Inferring XML Schema Definitions from XML Data

Geert Jan Bex
geertjan.bex@uhasselt.be

Frank Neven
frank.neven@uhasselt.be

Stijn Vansummeren*
stijn.vansummeren@uhasselt.be

Hasselt University and transnational University of Limburg, Belgium

ABSTRACT

Although the presence of a schema enables many optimizations for operations on XML documents, recent studies have shown that many XML documents in practice either do not refer to a schema, or refer to a syntactically incorrect one. It is therefore of utmost importance to provide tools and techniques that can automatically generate schemas from sets of sample documents. While previous work in this area has mostly focused on the inference of Document Type Definitions (DTDs for short), we will consider the inference of XML Schema Definitions (XSDs for short) – the increasingly popular schema formalism that is turning DTDs obsolete. In contrast to DTDs where the content model of an element depends only on the element's name, the content model in an XSD can also depend on the context in which the element is used. Hence, while the inference of DTDs basically reduces to the inference of regular expressions from sets of sample strings, the inference of XSDs also entails identifying from a corpus of sample documents the contexts in which elements bear different content models. Since a seminal result by Gold implies that no inference algorithm can learn the complete class of XSDs from positive examples only, we focus on a class of XSDs that captures most XSDs occurring in practice. For this class, we provide a theoretically complete algorithm that always infers the correct XSD when a sufficiently large corpus of XML documents is available. In addition, we present a variant of this algorithm that works well on real-world (and therefore incomplete) data sets.

1. INTRODUCTION

The advantages offered by the presence of a schema accompanying a collection of XML documents are numerous. Indeed, the presence of a schema facilitates automation and optimization of search, integration, translation and processing of XML data (cf., e.g., [14, 19, 23, 29, 6, 18, 30, 31, 41, 47]). Various software development tools such as Castor [1] and SUN's JAXB [3] rely on schemas as well to perform object-relational mappings for persistence. Additionally, the existence of schemas is imperative when integrating (meta) data through schema matching [44] and in the area of generic model management [7, 34]. Unfortunately, schemas are scarce and faulty in practice. For instance, Barbosa et al. [5, 35] have shown that approximately half of the XML documents available on the web do not refer to a schema. Furthermore, Bex et al. [8, 10] have noted that about two-thirds of the XML Schema Definitions (XSDs) gathered from schema repositories and from the web are not valid with respect to the W3C XML Schema specification [46], rendering them essentially useless for immedidate application.

Based on the above described benefits of schemas and their unavailability in practice, it is essential to devise algorithms that can infer a schema for a given collection of XML documents when none, or no syntactic correct one, is present. The latter problem is also acknowledged by Florescu [22] who emphasizes that in the context of data integration "*We need to extract good-quality schemas automatically from existing data and perform incremental maintenance of the generated schemas*". In this paper, we address the problem to automatically infer a concise XML Schema Definition describing a given set of XML Documents.

Previous research on schema inference for XML data has mainly focused on the inference of Document Type Definitions (DTDs) [9, 25, 15]. To appreciate the difference between inferring DTDs and XSDs, consider the XML document in Figure 1 that contains information about store orders and stock contents. Orders hold customer information and list the items ordered, with each item stating its id and price. The stock contents consists of the list of items in stock, with each item stating its id, the quantity in stock, and – depending on whether the item is atomic or composed from other items – some supplier information or the items of which they are composed, respectively. It is important to emphasize that order items do not include supplier information, nor do they mention other items. Moreover, stock items do not mention prices.

DTDs are incapable of distinguishing between order items and stock items because the content model of an element can only depend on the element's name in a DTD, and not on the context in which it is used [37]. For example, although the DTD in Figure 2 describes all intended XML documents, it also allows supplier information to occur in order items and price information to occur in stock items.

XML Schema, in contrast, is based on type definitions,

*Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

VLDB '07, September 23-28, 2007, Vienna, Austria.

```
<store>
  <order>
    <customer>
       <name>John Mitchell</name>
       <email> j.mitchell@yahoo.com </email>
    </customer>
    <item> <id> I18F </id>
          <price> 100 </price>
    </item>
    <item> ... </item> ... <item> ... </item>
  </order>
  <order> ... </order> ... <order> ... </order>
  <stock>
    <item>
      <id> IG8 </id> <qty> 10 </qty>
      <supplier> <name> Al Jones </name>
                 <email> a.j@gmail.com </email>
                 <email> a.j@dot.com </email>
       </supplier>
     </item>
     <item>
       <id> J38H </id> <qty> 30 </qty>
       <item>
          <id> J38H1 </id> <qty> 10 </qty>
          <supplier> ... </supplier>
       </item>
       <item>
          <id> J38H2 </id> <qty> 1 </qty>
          <supplier> ... </supplier>
      </item>
      <item> ... </item> ... <item> ... </item>
   </item>
   ...
   <item> ... </item>
 </stock>
</store>
```

Figure 1: Example XML document.

and therefore does allow the content model of an element to depend on the context in which it is used [32, 37]. For instance, it can be specified that an item is an order item when it occurs under an order element. In particular, XML schema can *exactly* describe the set of all intended documents, as we will show in Section 3. It is precisely this ability that makes inferring XSDs significantly more difficult than inferring DTDs. Indeed, whereas DTD inference basically reduces to the generation of regular expressions from sets of sample strings [9], inferring XSDs also entails identifying from a corpus of XML document the contexts in which elements bear different content models. Existing DTD inference engines do not identify such contexts and therefore always returns schemas like the one in Figure 2 that are too general with regard to the target schema.

To the best of our knowledge, we are the first to consider XSD inference in the presence of contexts identification. Indeed, although approaches like [16, 28] do infer schemas in XSD syntax, the obtained schemas are always structurally equivalent to DTDs.

Setting. It is unfortunate that the class of all XSDs cannot be learned from positive examples only. Indeed, a seminal result by Gold [26] implies that no matter how many example documents from a target XSD D are provided, no

```
<!ELEMENT store (order*, stock)>
<!ELEMENT order (customer, item+)>
<!ELEMENT customer (first, last, email*)>
<!ELEMENT item (id, price + (qty, (supplier + item+)))>
<!ELEMENT stock (item*)>
<!ELEMENT supplier (first, last, email*)>
```

Figure 2: A DTD describing the document in Figure 1.

algorithm exists that will always retrieve D given only the examples. As the framework for XML schema inference is exactly such that only positive example documents are provided, it is unrealistic to develop inference algorithms for the class of all XSDs. One of the main challenges therefore is to identify subclasses of XSDs that are widely used in practice and that can be learned efficiently from positive data only.

An examination of 225 XSDs gathered from the Cover Pages [17] (including many high-quality XML standards) as well as from the web at large, reveals that in more than 98% of the XSDs occurring in practice the content model of an element depends only on the label of the element itself, the label of its parent, and (sometimes) the label of its grandparent [32]. In Figure 1, for example, an item is an order item only if it is occurs in an order element. It is a stock item only if it occurs in a stock element or in an item element. We say that an XSD is *k-local* if its content models depend only on labels up to the k-th ancestor.

Although the class of k-local XSDs by itself is still too general to be learned from positive examples only, we show in this paper that the class of local XSDs with content models given by regular expressions in which each element name occurs at most once, *can* be learned efficiently from positive data only. The restriction to such *single occurrence regular expressions* (SOREs for short) is motivated by the same examination as above, which reveals that more than 99% of XSDs in practice consist solely of SOREs [32]. Furthermore, as shown in our earlier work [9], SOREs can be learned efficiently from positive data only, in contrast to the class of all regular expressions. Also, SOREs have the added benefit of succinctness: since every element name can occur only once, the size of a SORE is always linear in the number of different element names occurring in the corpus. The inferred content models are therefore naturally comprehensible.

Contributions. The contributions of this paper can be summarized as follows:

- We present a theoretically complete inference algorithm *i*Local that can infer any k-local and single occurrence target XSD from a corpus of XML documents, provided that this corpus is 'sufficiently large', a notion formally defined in Section 4. Furthermore, we show how standard tree automata techniques can be applied to minimize the derived schema's [33], an important post-processing step for obtaining comprehensible schema's. This minimization amounts to unification of unique types. That is, two types can be combined when they define the same set of XML fragments.

- Real world data is scarce and incomplete. Application of *i*Local on such data results in overly specific and

therefore meaningless schema's. Furthermore, minimization will only allow to combine types when they are equivalent. However, due to lack of data, types that actually represent the same context often have different inferred content models, prohibiting their unification. We therefore present a smoothing algorithm REDUCE that in addition to unifying equivalent types, also unifies types that are 'sufficiently similar' – a notion which is formally defined in Section 5. It is important to note that REDUCE is a general method, not specific to *i*LOCAL, that can be used as a post-processing step to smooth any XSD obtained through any method.

