = (Q’, X’, q₀’, δ’, HLang’, VDom’)

Output: true is returned if A ≡ A’; false is returned otherwise

1: MakeUsefulSA(A); MakeUsefulSA(A’)
2: MinimizeSA(A); MinimizeSA(A’)
3: create a list L that contains one tuple (q₀, q₀’)
4: mark q₀, q₀’ visited
5: while L is not empty do
6: pick (q, q’) from L and remove (q, q’) from L
7: if VDom(q) ≠ VDom(q’) or HLang(q) ≠ HLang(q) then
8: return false
9: end if
10: for all a ∈ X do
11: q₁ ← δ(q, a); q₁’ ← δ(q’, a)
12: if exactly one of q₁, q₁’ is ⊥ then
13: return false
14: else if both q₁, q₁’ are not ⊥ then
15: if exactly one of q₁, q₁’ is visited then
16: return false
17: else if both q₁, q₁’ are not marked visited then
18: put (q₁, q₁’) to L
19: mark q₁, q₁’ visited
20: end if
21: end if
22: end for
23: end while
24: return true

SA 1 (Figure 8) can be minimized to SA 2 (Figure 9) 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 5.15. Let A and A’ be two SAs. If $\mathcal{L}(A) \subseteq \mathcal{L}(A')$, 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, q₀, δ, HLang, 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 $q_0' \in Q'$ and $q_i' \in X'$. 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 5.16 (SubschemaSA) performs this subschema testing.

**Algorithm 5.16.** SubschemaSA  
**Input:** $A = (Q, X, q_0, \delta, \text{HLang}, \text{VDom})$  
**Input:** $A' = (Q', X', q_0', \delta', \text{HLang}', \text{VDom}')$  
**Output:** true is returned if $A$ is a subschema of $A'$; false is returned otherwise

1. 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**  
5. pick $(q, q')$ from $L$ and remove $(q, q')$ from $L$  
6. if $\text{VDom}(q) \not\subseteq \text{VDom}'(q')$ **then**  
7. report VDom incompatibility  
8. **end if**  
9. if $\text{HLang}(q) \not\subseteq \text{HLang}'(q)$ **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 $q_1 \neq \bot$ **then**  
15. if $q_1' = \bot$ **then**  
16. report transition incompatibility  
17. else if $(q_1, q_1')$ is not marked 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 (Figure 11) 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 reduce the original schema by extracting a smaller subschema that contains only the needed elements to improve the schema processing performance. Given some SA A and a set of permissible symbols $X'$, Algorithm 5.17 (ExtractSubschema) computes another SA $A'$ such that $A'$ accepts the 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 5.17. ExtractSubschema
Input: $SA A = (Q, X, q_0, \delta, HLang; VDom)$
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)

Here, we look at an example that illustrates this subschema extraction process. Suppose SA 2 (Figure 9) is given and the permissible symbol set is given as the symbol set of SA 2 excluding <Product>, i.e., {<Quote>, <Order>, <Line>, <Qty>, <Desc>, <Price>}. The extracted subschema SA should accept all instances of the original SA except those with any <Product> d-edges. Now, we run ExtractSubschema. L is initialized to contain one transition. In the first iteration of the while loop, this transition is removed. Since this transition is mandatory, q₁ needs to be deleted so transition q₂ is put into L for future deletion. In the second iteration, q₂ is removed. Since this transition is also mandatory, q₂ needs to be deleted and transition q₀ is added to L. In the third iteration, q₀ is deleted. But this time, since this transition is not mandatory, no other transition needs to be deleted. The HLang of q₀ is modified from \( L(<\text{Quote}|<\text{Order}) \) to \( L(<\text{Quote}) \). Since the above transitions are deleted, states q₂, q₄, and q₈ become inaccessible and are removed during MakeUsefulSA. The extracted subschema SA is shown in Figure 11, which corresponds to XSD 3 (Listing 5).

6. COMPLEXITY ANALYSIS AND PERFORMANCE IMPROVEMENT

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 inside these algorithms are PTIME except the following two. They are: (1) testing whether one RE \( r₁ \) is equivalent to another RE \( r₂ \) (i.e., \( L(r₁) = L(r₂) \)) and (2) testing whether \( r₂ \) includes \( r₁ \) (i.e., \( L(r₁) \subseteq L(r₂) \)) are PSPACE-complete [Martens et al. 2004]. The RE equivalence test on HLangs is used in MinimizeSA and EquivalentSA, and the RE inclusion test on HLangs is used in SubschemaSA. This makes these algorithms PSPACE-complete in worst-case.