- We call *i*XSD the algorithm that first applies *i*LOCAL and then smoothens the obtained XSD by applying REDUCE. We illustrate the effectiveness of *i*XSD by means of a detailed experimental analysis on both real-world and synthetic data. First, we show that *i*XSD greatly outperforms *i*LOCAL on incomplete data. Further, we assess the high quality of the schema's inferred by *i*XSD . We show that *i*XSD generalizes well, is conservative for DTDs and gives an approximation for non-local XSDs. Finally, we show that runtime performance is acceptable for real world applications.

Outline. This article is further organized as follows. We discuss related work in Section 2 and provide the necessary background on XML Schema Definitions in Section 3. Our algorithms *i*LOCAL and REDUCE are given in Section 4 and 5, respectively. Their experimental validation is given in Section 6. Finally, we conclude in Section 7.

2. RELATED WORK

Schema inference. Schemas for semi-structured data have been defined in [12, 20, 42] and their inference has been addressed in [27, 40, 39]. The methods in [27, 40] focus on the derivation of a graph summary structure (called full representative object or dataguide) for a semi-structured database. This data structure contains all paths in the database. Approximations of this structure are considered by restricting to paths of a certain length. The latter then basically reduces to the derivation of an automaton from a set of bounded length strings. Naively restricting the algorithms to trees rather than graphs is inappropriate since no order is considered between the children of a node so that XSD-like schemas cannot be derived. However, even the use of more sophisticated encodings of the XML documents using edges between siblings would be to no avail since no algorithms are given to translate the obtained automata to regular expressions. In [39], a schema is a typing by means of a datalog program. The complexity of optimal schema inference is NP-hard. Again, no algorithms are given to transform datalog types into regular expressions. So, these approaches can therefore not be used to derive XSDs, not even when the semi-structured database is tree-shaped.

DTD inference. As explained in the Introduction, techniques for DTD inference cannot be applied directly to generate XSDs. Indeed, whereas DTD inference reduces to the derivation of a content model for each element name, in an XSD the content model depends on the context in which the element name appears. Several approaches for DTD inference have been proposed. XTract [25, 36] generates candidate regular expressions for each element name selecting the best one using the Minimum Description Length (MDL) principle. In contrast, the methods in [9] first generate a finite automaton for each element name which in a second step is rewritten into a concise regular expression. In [45], several approaches to generate probabilistic string automata representing regular expressions are proposed. Unfortunately, no methods are presented to transform these into corresponding regular expressions.

XSD inference. Trang [16] is state of the art software written by James Clark intended as a schema translator for the schema languages DTDs, Relax NG, and XML Schema. In addition, Trang allows to infer a schema for a given set of XML documents for either of the three schema languages. However, on a structural level (ignoring data values), the expressiveness of the generated schema's does not go beyond that of DTDs. So, when applying Trang on the data in Figure 1, it would derive the DTD of Figure 2 in XSD syntax, rather than the intended XSD in Figure 4. Similarly, XStruct [28] focuses on scalable derivation of schema's in XSD syntax (including data types) but does not incorporate expressiveness beyond DTDs. Another class of tools provided in JAXB [3] and the .NET [2] framework derive DTDs or XSDs from class files and vice versa,

Learning of tree automata. Fernau [21] proposed a general framework of function distinguishability to learn tree automata for ranked trees. In essence, the framework gives a learning algorithm for the settings where a function is given which determines how to merge states. As XSDs constitute a subset of the regular unranked tree languages [32], our approach can be seen as an instantiation of that framework generalized to unranked trees. In contrast to the ranked setting, our algorithm has to deal with regular expressions inference as well. Unranked tree language inference received recent interest in the context of wrapper induction [13]. However, this setting considers the inference of node-selecting queries in the presence of both positive and negative examples which makes the problem entirely different. The most related to our work is [43], where it is shown that unranked (m, n)-contextual tree languages can be learned from positive data only. Basically, an algorithm is presented that learns a tree language from the sets of (m, n)-forks appearing in sample trees. Here, an (m, n)-fork is a subtree of depth n and width m. The (m, n) contextual languages form a strict superset of local SOXSDs. It hence follows immediately that local SOXSDs can be learned from positive data only. Our Theorem 1 shows, however, that for local SOXSDs we do not need to resort to general (m, n)-forks but that already path-shaped forks suffice. In addition, the techniques of [43] are geared towards inferring queries, not XSDs. As such, no optimizations in the direction of schema inference have been attempted.

3. BACKGROUND

XML fragments. For our purposes, an *XML fragment* is a (possibly empty) sequence $\texttt{<}a_1\texttt{>}\, f_1\, \texttt{</}a_1\texttt{>} \ldots \texttt{<}a_n\texttt{>} f_n \texttt{</}a_n\texttt{>}$ of elements where $a_1, \ldots, a_n$ are *element names*, and $f_1, \ldots, f_n$ are themselves XML fragments. In particular, we ignore attributes (as these can straightforwardly be added) and data

```
<store>
  <order>
    <customer> <name/> <email/> </customer>
    <item> <id/> <qty/> <price/>  </item>
    <item> <id/> <qty/> <price/>  </item>
  </order>
  <order>
    <customer> <name/> <email/> <email/> </customer>
    <item> <id/> <qty/> <price/>  </item>
  </order>
  <stock>
    <item>
       <id/> <qty/>
       <supplier/> <name/> <email/> </supplier>
     </item>
  </stock>
</store>
```

Figure 3: A sample XML fragment for the XSD in Figure 4.

values (as the inference of atomic data types has already been studied [28]).

As usual we abbreviate <a> by <a/>. Furthermore, if f is an XML fragment, then we write paths(f) for the set of all labeled paths starting at a root element in f. For example, for the XML fragment in Figure 3, paths(f) includes the empty path λ, the path `store`, the path `store order`, the path `store stock`, the path `store order customer`, and so on. We write strings(f, p) for the set of all strings of element names occurring below an occurrence of path p in f. For example, for the XML fragment in Figure 3, strings(f, λ) = {`store`}, strings(f, `store`) = {`order order stock`}, and

$$\text{strings}(f, \texttt{store order}) = \{\texttt{customer item item}, \texttt{customer item}\}.$$

The first `order` element in Figure 3 ensures the presence of `customer item item`, while the second `order` element ensures the presence of `customer item`. For paths like `store order customer name` that end in a leaf of f, strings(f, p) always includes the empty string λ.

XML Schema Definitions. The W3C specification [46] essentially defines an XSD D to be a collection of *type definitions*, which, if we abstract away from the concrete XML representation of XSDs, are rules like

$$\mathit{store} \rightarrow \texttt{order}[\mathit{order}]^*, \texttt{stock}[\mathit{stock}] \qquad (\star)$$

that map type names to regular expressions over pairs $a[t]$ of element names a and type names t. Throughout the article we use the convention that element names are typeset in typewriter font, and type names are typeset in italic. Intuitively, this particular type definition specifies an XML fragment to be of type *store* if it is of the form

$$\texttt{<order>}\, f_1\, \texttt{</order>} \ldots \texttt{<order>}\, f_n \texttt{</order>}\ \texttt{<stock>}\, g\, \texttt{</stock>}$$

where $n \geq 0$; $f_1, \ldots, f_n$ are XML fragments of type *order*; and g is an XML fragment of type *stock*. Each type name that occurs on the right hand side of a type definition in an XSD must also be defined in the XSD, and each type name may be defined only once.

$$\begin{array}{lcl}
\mathit{root} & \rightarrow & \texttt{store}[\mathit{store}] \\
\mathit{store} & \rightarrow & \texttt{order}[\mathit{order}]^*, \texttt{stock}[\mathit{stock}] \\
\mathit{order} & \rightarrow & \texttt{customer}[\mathit{person}], \texttt{item}[\mathit{item}_1]^+ \\
\mathit{person} & \rightarrow & \texttt{name}[\mathit{emp}], \texttt{email}[\mathit{emp}]^+ \\
\mathit{item}_1 & \rightarrow & \texttt{id}[\mathit{emp}], \texttt{qty}[\mathit{emp}], \texttt{price}[\mathit{emp}] \\
\mathit{stock} & \rightarrow & \texttt{item}[\mathit{item}_2]^+ \\
\mathit{item}_2 & \rightarrow & \texttt{id}[\mathit{emp}], \texttt{qty}[\mathit{emp}], \\
 & & \quad (\texttt{supplier}[\mathit{person}] + \texttt{item}[\mathit{item}_2]^+) \\
\mathit{emp} & \rightarrow & \lambda
\end{array}$$

Figure 4: An XSD describing the XML document in Figure 1. The symbol λ denotes the empty string.

It is important to remark that the 'Element Declaration Consistent' constraint of the W3C specification [46] requires multiple occurrences of the same element name in a single type definition to occur with the same type. Hence, type definition ($\star$) is legal, but

$$\mathit{persons} \rightarrow (\texttt{person}[\mathit{male}] + \texttt{person}[\mathit{female}])^+$$

is not, as `person` occurs both with type *male* and type *female*. Of course, element names in *different* type definitions can occur with different types (which is exactly what yields the ability to let the content model of an element depend on its context). For example, Figure 4 shows a legal XSD describing the intended set of store document from the Introduction. Notice in particular the use of the types item_1 and item_2 to distinguish between order items and stock items.

Due to the 'Element Declaration Consistent' constraint, each element name a occurring in the type definition of a type t is associated with a unique type $\tau(t, a)$ in this type definition. For example, for the XSD in Figure 4 we have

$$\begin{array}{ll}
\tau(\mathit{root}, \texttt{store}) = \mathit{store}, & \tau(\mathit{store}, \texttt{order}) = \mathit{order}, \\
\tau(\mathit{store}, \texttt{stock}) = \mathit{stock}, & \tau(\mathit{order}, \texttt{item}) = \mathit{item}_1, \\
\tau(\mathit{item}_1, \texttt{id}) = \mathit{emp}, & \tau(\mathit{stock}, \texttt{item}) = \mathit{item}_2,
\end{array}$$

and so on. Then let $\rho(t)$ stand for the ordinary regular expression over element names only that we obtain by removing all types names in the definition of t. For example, for the XSD in Figure 4 we have

$$\begin{array}{ll}
\rho(\mathit{root}) = \texttt{store} & \rho(\mathit{store}) = \texttt{order}^*, \texttt{stock} \\
\rho(\mathit{order}) = \texttt{customer}, \texttt{item}^+ & \rho(\mathit{person}) = \texttt{name}, \texttt{email}^+
\end{array}$$

and so on. Then we can view an XSD D simply as a triple consisting only of (1) the set of types T being defined, (2) the mapping τ, and (3) the mapping ρ. Indeed, observe that the type definition of for example *order*,

$$\mathit{order} \rightarrow \texttt{customer}[\mathit{person}], \texttt{item}[\mathit{item}_1]^+$$

is easily obtained by replacing every element name a in the regular expression $\rho(\mathit{order}) = \texttt{customer}, \texttt{item}^+$ by the pair $a[\tau(\mathit{order}, a)]$. Since this view is more amenable to algorithmic manipulation, we will take it as the *definition* of an XSD, although for presentation purposes we will continue to represent XSDs as in Figure 4.

Definition 1. *An XSD is a triple $D = (T, \rho, \tau)$ consisting of a finite set of* types *T; a mapping ρ from T to regular*

expressions r as given by the syntax

$$r ::= \lambda \mid a \mid r, r \mid r + r \mid r^* \mid r^+ \mid r?$$

where λ denotes the empty string and a ranges over element names; and a mapping τ that assigns a type to each pair (t, a) with the element name a occurring in $\rho(t)$.

We remark that the W3C specification also requires regular expressions to be *deterministic* [46]. We do not go into details here, as the regular expressions for the classes of XSDs we will be inferring are deterministic by definition.

The semantics of an XSD is given by the following simple algorithm to validate an XML fragment $f = \texttt{<}a_1\texttt{>}f_1\texttt{</}a_1\texttt{>}\ldots\texttt{<}a_n\texttt{>}f_n\texttt{</}a_n\texttt{>}$ against a type t in an XSD $D = (T, \rho, \tau)$ [32, 37]. First, we check that the string of element names $a_1 \ldots a_n$ is matched by the regular expression $\rho(t)$. For example, when $t = \mathit{order}$ as defined in Figure 4, $a_1 \ldots a_n$ would be matched against $\texttt{customer}, \texttt{item}^+$. If this check fails, then the fragment is rejected. Otherwise, we validate each f_i against the type $\tau(t, a_i)$ of a_i in t, and accept the fragment if all these validations succeed. For example, when $t = \mathit{order}$ as defined in Figure 4 and $a_i = \texttt{item}$, f_i would be validated against $\tau(\mathit{order}, \texttt{item}) = \mathit{item}_1$. We write $\mathcal{F}(D, t)$ for the set of all XML fragments of type t in D.

Contextual power. The validation algorithm above actually implies that the content model of an element occurring in $f \in \mathcal{F}(D; t)$ is completely determined by the labeled path from the root to that element – a property of XSDs first noted by Martens et al. [32]. Indeed, for $f = \texttt{<}a_1\texttt{>}f_1\texttt{</}a_1\texttt{>}\ldots\texttt{<}a_n\texttt{>}f_n\texttt{</}a_n\texttt{>}$ to be of type t, each f_i must be valid w.r.t. $\tau(t, a_i)$. This is true only if $f_i = \texttt{<}b_1\texttt{>}g_1\texttt{</}b_1\texttt{>}\ldots\texttt{<}b_m\texttt{>}g_m\texttt{</}b_m\texttt{>}$ and every g_j is valid w.r.t. $\tau(\tau(t, a_i), b_j)$. We can continue this reasoning until we reach the desired element, where we see that its child fragment h must be of type $\tau(\ldots\tau(\tau(t, a_i), b_j)\ldots, c)$ with $a_i b_j \ldots c$ the labeled path from the root to the element. This leads us to the following alternative view on validation, which forms the cornerstone of our inference algorithms. Let, for a path $p = ab\ldots c$, $\tau(s, p) \to t$ denote that $\tau(\ldots\tau(\tau(s, a), b)\ldots, c)$ is defined and equals t. Let $L(r)$ denote the set of all strings matched by regular expression r.

Proposition 1. ([32]) *An XML fragment f has type s in an XSD (T, ρ, τ) iff for every path $p \in \text{paths}(f)$ there exists t such that $\tau(s, p) \to t$ and $\text{strings}(f, p) \subseteq L(\rho(t))$.*

Locality. The content model of an element in more than 98% of XSDs in practice turns out not to depend on the whole labeled path from the root to the element, but only on the k last element names in that path, with typically $k \leq 3$ [32]. The formal definition of such *k-local XSDs* is as follows. Let $p|_k$ stand for the path formed by the k last element names of a path p (if $\text{length}(p) \leq k$ then we take $p|_k = p$). Two paths p and q are *k-equivalent* if $p|_k = q|_k$. In particular, when $\text{length}(p) < k$, p is only k-equivalent to itself.

Definition 2. *A pair (D, s) with D an XSD and s a type in D is called* k-local *if for all k-equivalent p and q such that $\tau(s, p) \to t$ and $\tau(s, q) \to t'$ we have $t = t'$.*

For example, (D, root) with D as in Figure 4 is 2-local but not 1-local since $p = \texttt{store order item}$ and $q = \texttt{store stock item}$ are 1-equivalent, yet

$$\tau(\mathit{root}, p) \to \mathit{item}_1 \text{ and } \tau(\mathit{root}, q) \to \mathit{item}_2.$$

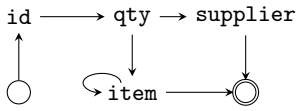

Figure 5: The SOA accepting the same language as the SORE $\texttt{id}, \texttt{qty}, (\texttt{supplier} + \texttt{item}^+)$.

Observe that the 1-local XSDs are in fact just DTDs.

Single occurrence. As already explained in the Introduction, a seminal result by Gold [26] implies that the class of k-local XSDs is still too large to be learned from positive examples only. Fortunately, the regular expressions in more than 99% of XSDs in practice are of a very specific form: each element name occurs at most once in them [32].

Definition 3. *A regular expression r is* single occurrence *if every element name occurs at most once in it. An XSD is single occurrence if it contains only single occurrence regular expressions.*

For instance, $\texttt{customer}, \texttt{item}^+$ and $(\texttt{school} + \texttt{institute})^+$ are both single occurrence, but $\texttt{id}, (\texttt{qty} + \texttt{id})$ is not as $\texttt{id}$ occurs twice. We abbreviate 'single occurrence regular expression' by SORE and 'single occurrence XSD' by SOXSD.