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 pruning technique by leveraging some common XSD usage patterns. First, most industry XSDs express \( \text{xs : complexType} \) content models (i.e., HLangs) in simple combinations of \( \text{xs : sequence} \) and \( \text{xs : choice} \) (i.e, REs). [Bex et al. 2004] suggested that 97% of XSDs expressed the content models in some simple forms of REs. Also, [Martens et al. 2004] 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 \( \text{xs : sequence} \) or \( \text{xs : choice} \) must be one yet the occurrence of each \( \text{xs : element} \) 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 prune many positive RE equivalence cases. Because of the above properties, we may speed up the RE equivalence / inclusion test as follows.

1. If two REs are literally equal then conclude two REs are equivalent;
2. else if the forms of REs are supported by the weak test then: do the weak test on the REs and report the result;
3. else do the full test and report the result.

One of our experiments has showed that the running time of the algorithm SubschemaSA using this filtering technique runs 14 times faster than that using only the full test.

7. EXPERIMENTS ON XML SCHEMA COMPUTATIONS

This section presents and 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 the following 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 Experiment 1: xCBL Compatibility Testing

The xCBL 3.5 website [xCBL.org 2000b] 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 the following four incompatibility errors:

1. xCBL 3.0 declares a root element named 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 SchemaSource is declared after ValidateAttributes in xCBL 3.5.
(3) Under complex type CatalogHeader, element CatalogProvider is declared with 
\text{minOccurs} = \text{"0"} \text{ in xCBL 3.0 but \text{minOccurs} = \text{"1"} in xCBL 3.5.}

(4) Under complex type SchemaCategory, element CategoryID is declared with 
\text{minOccurs = \text{"1"} in xCBL 3.5 but this element is not declared in xCBL 3.0.}

After we have fixed the above errors, the XSDs can pass the subschema test. We 
believe these were caused by human errors when updating xCBL 3.0 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. One of such instances is listed in Listing 8.

Listing 8. Example XSD 3.0 instance that does not confirm to xCBL 3.5 XSD

\begin{verbatim}
<Carrier xmlns="rrn:org.xcbl:schemas/xcbl/v3_0/xcbl30.xsd">
  <CarrierCoded>Other</CarrierCoded>
</Carrier>
\end{verbatim}

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

(1) \textbf{Full only:} It did not use any filtering technique and performed only the full test 
on every RE comparison.

(2) \textbf{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.

(3) \textbf{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.

\begin{table}
\centering
\caption{Performance of different filtering techniques for HLang RE tests.}
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{strategy} & \textbf{eq. tests} & \textbf{weak tests} & \textbf{full tests} & \textbf{time (ms)} \\
\hline
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 \\
\hline
\end{tabular}
\end{table}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{performance.png}
\caption{Performance of different filtering techniques for HLang RE tests.}
\end{figure}

The performance results are compared in Table IV, and Figure 12, where the numbers of equality tests (eq. tests), weak tests, and full tests are shown, together with the running time of Algorithm SubschemaSA. There were a total of 1,258 RE inclusion tests to execute. Using the weak+full strategy, our weak test could conclude 596 (47%) RE comparisons. Using the equality+weak+full strategy, the equality test could conclude 1,196 (95%) comparisons; the weak test could conclude 59 out of the remaining 62 tests; finally, only 3 full tests were needed. The performance gain of the equality+weak+full strategy is over 14 times relative to the full-only strategy.

7.2 Experiment 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, Invoice). These document types are grouped into different domains. For example, the quotation domain consists of RFQ and Quote. The ExtractSubschema program 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[Apache.org 2004] 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 V and Table VI, and plotted in Fig. 13 and 14. The number of document types (doctypes) in each domain is indicated in the