4. INFERENCE OF LOCAL SOXSDs

Our goal in this section is to infer a k-local SOXSD (D', t') equivalent to a target k-local SOXSD (D, t) given only a finite corpus of XML documents $\mathcal{C} \subseteq \mathcal{F}(D, t)$. This entails identifying from $\mathcal{C}$ the contexts (i.e., types) in which elements may bear different content models, as well as these content models (i.e., single occurrence regular expressions) themselves. Intuitively, we will use the paths occurring in $\mathcal{C}$ to identify types and the strings in $\mathcal{C}$ occurring below these paths to identify the SOREs. The latter essentially boils down to inferring a SORE from a set of sample strings, which can be done as follows [9].

Inferring SOREs. To infer a SORE from a set of sample strings S, we first learn from S a *single occurrence automaton* (SOA for short). A SOA is a specific kind of deterministic finite state automaton in which all states, except for the initial and final state, are element names. Figure 5 gives an example. Note that in contrast to the classical definition of automata, no edges are labeled: all incoming edges in a state a are assumed to be labeled by a. As such, a string $a_1, \ldots, a_n$ is accepted if there is an edge from the initial state to a_1, an edge from a_1 to a_2,..., and an edge from a_n to the final state. In other words, the SOA in Figure 5 accepts the same language as $\texttt{id}, \texttt{qty}, (\texttt{supplier} + \texttt{item}^+)$.

Definition 4. *Let* in *and* out *be two special symbols, distinct from the element names, that will serve as the initial and final state, respectively. A* single occurrence automaton *is a graph $A = (V, E)$ where all states in $V - \{\text{in}, \text{out}\}$ are element names, and $E \subseteq (V - \{\text{out}\}) \times (V - \{\text{in}\})$ is the edge relation.*

We write $L(A)$ for the set of all strings accepted by A. Algorithm 1, iSOA, due to Garcia and Vidal [24] then learns

Algorithm 1 iSOA

Input: a finite set of sample strings S
Output: a SOA A such that $S \subseteq L(A)$

1: Let V be the set of states consisting of all element names occurring in S plus the initial state in and final state out
2: Initialize $E := \emptyset$
3: **for** each string $a_1 \dots a_n$ in S **do**
4: add the edges $(\text{in}, a_1), (a_1, a_2), \dots, (a_n, \text{out})$ to E
5: Return $A = (V, E)$

a SOA A from a finite set of sample strings S. For instance, Figure 5 shows the result of running iSOA on $S = \{\texttt{id qty supplier}, \texttt{id qty item item}\}$.

It is clear that iSOA is sound: $S \subseteq L(i\text{SOA}(S))$ for each S. Moreover, iSOA is complete when S is sufficiently large, in the following sense.

Proposition 2. ([9]) *If r is a SORE and $S \subseteq L(r)$ is a set of sample strings that includes all strings in $L(r)$ of length at most $2n$ where n is the number of different element names occurring in r, then $L(i\text{SOA}(S)) = L(r)$.*

Having learned the SOA iSOA(S) for S, we can transform it into a SORE using the ToSore algorithm of [9].[1] This algorithm has the following desirable properties.

Proposition 3. ([9]) *If A is a SOA that is equivalent to a SORE r, then $L(\text{ToSore}(A)) = L(A)$. If A is not equivalent to a SORE, then $L(A) \subseteq L(\text{ToSore}(A))$.*

We refer to [9] for examples of SOAs that are not equivalent to a SORE. It immediately follows that if S contains all strings in $L(r)$ of length $2n$ where n is the number of different element names occurring in r, then ToSore$(i\text{SOA}(S))$ returns a SORE equivalent to r.

The algorithm. Inference of k-local SOXSDs can now be done as follows. Let paths$(\mathcal{C})$ stand for the set of all paths occurring in fragments in the corpus $\mathcal{C}$. Let k-strings$(\mathcal{C}, p|_k)$ stand for the set of all strings in $\mathcal{C}$ that occur below paths that are k-equivalent to p:

$$k\text{-strings}(\mathcal{C}, p|_k) := \bigcup\{\text{strings}(f, q) \mid f \in \mathcal{C}, q \in \text{paths}(f), p|_k = q|_k\}.$$

Algorithm 2, iLocal, then infers a k-local SOXSD from a finite corpus of XML fragments $\mathcal{C}$.

Let us illustrate iLocal's operation by running it on the corpus $\mathcal{C}$ consisting of the XML fragments of Figure 3 and 6, which both adhere to the target XSD in Figure 4. In line 1, iLocal constructs a type $p|_k$ for each path p in paths$(\mathcal{C})$. For $k = 2$, this yields the set of types shown in Figure 7. Next, iLocal constructs the content models for these types in lines 3 and 4. It does so by first learning a SOA for the set k-strings$(\mathcal{C}, p|_k)$ of all strings occurring in $\mathcal{C}$ below a path q that is k-equivalent to the type $p|_k$ under inspection, and subsequently transforming this SOA into a SORE. For $k = 2$ and $p|_k = \texttt{stock item}$, this set of strings

[1] Unfortunately, ToSore is called iDTD in [9], although its sole purpose is to transform SOAs into single occurrence regular expressions, not DTDs. We call it ToSore in this paper to avoid confusion.

Algorithm 2 iLocal

Input: a natural number k and corpus $\mathcal{C}$
Output: a k-local SOXSD (D, t) such that $\mathcal{C} \subseteq \mathcal{F}(D, t)$

1: Let the set of types T consist of all $p|_k$ with $p \in \text{paths}(\mathcal{C})$
2: Initialize the mappings ρ and τ to empty
3: **for** each type $p|_k$ in T **do**
4: add $p|_k \mapsto \text{ToSore}(i\text{SOA}(k\text{-strings}(\mathcal{C}, p|_k)))$ to ρ
5: **for** each path pa in paths(C) **do**
6: add $(p|_k, a) \mapsto (pa)|_k$ to τ
7: Return (D, t) with $D = (T, \rho, \tau)$ and $t = \lambda$

is $\{\texttt{id qty supplier}, \texttt{id qty item item}\}$ as the only path 2-equivalent to `stock item` in $\mathcal{C}$ is `store stock item`. Hence, for `stock item`, iLocal will first learn the SOA from Figure 5, which is subsequently transformed into the SORE $\texttt{id}, \texttt{qty}, (\texttt{supplier} + \texttt{item}^+)$. Note that iLocal hence correctly infers that stock items do not contain `price` elements. After termination of the for loop in line 3 we have hence inferred the content models for all types as shown in Figure 8.

Finally, in lines 5 and 6 iLocal determines the types associated with the element names in these content models. It does so by adding $(p|_k, a) \mapsto (pa)|_k$ to τ, for every element name a occurring in the content model of type $p|_a$. For $k = 2$ this yields, among others,

$$(\lambda, \texttt{store}) \mapsto \texttt{store},$$
$$(\texttt{store}, \texttt{stock}) \mapsto \texttt{store stock},$$
$$(\texttt{store stock}, \texttt{item}) \mapsto \texttt{stock item},$$
$$(\texttt{stock item}, \texttt{item}) \mapsto \texttt{item item},$$
$$(\texttt{item item}, \texttt{item}) \mapsto \texttt{item item}.$$

Note in particular the recursion introduced in the last rule.

A careful analysis shows that for this specific example corpus, iLocal has successfully inferred the target XSD D from Figure 4: $\mathcal{F}(D, \mathit{root}) = \mathcal{F}(i\text{Local}(2, \mathcal{C}))$. This is actually not a coincidence, as iLocal is *complete* on corpora that are "sufficiently large" in the following sense.

Definition 5. *Let D be an XSD, let t be a type in D, and let m be the number of types in D. A corpus $\mathcal{C}$ is called* k-complete *for (D, t) if (1) $\mathcal{C} \subseteq \mathcal{F}(D, t)$; (2) $\text{paths}(C)$ contains all paths p of length at most $m + k + 1$ such that $\tau(t, p) \to t'$ for some t'; and (3) for each such path, k-strings$(\mathcal{C}, p)$ contains all strings in $L(\rho(t'))$ of length at most $2n$, where n is the number of different element names occurring in $\rho(t')$.*

Theorem 1. *If (D, t) is a k-local SOXSD and corpus $\mathcal{C}$ is k-complete for (D, t), then $\mathcal{F}(i\text{Local}(k, \mathcal{C})) = \mathcal{F}(D, t)$.*

Using Proposition 1 it is not difficult to see that iLocal is always sound, even on incomplete corpora.

Proposition 4. *$\mathcal{C} \subseteq i\text{Local}(k, \mathcal{C})$, for every $\mathcal{C}$ and k.*

Proof. By Proposition 1 it suffices to show that for each XML fragment $f \in \mathcal{C}$ and each path $p \in \text{paths}(f)$ there exists a type t in $((T, \rho, \tau), \lambda) = i\text{Local}(k, \mathcal{C})$ such that $\tau(\lambda, p) \to t$ and $\text{strings}(f, p) \subseteq L(\rho(t))$. It is not difficult to see that it suffices to take $p|_k$ for t. Indeed, $\tau(\lambda, p) \to p|_k$ by construction, and

$$\begin{aligned}\text{strings}(f, p) &\subseteq k\text{-strings}(\mathcal{C}, p|_k)\\ &\subseteq L(\text{ToSore}(i\text{SOA}(k\text{-strings}(\mathcal{C}, p|_k))))\\ &= L(\rho(p|_k)),\end{aligned}$$

```
<store>
  <stock>
    <item>
       <id/> <qty/>
       <supplier/>
          <name/> <email/> <email/>
       </supplier>
     </item>
     <item>
       <id/> <qty/>
       <item>
          <id/> <qty/>
          <supplier> <name/> <email/> </supplier>
        </item>
        <item>
          <id/> <qty/>
          <item>
            <id/> <qty/>
            <supplier> <name/> <email/> </supplier>
          </item>
          <item>
            <id/> <qty/>
            <supplier> <name/> <email/> </supplier>
          </item>
       </item>
     </item>
  </stock>
</store>
```

Figure 6: Another sample XML fragment the XSD in Figure 4.

λ,	store,	store order,
order customer,	customer name,	customer email,
order item,	item id,	item qty,
item price,	store stock,	stock item,
item supplier,	supplier name,	supplier email,
item item		

Figure 7: The types inferred when running iLocal on the corpus consisting of the XML fragments in Figure 3 and 6, for $k = 2$.

by Propositions 2 and 3. Hence, $f \in \mathcal{F}(i\text{Local}(k, \mathcal{C}))$. □

Minimization. Although iLocal is complete on sufficiently large corpora, it has the disadvantage that the inferred XSDs may have more types than necessary. For instance, the inferred XSD of Figure 7 consists of 16 types, 8 types more than target XSD of Figure 4. In the worst case, $i\text{Local}(k, \mathcal{C})$ may return an XSD with $O(n^k)$ types where n is the number of different element names appearing in $\mathcal{C}$. For subsequent processing and presentation to the user, it is hence desirable to minimize the results of iLocal.

The algorithm Minimize due to Martens and Niehren [33] shown in Algorithm 3 minimizes an XSD D by unifying equivalent types in D. Here s is said to be *equivalent* to t if $\mathcal{F}(D, s) = \mathcal{F}(D, t)$.

Note that Minimize updates D during its execution. Lines 2 and 3 perform the actual unification of s and t by replacing t by s. The condition $t \neq r$ in line 1 ensures that the start type r is never removed. The equivalence condition $\mathcal{F}(D, s) = \mathcal{F}(D, t)$ in that line can be checked as follows.

$$\begin{aligned}
\lambda &\to \texttt{store}\\
\texttt{store} &\to \texttt{order}^*, \texttt{stock}\\
\texttt{store order} &\to \texttt{customer}, \texttt{item}^+\\
\texttt{order customer} &\to \texttt{name}, \texttt{email}^+\\
\texttt{order item} &\to \texttt{id}, \texttt{qty}, \texttt{price}\\
\texttt{store stock} &\to \texttt{item}^+\\
\texttt{stock item} &\to \texttt{id}, \texttt{qty}, (\texttt{supplier} + \texttt{item}^+)\\
\texttt{item supplier} &\to \texttt{name}, \texttt{email}^+\\
\texttt{item item} &\to \texttt{id}, \texttt{qty}, (\texttt{supplier} + \texttt{item}^+)
\end{aligned}$$

Figure 8: The content models inferred when running iLocal on the corpus consisting of the XML fragments in Figure 3 and 6, for $k = 2$. The types with empty content model λ have been omitted for space efficiency.

Algorithm 3 Minimize

Input: an XSD $D = (T, \rho, \tau)$ and type $r \in T$
Output: (D, r) with redundant types in D removed
1: **while** there are distinct types s and t in T with $t \neq r$ and $\mathcal{F}(D, s) = \mathcal{F}(D, t)$ **do**
2: replace each $(s', a) \mapsto t$ in τ by $(s', a) \mapsto s$
3: remove $t \mapsto \rho(t)$ from ρ and t from T

Definition 6. *For an XSD $D = (T, \rho, \tau)$, let* $\text{elems}_D(t)$ *denote the set of all element names a for which $\tau(t, a)$ is defined. The set* $\text{reach}_D(s, t)$ *of* pairs of types jointly reachable from (s, t) *is the least set containing (s, t) such that $(s', t') \in \text{reach}_D(s, t)$ and $a \in \text{elems}_D(s') \cap \text{elems}_D(t')$ implies that $(\tau(s', a), \tau(t', a)) \in \text{reach}_D(s, t)$.*

Clearly, $\text{reach}_D(s, t)$ can be computed by a standard fixpoint algorithm. Intuitively, $\text{reach}_D(s, t)$ is the set of all pairs (s', t') for which there exists a path p such that $\tau(s, p) \to s'$ and $\tau(t, p) \to t'$.

By the following proposition we can check that $\mathcal{F}(D, s) = \mathcal{F}(D, t)$ by computing $\text{reach}_D(s, t)$ and verifying that $\rho(s')$ and $\rho(t')$ match the same strings for every pair $(s', t') \in \text{reach}_D(s, t)$. The latter can be done in linear time for SOREs by converting $\rho(s')$ and $\rho(t')$ to deterministic finite automata (e.g. by the Glushkov construction [11]) and subsequently checking equivalence.

Proposition 5. *Let $D = (T, \rho, \tau)$ be an XSD and let s and t be two types in D. Then $\mathcal{F}(D, s) = \mathcal{F}(D, t)$ if, and only if, $L(\rho(s')) = L(\rho(t'))$ for all $(s', t') \in \text{reach}_D(s, t)$.*

5. PRACTICAL HEURISTICS

Unfortunately, when iLocal is run on an incomplete corpus, it will rarely happen that for distinct inferred types $p|_k$ and $q|_k$ that actually represent the same type in the target XSD we have $\mathcal{F}(D, p|_k) = \mathcal{F}(D, q|_k)$. For instance, if $k = 2$

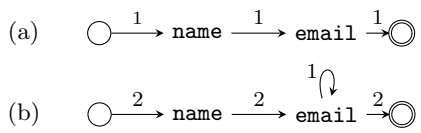

Figure 9: Inference of SOAs with support. SOA (a) is the result of iSOA({`name email`}). SOA (b) is the result of iSOA({`name email`, `name email email`}).

and $\mathcal{C}$ consists solely of the XML fragment in Figure 3, then

$$\text{2-strings}(\mathcal{C}, \texttt{order customer}) = \{\texttt{name email}, \texttt{name email email}\},$$

while 2-strings($\mathcal{C}$, `item supplier`) = {`name email`}. Hence, although `order customer` and `item supplier` both represent the type *person* in the target XSD of Figure 4, we will infer

$$\rho(\texttt{order customer}) = \texttt{name}, \texttt{email},$$
$$\rho(\texttt{item supplier}) = \texttt{name}, \texttt{email}^+.$$

Since `order customer` and `item supplier` are hence not equivalent, MINIMIZE will fail to unify them.

This illustrates that on incomplete corpora, iLOCAL risks identifying more types than that are present in the target XSD. In practice, therefore, we need a minimization algorithm that not only unifies equivalent types, but also unifies 'similar' types. Our goal in this section is to present such an algorithm, called REDUCE. Intuitively, REDUCE measures the similarity of two types s and t in an inferred XSD $(D, \lambda) = i\textsc{Local}(k, \mathcal{C})$ based on the SOAs learned for s and t. For all s and t that are similar enough, REDUCE subsequently adapts D such that $\mathcal{F}(D, s)$ and $\mathcal{F}(D, t)$ become equal. This adaption can be seen as generalizing the content models of s and t to compensate for missing data. Finally, the hence modified XSD D is minimized. Clearly, since all similar s and t have already been made equivalent, this causes all similar s and t to be unified.

Similarity. To define the notion of 'similarity' for types in an inferred XSD $(D, \lambda) = i\textsc{Local}(k, \mathcal{C})$ we first adapt iSOA such that for each edge (a, b) of the automaton A learned for a sample S we also keep the *support* $\text{supp}_A(a, b)$ of (a, b). This is the number of strings in S for which (a, b) needed to be added to the edges of A. Figure 9 gives an example.