<table>
<thead>
<tr>
<th>XSD (docs)</th>
<th>enames</th>
<th>types</th>
<th>edecls</th>
<th>ctime (s)</th>
<th>rtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (42)</td>
<td>1,905</td>
<td>1,290 (100%)</td>
<td>3,728</td>
<td>29.1 (100%)</td>
<td>N/A</td>
</tr>
<tr>
<td>Invoice (8)</td>
<td>904</td>
<td>412 (32%)</td>
<td>1,154</td>
<td>14.1 (48%)</td>
<td>3.11</td>
</tr>
<tr>
<td>Order (6)</td>
<td>722</td>
<td>352 (27%)</td>
<td>910</td>
<td>13.2 (46%)</td>
<td>3.17</td>
</tr>
<tr>
<td>Quote (2)</td>
<td>621</td>
<td>299 (23%)</td>
<td>721</td>
<td>12.9 (44%)</td>
<td>3.01</td>
</tr>
<tr>
<td>Auction (4)</td>
<td>555</td>
<td>266 (21%)</td>
<td>646</td>
<td>12.6 (43%)</td>
<td>3.01</td>
</tr>
<tr>
<td>Catalog (1)</td>
<td>156</td>
<td>81 (6%)</td>
<td>190</td>
<td>9.6 (34%)</td>
<td>2.74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XSD (docs)</th>
<th>enames</th>
<th>types</th>
<th>edecls</th>
<th>ctime (s)</th>
<th>rtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (51)</td>
<td>2,263</td>
<td>1,476 (100%)</td>
<td>4,473</td>
<td>30.5 (100%)</td>
<td>N/A</td>
</tr>
<tr>
<td>Invoice (9)</td>
<td>1,018</td>
<td>460 (31%)</td>
<td>1,305</td>
<td>15.3 (50%)</td>
<td>3.28</td>
</tr>
<tr>
<td>Order (7)</td>
<td>820</td>
<td>384 (26%)</td>
<td>1,052</td>
<td>13.7 (45%)</td>
<td>3.18</td>
</tr>
<tr>
<td>Quote (2)</td>
<td>621</td>
<td>319 (22%)</td>
<td>786</td>
<td>12.7 (42%)</td>
<td>3.32</td>
</tr>
<tr>
<td>Auction (4)</td>
<td>612</td>
<td>291 (20%)</td>
<td>711</td>
<td>12.4 (40%)</td>
<td>3.25</td>
</tr>
<tr>
<td>Catalog (1)</td>
<td>189</td>
<td>91 (6%)</td>
<td>231</td>
<td>10.7 (35%)</td>
<td>2.95</td>
</tr>
</tbody>
</table>
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. If we use the number of data types (i.e., states) to measure the size of a schema, 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%. This has evidently demonstrated ExtractSubschema can effectively reduce the schema size and processing time. Overall, the experiment took around 3 seconds to run the ExtractSubschema program in each case.

8. CONCLUSIONS

In this paper, we have studied the data interoperability between two web services and its relationship to the compatibility between XML schemas. We have formalized the Schema Automaton (SA) and Data Tree models to represent W3C XML Schema Definitions (XSDs) and XML instances for performing various computations on XSDs. Based on SA, we have investigated into two problems: schema compatibility testing and subschema extraction. On the schema compatibility problem, we have developed the algorithms for schema minimization, schema equivalence testing, and subschema testing. For schema minimization, we have proposed the concept of the usefulness of SA states and have developed an algorithm to compute the minimum (canonical) SA.
The schema equivalence and subschema testing can be used to verify the data interoperability between two web services and the compatibility between two schema versions. On the subschema extraction problem, we have developed an algorithm to compute a subschema from a given schema to recognize only a given subset of the symbols. The subschema extraction is useful to reduce the size of a huge schema when an application only requires a small subschema that recognizes some but not all of the element names. Moreover, we have proposed some effective pruning mechanisms to speed up the PSPACE-complete regular expression tests required by the subschema equivalence and subschema testing.

We have conducted two experiments to verify the practicality and effectiveness of the above algorithms. We used the XSDs of xCBL 3.0 and xCBL 3.5 as the real datasets for the experiments. In the first experiment, we ran the subschema testing algorithm to check whether xCBL 3.5 was backward compatible with xCBL 3.0, i.e., whether xCBL 3.5 was a superschema of xCBL 3.0. Though the backward compatibility is claimed on the xCBL 3.5 website, the experimental result has showed that xCBL 3.5 is actually not a superschema of xCBL 3.0 and has refuted the claim. Moreover, the experiment has also demonstrated that the pruning mechanisms for the regular expression equality test are effective in performance improvement of the algorithm. In the second experiment, we ran the subschema extraction algorithm to get different subschemas from xCBL 3.0 and xCBL 3.5 XSDs for different applications.

The size of an extracted subschema was only 6-32% of the size of the original schema. The time required for XMLBeans to compile each subschema was reduced to 34-50% of the time required to compile the original schema.

Based on the SA model, we anticipate other schema computation techniques can be derived. 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 novel schema computational techniques can be applied to develop new web services design tools and runtime engines.