Next, we adapt iLOCAL such that for each inferred type s we also keep the SOA soa(s) learned for s. That is, for $s = p|_k$,

$$\text{soa}(s) := i\text{SOA}(k\text{-strings}(\mathcal{C}, p|_k)).$$

Based on these extra data structures, we can define the similarity of two types s and t in an inferred XSD $(D, r) = i\textsc{Local}(k, \mathcal{C})$ as follows. Let dist(A, B) be the *normalized edit distance* between the support-annotated SOAs $A = (V, E)$ and $B = (W, F)$:

$$\text{dist}(A, B) := \frac{\sum_{(a,b)\in E-F} \text{supp}_A(a, b)}{\sum_{(a,b)\in E} \text{supp}_A(a, b)} + \frac{\sum_{(a,b)\in F-E} \text{supp}_B(a, b)}{\sum_{(a,b)\in F} \text{supp}_B(a, b)}.$$

Intuitively, dist(A, B) measures the *dissimilarity* of A and B by counting the number of edges present in A but not in B and the number of edges in B but not in A, weighted by the support these edges have in the original sample. For instance, for A the SOA in Figure 9(a) and B the SOA in Figure 9(b) we have $\text{dist}(A, B) = 0 + \frac{1}{7} = \frac{1}{7}$. The smaller the value of dist(A, B), the more similar A and B are. In particular, $L(A) = L(B)$ if dist(A, B) = 0.

The *edit distance* $\text{dist}_D(s, t)$ between the inferred types s and t is then defined as

$$\text{dist}_D(s, t) := \max_{(s',t')\in \text{reach}_D(s,t)} \text{dist}(\text{soa}(s'), \text{soa}(t')).$$

Again, the smaller $\text{dist}_D(s, t)$ is, the more similar s and t are. In particular, $\mathcal{F}(D, s) = \mathcal{F}(D, t)$ when $\text{dist}_D(s, t) = 0$, as $L(\rho(s')) = L(\rho(t'))$ for all $(s', t') \in \text{reach}_D(s, t)$ in that case.

Algorithm 4 REDUCE

Input: an inferred XSD $(D, r) = i\textsc{Local}(k, \mathcal{C})$ for some k and $\mathcal{C}$, and a similarity threshold ε
Output: (D, r) with similar types in D merged, and redundant types removed

```
1: let (T, ρ, τ) = D
2: initialize M := {(s,t) ∈ T² | 0 < dist_D(s,t) < ε}
3: while M is non-empty do
4:   for each (s,t) ∈ M do
5:     for each (s',t') ∈ reach_D(s,t) do
6:       set soa(s') := soa(s') ⊎ soa(t')
7:       set soa(t') := soa(s')
8:       for each a in elems_D(t') − elems_D(s') do
9:         add (s',a) ↦ τ(t',a) to τ
10:      for each a in elems_D(s') − elems_D(t') do
11:        add (t',a) ↦ τ(s',a) to τ
12:  recompute M := {(s,t) ∈ T² | 0 < dist_D(s,t) < ε}
13: for each type t in T do
14:   replace each t ↦ ρ(t) in ρ by t ↦ ToSore(soa(t))
15: Minimize(D, r)
```

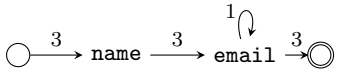

Figure 10: Adjunction of the SOA in Figure 9(a) with the SOA in Figure 9(b).

The algorithm. REDUCE then operates as shown in Algorithm 4: it merges types whose edit distance is less than some threshold parameter ε. Lines 4–10 are responsible for the actual merging of the selected types s and t, and ensure that $\mathcal{F}(D, s)$ becomes equal to $\mathcal{F}(D, t)$. In particular, the operation $\text{soa}(s') \uplus \text{soa}(t')$ in line 6 stands for the *adjunction* of soa(s') with soa(t'). This the SOA we obtain by adding to soa(s') all states and edges in soa(t') that are not in soa(s') and setting

$$\text{supp}_{\text{soa}(s')\uplus\text{soa}(t')}(a, b) := \text{supp}_{\text{soa}(s')}(a, b) + \text{supp}_{\text{soa}(t')}(a, b),$$

(where for simplicity we assume that $\text{supp}_{\text{soa}(s')}(a, b) = 0$ if (a, b) is not an edge in soa(s'), and similarly for $\text{supp}_{\text{soa}(t')}$). For instance, Figure 10 shows the adjunction of the SOAs in Figure 9(a) and Figure 9(b). Lines 13–14 converts the updated SOAs into SOREs. Finally, line 15 minimizes the resulting XSD.

6. EXPERIMENTAL EVALUATION

In this section, we validate our approach by means of an experimental analysis. Let iXSD be the composition of iLocal and Reduce, i.e., let

$$i\text{XSD}(k, \mathcal{C}) := \text{Reduce}(i\text{Local}(k, \mathcal{C})).$$

We first asses iXSD's precision in Section 6.2 by comparing inferred XSDs with their corresponding target XSDs. We next asses iXSD's sensitivity to its parameters (the context size k and the similarity threshold ε) in Section 6.3. We subsequently assess iXSD's capacity to generalize on corpora with only a limited amount of data in Section 6.4. We conclude in Section 6.5 with a short discussion of the runtime performance. Let us begin with discussing the corpora used in our experiments and their corresponding target XSDs.

6.1 The test corpora and their target XSDs

The first corpus we consider is $\mathcal{C}_{\text{XSD}}$, which itself consists purely of XSDs in XML syntax. Hence when run on this corpus, iXSD will attempt to infer the XSD for XML Schema Definition documents as it is defined in the W3C specification [46]. We assembled $\mathcal{C}_{\text{XSD}}$ from XSD documents found on the the Cover Pages [17], as well as from the web at large using the Google and Yahoo! search engines, bringing the total number of fragments in $\mathcal{C}_{\text{XSD}}$ to 697.

An analysis reveals that XSD for XML Schema Definition is 2-local, and is therefore a suitable target schema. The elements `attributeGroup`, `group`, and `extension` occur with different content models in two distinct contexts, `restriction` in three. The XSD for XML Schema Definitions contains a few more context dependent type definitions, but those differ only with respect to attributes, which we do not take into account here. All in all, this leaves us with 48 type definitions to infer. The total number of vertices in the corresponding SOAs is 202, while the number of edges totals 1024.

To gauge the precision of iXSD on XML documents that are described by a DTD, we reuse the real-world corpora mentioned in our previous work on DTD inference [9].

In the context of our work on DTD inference [9], we have already mentioned that very few corpora of XML documents exist with an interesting schema. In the present setting, this problem is aggravated by the fact that one requires an schema with at least some type definitions that depend on the context. Apart from $\mathcal{C}_{\text{XSD}}$, no suitable real-world corpus could be obtained. Hence we resorted to synthetic XSDs and XML corpora for additional experiments. We hand crafted 8 target XSDs to exhibit a specific set of features to test. All XSDs are recursive and hence define tree languages of unbounded depth while most have at least one content model that contains strings of unbounded length so that those tree languages have unbounded width.

In this set of hand crafted XSDs, many of the features that can make inference hard are present. In particular, all of these XSDs were constructed such that the content models of types in different contexts have the same alphabet as for the types b_1 and b_2 in Figure 11. Note that this particular XSD is 2-local. Our hand crafted set of XSDs, however, also contains XSDs that are 3-local but not 2-local. Moreover, one of the XSDs is 1-local, i.e., a DTD. Given that these XSDs had to be crafted by hand, they contain between 12 and 23 types. Multiple types are associated with at least two elements, while one grammar associates multiple types

$$\begin{array}{lcl}
\mathit{root} & \to & \mathtt{a}[a_1] \\
a_1 & \to & ((\mathtt{b}[b_1], \mathtt{c}[c]) + (\mathtt{d}[d], \mathtt{e}[e])), \mathtt{f}[f]^* \\
a_2 & \to & ((\mathtt{b}[b_2], \mathtt{d}[d]) + (\mathtt{e}[e], \mathtt{c}[c])), \mathtt{f}[f]? \\
b_1 & \to & (\mathtt{c}[c] + \mathtt{d}[d])^+ \\
b_2 & \to & \mathtt{c}[c], \mathtt{d}[d]? \\
c & \to & \lambda \\
d & \to & \mathtt{b}[b_2] \\
e & \to & \mathtt{b}[b_1] \\
f & \to & \mathtt{a}[a_2]
\end{array}$$

Figure 11: A hand crafted 2-local XSD.

with six elements. For each of the XSDs, a corpus of 200 XML documents was generated using ToxGene [4]. These corpora will be denoted by $\mathcal{C}_i$ for $1 \leq i \leq 8$.