REFERENCES


LAKSHMANAN, L. S. AND SADRI, F. 2003. Xml interoperability. WebDB.


PAPAKONSTANTIPOULOU, Y. AND VIANUY, V. 2000. DTD Inference for Views of XML Data. PODS.


APPENDICES

A. ADDITIONAL ALGORITHMS

A.1 Algorithm for Schema Automaton to Validate Data Tree

Algorithm A.1. AcceptDT

Input: SA $A = (Q, X, q_0, \delta, HLang, VDom)$

Input: DT $T = (N, E, Y, n_0, CEdges, Val, Sym)$

Output: if $A$ accepts $T$ then true is returned; otherwise false is returned

Output: if $A$ accepts $T$ then a well-defined binding map $\text{Bind}[n]$ is computed

1: initialize a list $L$ to contain one tuple $(q_0, n_0)$

2: while $L$ is not empty do

3: pick $(q, n)$ from $L$ and remove $(q, n)$ from $L$

4: if $\text{Val}(n) \notin \text{VDom}(q)$ then
5: return false
6: end if
7: if CSeq(n) $\not\in$ HLang(q) then
8: return false
9: end if
10: set Bind[n] to q
11: for all $e = (n, n_{child})$ in the sequence CEdges(n) do
12: $q_{next} \leftarrow \delta(q, \text{Sym}(e))$
13: add $(q_{next}, n_{child})$ to L
14: end for
15: end while
16: return true

A.2 Mandatory Symbol Testing
When a symbol occurs in every string of a regular language, this symbol is said to be mandatory in the language. Checking whether a symbol is mandatory in a language specified by an RE can be efficiently done in PTIME. An RE can be represented by a tree of sub-REs. For example, RE $(A^+ | (AB)^+) + (B^+ | (BA)^+)^*$ can be represented by an RE tree as shown in Figure 15. (Sequence means concatenation while choice means alternation.) Algorithm A.2 tests whether a symbol is mandatory in the RE. It constructs a boolean expression tree from the RE tree. If the boolean expression is evaluated to be true, then the symbol is mandatory in the RE.

Algorithm A.2. IsSymbolMandatory
Input: regular expression tree $R$
Input: symbol $a$ to test whether it occurs in every string of the language represented by $R$
1: Replace each subtree of $R$ at a node with zero minimum multiplicity (e.g., Kleene star $*$) by false
2: Replace each remaining leaf of $R$ with symbol equal to $a$ by true and each remaining leaf with symbol not equal to $a$ by false
3: Replace each remaining sequence node of $R$ with or

---

Figure 15. RE tree of $(A^+ | (AB)^+) + (B^+ | (BA)^+)^*$. 
4: Replace each remaining choice node of $R$ with and
5: Evaluate the transformed boolean expression tree

The boolean expression tree shown in Figure 16 evaluates whether $A$ is mandatory in $((A+|(AB)+)+(B+|(BA)^*))+$ and the result is true. On the contrary, $B$ is not mandatory as evaluated in the boolean expression tree shown in Figure 17.

B. PROOFS

Theorem B.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.

Proof. If part: If $q$ is inaccessible then $q$ cannot be reached from $q_0$ and $q$ can never accept any d-node of any instance. Therefore, $q$ is useless. Suppose $q$ is accessible but irrational, and there is some DT $T$ such that some d-node $n$ of $T$ is being validated by $A$ against $q$. Since $q$ is irrational, there exists $q, q_2, ..., q_k \in Q$ and $a_1, ..., a_k$ such that a cycle of mandatory transitions $\overline{a_1} \rightarrow q_2 \rightarrow a_2 \rightarrow \cdots \rightarrow q_k \rightarrow a_k \rightarrow q$ is formed. Since $T$ is finite, $n$ cannot be accepted by $q$ because $q$ requires $n$ to have infinite descendants of d-edges, $a_1, a_2, ..., a_k$. Therefore, $q$ is useless. Suppose $q$ is useful and there is a path of mandatory transitions from $q$ to some irrational state $q'$. Let $T$ be an instance of $A$ such that some d-node $n$ of $T$ is validated against $q$. There exists another d-node $n'$ of $T$ at which $A$ follows the mandatory transitions to $q_1$. However, $q'$ requires $n'$ to have infinite descendants. This is a contradiction because $T$ cannot be an instance of $A$. Therefore, $q$ is useless. If every transition path from the initial state to some state $q$ passes through an irrational state, $q$ is behind some irrational state, i.e., $q$ cannot be reached from the initial state without passing through some useful state. Hence, $q$ cannot accept any d-node of any instance. Therefore, $q$ is useless.

Figure 16. Boolean expression tree to test if $A$ is mandatory.

Figure 17. Boolean expression tree to test if $B$ is mandatory.
Only-if part: Suppose (1) $q$ is accessible, (2) $q$ is rational, (3) there exists no path of mandatory transitions from $q$ to any irrational state, and (4) there is some transition path from $q_0$ to $q$ in $A$, which contains all useful states. Since $q$ is accessible via useful states, $q$ can be reached from $q_0$. Since $q$ is rational and there exists no path of mandatory transitions to any irrational state, $A$ must go from $q$ to some state $q_0$ such that $\epsilon \in \pi(q')$ along every possible path in a finite number of transitions. Thus, we can construct a finite DT accepted by $q$. Therefore, $q$ is useful.

Theorem B.2. Let $A = (Q, X, q_0, \delta, H\text{Lang}, V\text{Dom})$ be a useful SA and $q, q' \in Q$ be two states of $A$. $q$ and $q'$ are equivalent if and only if all of the following conditions hold.

1. $H\text{Lang}(q) = H\text{Lang}(q')$.
2. $V\text{Dom}(q) = V\text{Dom}(q')$.
3. For each $a \in X$, $\delta(q, a) = \delta(q', a) = \perp$ or $\delta(q, a)$ and $\delta(q', a)$ are equivalent.

Proof. If part: Since $A$ is useful, all its states are useful. Suppose there exist useful states $q, q' \in Q$ such that all of the above conditions hold. Condition 1 guarantees $q, q'$ accept the same set of symbol sequences of child d-edges. Condition 2 guarantees $q, q'$ accept the same set of values. Condition 3 guarantees that $q, q'$ share the same “future.” Therefore, $q$ and $q'$ accept the same set of DT subtrees; the two states are equivalent.

Only-if part: Suppose $q, q' \in Q$ are equivalent but some of the above conditions do not hold. Since $q, q'$ are useful, each of them accepts at least one DT subtree. For conditions (1) and (2), if $H\text{Lang}(q) \neq H\text{Lang}(q')$ or $V\text{Dom}(q) \neq V\text{Dom}(q')$ then there must exist some d-node $n$ of some instance $T = (N, E, Y, n_0, C\text{Edges}, \text{Val}, \text{Sym})$ such that $\text{Val}(n)$ or $\text{CSeq}(n)$ is accepted by exactly one (but not both) of $q$ or $q'$: $q$ and $q'$ cannot be equivalent. Moreover, if condition 3 does not hold, $q, q'$ has different future. $q, q'$ cannot be equivalent too.

Theorem B.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 $A$.

Proof. Since $A$ is a useful SA, every state of $A$ is useful and accepts some DT subtrees. Hence, every all states in each equivalence class of $A$ accepts some DT subtrees. For the sake of contradiction, suppose there is an SA $A'$ equivalent to $A$ where $A'$ has fewer states than the equivalence classes of $A$. By the Pigeon Hole Principle, there exists some state $q$ of $A$ such that no state of $A'$ accepts the same set of DT subtrees that are accepted by $q$. Since $A$ is useful, $q$ is useful and accepts some DT subtrees in some instances. We can find some instance $T$ of $A$, which is also an instance of $A'$ such that when $A$ is validating $T$, $q$ is validating a d-node $n$ of $T$. Now suppose $A'$ is validating $T$ and state $q'$ of $A'$ is validating $n$. We can modify the DT subtree at $n$ to a DT that is accepted by exactly one of $q$ and $q'$. Hence, $T$ can only be accepted by exactly one of $A$ and $A'$, which is a contradiction.

Theorem B.4. Let $A$ and $A'$ be two equivalent SAs where $A$ and $A'$ are minimal. $A$ and $A'$ are isomorphic. In other words, the minimum SA of a language is unique up
to isomorphism.