6.2 Precision

In order to assess iXSD's precision we will compare, for each of the corpora $\mathcal{C}_{\text{XSD}}, \mathcal{C}_1, \ldots, \mathcal{C}_8$ described in Section 6.1, the inferred types with their corresponding types in the respective target XSDs. Here, "corresponding types" is defined as follows. Let (D_1, r_1) and (D_2, r_2) be the inferred and target XSD, respectively, and let $D_1 = (T_1, \rho_1, \tau_1)$ and $D_2 = (T_2, \rho_2, \tau_2)$. Then clearly, type r_1 in D_1 corresponds to type r_2 in D_2, as all fragments f valid w.r.t. r_1 in D_1 should also be valid w.r.t. r_2 in D_2 and vice versa. Now, we know that for each such particular $f =$ <a_1>f_1</a_1> ... <a_n>f_n</a_n>, each f_i should be valid with regard to $\tau_1(r_1, a_i)$ in D_1, and similarly should be valid with regard to $\tau_2(r_2, a_i)$ in D_2. Hence, in this sense, $\tau_1(r_1, a)$ corresponds to $\tau_2(r_2, a)$. Extending this reasoning further, we say that a type t_1 in D_1 corresponds to a type t_2 in D_2 if there exists a path p such that $\tau_1(r_1, p) \to t_1$ and $\tau_2(r_2, p) \to t_2$. We remark that it is straightforward to compute all pairs of corresponding types by inspecting τ_1 and τ_2.

Now observe that there are four ways in which iXSD may be imprecise.

1. There can be a type t in the target XSD that corresponds to no type in the inferred XSD. This happens only if the corpus does not contain fragments in which type t is used.

2. There can be a type t in the target XSD that corresponds to multiple types $s_1, \ldots, s_n$ in the inferred XSD. This happens when there are multiple distinct paths $p_1, \ldots, p_n$ such that $\tau_1(r_1, p_i) \to s_i$, whereas $\tau_2(r_2, p_i) \to t$ for all i. In this case, iXSD has failed to recognize that $s_1, \ldots, s_n$ actually represent the same type t and we call $s_1, \ldots, s_n$ *false positives* of t.

3. Conversely, there can be a type s in the inferred XSD that corresponds to multiple types $t_1, \ldots, t_m$ in the target XSD. This happens when there are distinct paths $p_1, \ldots, p_n$ such that $\tau_1(r_1, p_i) \to s$ for all i, whereas $\tau_2(r_2, p_i) \to t_i$. In this case, iXSD has falsely merged the types $t_1, \ldots, t_m$ into s, and we call s a *false negative* of $t_1, \ldots, t_n$.

4. Finally, the content models inferred for corresponding types s and t may differ.

No algorithm can hope to avoid imprecision (1) based solely on positive examples. We will therefore not consider this imprecision further.

False positives/negatives. As far as imprecisions (2) and (3) are concerned, *i*XSD produced neither false positives nor false negatives when run on $\mathcal{C}_{\text{XSD}}$ with $k = 2$. Hence the algorithm was successful in identify all and only the types in the target XSD for XML Schema Definitions. This is quite remarkable as we will detail in the next paragraph the high level of incompleteness of $\mathcal{C}_{\text{XSD}}$. The case $k = 3$ will be discussed in Section 6.3. The XSD inferred by *i*LOCAL alone contains 29 false positives, clearly illustrating the necessity and indeed the power of REDUCE.

On the synthetic corpora $\mathcal{C}_1, \ldots, \mathcal{C}_8$, *i*XSD performs excellently, reproducing the target XSD in each case.

Comparison of content models. In order to asses imprecision (4) we first note that, although the XSD corpus $\mathcal{C}_{\text{XSD}}$ is fairly large, it nevertheless is not exhaustive. In particular, across the content models r of all target types in the XSD for XML Schema Definitions actually used in $\mathcal{C}_{\text{XSD}}$, there are 511 edges (a, b) in the SOA corresponding to r for which no string in $\mathcal{C}_{\text{XSD}}$ would actually add the edge (a, b) when learning r by *i*SOA. This is a large amount compared to 1024, the total number of edges.

To obtain a fair assessment, we therefore first compute the *adapted content model* r'_t for each target type t in the XSD for XML Schema Definitions. This is the SOA we obtain by transforming the content model r of t into a SOA, and by subsequently removing all edges for which there is no subfragment of type t in $\mathcal{C}_{\text{XSD}}$ that would actually add the edge (a, b). The similarity between the inferred content model and r should be better than between r and r'_t. Here, *similarity* is simply the number of edges that are present in the first SOA but not in the second plus the number of edges present in the second but not in the first (not taking supports into account).

The experiment for $k = 2$ then shows that the content models of the derived XSD are at least as good (38 out of 47) or better than the baseline (9 out of 47), in five cases by more than 10 %. The fact that *i*XSD exceeds the baseline expectations is due to the combined effect of REDUCE's smoothening and TOSORE's generalization capacity.

6.3 Sensitivity to parameters

An important consideration is the sensitivity of the algorithm with respect to the choice of the parameters, the context size k and the similarity threshold ε.

A low number of false positives implies good generalization, while a low number of false negatives is a mark of precision. Ideally, neither should occur, but it is obvious that trying to minimize the number of false positives will cause an increase of the number of false negatives and vice versa. The parameters fine tune *i*XSD's performance. For increasing context size k, the number of false positives will increase, the same effect occurs for decreasing similarity threshold ε.

If the target schema of the corpus is a DTD, *i*XSD produces no false positives for $k = 2$. This is the case for the real-world corpora and as well as the synthetic corpus mentioned in Section 6.1.

For context size $k = 2$, the XSD derived from $\mathcal{C}_{\text{XSD}}$ has neither false positives, nor false negatives, which confirms the quality of the algorithm. For $k = 3$, 11 false positives crop up in the derived XSD. For example, as illustrated in Figure 12 for `restriction`, we have three types for $k = 2$, which are subsequently refined into false positives for $k = 3$.

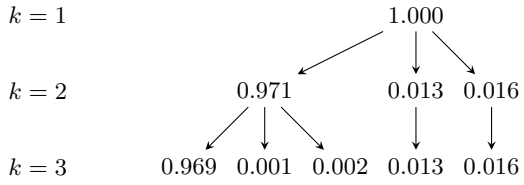

Figure 12: Distribution of the number of examples over types as a function of k for XSD's `restriction` element.

We note, however, that 2 of them can be identified as such since they are caused by 0.5 % or less of the examples that make up the corresponding type for $k = 2$ and hence are very unlikely. We consider this a good rule of thumb for the identification of false positives when the target schema is not known. So, *i*XSD can be run for increasing values of k until too large discrepancies are encountered.

The sensitivity of the algorithm with respect to the similarity threshold ε is illustrated in Table 1 which shows the number of false positives and false negatives as a function of ε. It is clear that the algorithm is not overly sensitive to its value: the target XSD is derived whenever $0.05 \lesssim \varepsilon \lesssim 0.15$. The results are similar for each of the synthetic corpora $\mathcal{C}_i$.

ε	false pos.	false neg.
0.01	2	0
0.05	0	0
0.10	0	0
0.15	0	0
0.20	0	3
0.50	0	3

Table 1: Sensitivity of the algorithm with respect to the similarity threshold ε for an 3-local XSD.

6.4 Generalization

In order to assess *i*XSD's robustness with respect to missing data, we split one of the corpora $\mathcal{C}_i$ into two parts: the first is used to derive an XSD, the second to validate that XSD. The *generalization* is the number of XML documents in the validation set that is valid with respect to the inferred XSD. In Figure 13 we show the generalization as a function of the training set size. Here the corpora for two of the synthetic XSDs were used with $k = 2$, the second of which (denoted by $\times$'s in Figure 13) has a content model containing a term of the form $(a_1 + \cdots + a_{12})^+$ which is quite hard to derive. It is clear from the plot that the algorithm performs well for even a relatively small number of XML documents as training set, 50 and 200 in this case.

6.5 Runtime performance

Although the Java code is in a prototype stage and hence not optimized for speed, the algorithm runs quite fast. The process of parsing the 697 XSDs of the W3C XML Schema Document corpus, a total of over 40 Mb, and the derivation of its XSD for $k = 2$ takes less than 15 seconds on an off the shelf laptop with a 1.73 GHz Pentium-M processor. Deriving a $k = 3$ XSD from the same set of data takes 17 seconds. Given that the number of distinct ancestor strings is 136 for $k = 2$ and 299 for $k = 3$, the algorithm's scales well with

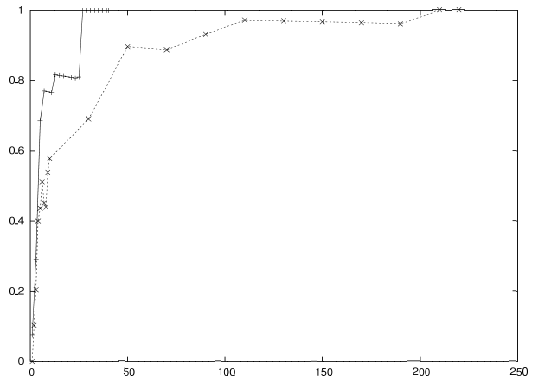

Figure 13: Generalization as function of the corpus size for two XSDs, one (×) with a large fan out.

the complexity of the target XSD. Even non-optimized, the algorithm can be comfortably used for real world applications.