Proof. Let $A = (Q, X, q_0, \delta, \text{HLang}, \text{VDom})$ and $A' = (Q', X', q_0', \delta', \text{HLang}', \text{VDom}')$ be two useful, equivalent, and minimal SAs. By Theorem 5.11, both $A$ and $A'$ contain the same number of states, which equals the number of equivalence classes of states. We construct another SA $\bar{A} = (\bar{Q}, \bar{X}, q_0, \bar{\delta}, \text{HLang}, \text{VDom})$ as follows.

\[
\bar{Q} = Q \cup Q' \cup \{\bar{q}_0\}, \text{ where } \bar{q}_0 \notin Q \cup Q'.
\]
\[
\bar{X} = X \cup X' \cup \{a_0, a_0'\}, \text{ where } a_0, a_0' \notin X \cup X'.
\]
\[
\bar{\delta}(q, a) = \begin{cases} 
\delta(q, a) & \text{if } q \in Q \text{ and } a \in X, \\
\delta'(q, a) & \text{if } q \in Q' \text{ and } a \in X', \\
q_0 & \text{if } q = \bar{q}_0 \text{ and } a = a_0, \\
q_0' & \text{if } q = \bar{q}_0 \text{ and } a = a_0', \\
\bot & \text{if } q = \bar{q}_0 \text{ and } a \neq a_0 \text{ and } a \neq a_0'.
\end{cases}
\]
\[
\text{HLang}(q) = \begin{cases} 
\text{HLang}(q) & \text{if } q \in Q, \\
\text{HLang}'(q) & \text{if } q \in Q', \\
a_0a_0' & \text{if } q = \bar{q}_0.
\end{cases}
\]
\[
\text{VDom}(q) = \begin{cases} 
\text{VDom}(q) & \text{if } q \in Q, \\
\text{VDom}'(q) & \text{if } q \in Q', \\
\{\epsilon\} & \text{if } q = \bar{q}_0.
\end{cases}
\]

Intuitively, $\bar{A}$ is an SA combining $A$ and $A'$. $\bar{A}$ has one new initial state $\bar{q}_0$ with transitions to the initial states of $A$ and $A'$ on new symbols $a_0$ and $a_0'$, respectively. Let $N = |A| = |A'|$. The number of states of $\bar{A}$ is $2N + 1$. We know that the number of states of a minimum SA equivalent to $\bar{A}$ is $N + 1$. It is because we can construct this minimum SA with $N + 1$ states by redirecting the two transitions of $\bar{q}_0$ on $b$ and $b'$ to $q_0$ of $A$ only without using $A'$ as $A'$ is equivalent to $A$. Since $A$ is minimal, and $a_0$ and $a_0'$ are new symbols, we cannot minimize this SA anymore. Now we can minimize $\bar{A}$ by grouping equivalent states. We know there is no other state of $\bar{A}$ equivalent to $\bar{q}_0$ because $a_0$ and $a_0'$ are new symbols. By Theorem 5.11, since $A$ and $A'$ are minimal, each state in $A$ is not equivalent to any other state in the same SA. Therefore, the only way to group $2N + 1$ states to $N + 1$ equivalence classes is to pair up one state $q$ from $A$ and one state $q'$ from $A'$, where $q$ is equivalent to $q'$; while $\bar{q}_0$ forms a distinct equivalence class on its own. By Theorem 5.9, $q$ and $q'$ have the same HLang and VDom, and share the same future for each pair of $q, q'$. Therefore, $A$ and $A'$ are isomorphic.
Thomas Y. Lee is a researcher in The University of Hong Kong (HKU). With 16 years of industry and academic experience, Dr Lee specializes in e-commerce, semi-structured database and semantic web. He was the Chief Technology Officer of the Center for E-Commerce Infrastructure Development (CECID). Under his leadership, CECID developed the first open-source ebXML messaging handler namely Hermes B2B gateway, which was deployed in many governments and enterprises worldwide. Moreover, he actively participates in various e-commerce standardization initiatives, e.g., OASIS Universal Business Language and ebXML Asia. Dr Lee received his Ph.D., M.B.A., M.Sc. and B.Eng, degrees from HKU. He was also an honoree of the Hong Kong Computer Society Outstanding IT Achiever Awards in IT Research.

David W. Cheung received the M.Sc. and Ph.D. degrees in computer science from Simon Fraser University, Canada, in 1985 and 1989, respectively. He is the Head and Professor of the Department of Computer Science in The University of Hong Kong (HKU). His research interests include database, data mining and e-commerce. Dr. Cheung was the Program Committee Chairman of PAKDD 2001, Program Co-Chair of PAKDD 2005, Conference Chair of PAKDD 2007, and Conference Co-Chair of CIKM 2009. He received the Outstanding Researcher Award from HKU on 1999. Dr. Cheung is a member of the ACM and the IEEE Computer Society.