7. CONCLUSIONS

We introduced two novel algorithms for the inference of concise XSDs. *i*Local is theoretically complete in the sense that it derives any target local SOXSD given enough data. A second algorithm, *i*XSD , is the algorithm *i*Local followed by a smoothening of the obtained XSD through Reduce to compensate for the lack of data. We have shown that *i*XSD performance is excellent on both real world and synthetic data. One of the main open issues in our framework is how to determine the best value of k. Although, we provide a rule of thumb which gave optimal results in our experiments, it would be worthwhile to look into machine learning techniques for parameter estimation. In future work, we want to extend our algorithms to larger classes of XSDs and regular expressions.

Acknowledgment

We thank Wouter Gelade for his comments on a first draft of this paper.

8. REFERENCES

[1] Castor. `www.castor.org`.

[2] Microsoft XML Schema Definition Tool. `http://msdn2.microsoft.com/en-us/library/x6c1kb0s(vs.71).aspx`.

[3] Sun JAXB. `java.sun.com/webservices/jaxb`.

[4] Denilson Barbosa, Alberto O. Mendelzon, John Keenleyside, and Kelly A. Lyons. ToXgene: an extensible template-based data generator for XML. In *WebDB*, pages 49–54, Madison, WI, 2002.

[5] Denilson Barbosa, Laurent Mignet, and Pierangelo Veltri. Studying the XML Web: gathering statistics from an XML sample. *World Wide Web*, 8(4):413–438, 2005.

[6] Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. In Chen Li, editor, *PODS*, pages 25–36. ACM, 2005.

[7] Philip A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *CIDR*, 2003.

[8] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. Expressiveness of XSDs: from practice to theory, there and back again. In *Proceedings of the 14th international World Wide Web Conference*, pages 712–721, Chiba, Japan, 2005.

[9] Geert Jan Bex, Frank Neven, Thomas Schwentick, and Karl Tuyls. Inference of concise DTDs from XML data. In *VLDB*. ACM, 2006.

[10] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs versus XML Schema: a practical study. In *Proceedings of WebDB 2004*, pages 79–84, 2004.

[11] Anne Brüggeman-Klein. Regular expressions into finite automata. *Theor. Comput. Sci.*, 120(2):197–213, 1993.

[12] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding structure to unstructured data. In Foto N. Afrati and Phokion G. Kolaitis, editors, *ICDT*, volume 1186 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 1997.

[13] Julien Carme, Aurélien Lemay, and Joachim Niehren. Learning node selecting tree transducer from completely annotated examples. In Georgios Paliouras and Yasubumi Sakakibara, editors, *ICGI*, volume 3264 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2004.

[14] Dunren Che, Karl Aberer, and M. Tamer Özsu. Query optimization in XML structured-document databases. *VLDB J.*, 15(3):263–289, 2006.

[15] Boris Chidlovskii. Schema extraction from XML: a grammatical inference approach. In Maurizio Lenzerini, Daniele Nardi, Werner Nutt, and Dan Suciu, editors, *KRDB*, volume 45 of *CEUR Workshop Proceedings*. Technical University of Aachen (RWTH), 2001.

[16] James Clark. Trang: Multi-format schema converter based on RELAX NG. `http://www.thaiopensource.com/relaxng/trang.html`.

[17] R. Cover. The Cover Pages. http://xml.coverpages.org/, 2003.

[18] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD Conference*, pages 431–442. ACM Press, 1999.

[19] Juliana Freire Fang Du, Sihem Amer-Yahia. ShreX: Managing XML Documents in Relational Databases. In Nascimento et al. [38], pages 1297–1300.

[20] M. F. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *International Conference on Data Engineering*, pages 14–23, 1998.

[21] Henning Fernau. Learning Tree Languages from Text. In Jyrki Kivinen and Robert H. Sloan, editors, *COLT*, volume 2375 of *Lecture Notes in Computer Science*, pages 153–168. Springer, 2002.

[22] Daniela Florescu. Managing semi-structured data. *ACM Queue*, 3(8), October 2005.

[23] Juliana Freire, Jayant R. Haritsa, Maya Ramanath,

Prasan Roy, and Jérôme Siméon. StatiX: making XML count. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *SIGMOD Conference*, pages 181–191. ACM, 2002.

[24] P. García and E. Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(9):920–925, 1990.

[25] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: learning document type descriptors from XML document collections. *Data mining and knowledge discovery*, 7:23–56, 2003.

[26] E.M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, May 1967.

[27] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *International Conference on Very Large Databases*, pages 436–445, 1997.

[28] Jan Hegewald, Felix Naumann, and Melanie Weis. XStruct: efficient schema extraction from multiple and large XML documents. In Roger S. Barga and Xiaofang Zhou, editors, *ICDE Workshops*, page 81. IEEE Computer Society, 2006.

[29] Carl-Christian Kanne and Guido Moerkotte. Efficient Storage of XML Data. In *ICDE*, page 198, 2000.

[30] Christoph Koch, Stefanie Scherzinger, Nicole Schweikardt, and Bernhard Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In Nascimento et al. [38], pages 228–239.

[31] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering XML Queries on Heterogeneous Data Sources. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB*, pages 241–250. Morgan Kaufmann, 2001.

[32] Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and Complexity of XML Schema. *ACM TODS*, 31(3), 2006.

[33] Wim Martens and Joachim Niehren. On the Minimization of XML Schemas and Tree Automata for Unranked Trees. *Journal of Computer and System Sciences*, 73(4):550–583, 2007.

[34] Sergey Melnik. *Generic Model Management: Concepts and Algorithms*. Ph.d. dissertation, springer lncs 2967, University of Leipzig, 2004.

[35] L. Mignet, D. Barbosa, and P. Veltri. The XML web: a first study. In *Proceedings of the Twelfth International World Wide Web Conference*, pages 500–510, Budapest, Hungary, 2003.

[36] Jun-Ki Min, Jae-Yong Ahn, and Chin-Wan Chung. Efficient extraction of schemas for XML documents. *Inf. Process. Lett.*, 85(1):7–12, 2003.

[37] M. Murata, D.Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. to be submitted to ACM TOIT, 2003.

[38] Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors. *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, Toronto, Canada, 2004. Morgan Kaufmann.

[39] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting Schema from Semistructured Data. In *International Conference on Management of Data*, pages 295–306. ACM Press, 1998.

[40] Svetlozar Nestorov, Jeffrey D. Ullman, Janet L. Wiener, and Sudarshan S. Chawathe. Representative objects: concise representations of semistructured, hierarchial data. In W. A. Gray and Per-Åke Larson, editors, *ICDE*, pages 79–90. IEEE Computer Society, 1997.

[41] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *Proceedings of the 9th International Conference on Database Theory (ICDT 2003)*, pages 315–329, 2003.

[42] Dallan Quass, Jennifer Widom, Roy Goldman, Kevin Haas, Qingshan Luo, Jason McHugh, Svetlozar Nestorov, Anand Rajaraman, Hugo Rivero, Serge Abiteboul, Jeffrey D. Ullman, and Janet L. Wiener. LORE: a Lightweight Object REpository for semistructured data. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, page 549. ACM Press, 1996.

[43] Stefan Raeymaekers, Maurice Bruynooghe, and Jan Van den Bussche. Learning (k, l)-contextual tree languages for information extraction. In João Gama, Rui Camacho, Pavel Brazdil, Alípio Jorge, and Luís Torgo, editors, *ECML*, volume 3720 of *Lecture Notes in Computer Science*, pages 305–316. Springer, 2005.

[44] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334–350, 2001.

[45] J. Sankey and R. K. Wong. Structural Inference for Semistructured Data. In *International Conference on Information and Knowledge Management*, pages 159–166. ACM Press, 2001.

[46] H.S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema part 1: structures*. W3C, May 2001.

[47] Guoren Wang, Mengchi Liu, Jeffrey Xu Yu, Bing Sun, Ge Yu, Jianhua Lv, and Hongjun Lu. Effective schema-based XML query optimization techniques. In *IDEAS*, pages 230–235, Hong Kong, China, 2003. IEEE Computer Society.