Enhancing Semistructured Data Mediators with Document Type Definitions*

Yannis Papakonstantinou, Pavel Velikhov
Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{yannis,pvelikho}@cs.ucsd.edu

Abstract

Mediation is an important application of XML. The MIX mediator uses Document Type Definitions (DTDs) to assist the user in query formulation and query processors in running queries more efficiently. We provide an algorithm for inferring the view DTD from the view definition and the source DTDs. We develop a metric of the quality of the inference algorithm's view DTD by formalizing the notions of soundness and tightness. Intuitively, tightness is similar to precision, i.e., it deteriorates when "many" objects described by the view DTD can never appear as content of the view. In addition we show that DTDs have some inherent deficiencies that prevent the development of tight DTDs. We propose "DTDs with specialization" as a way to resolve this problem.

1. Introduction

XML becomes the emerging standard for information exchange. Information mediation is expected to be one of XML's most important applications.

The MIX mediator project views XML as a database model (as opposed to a document model) and uses the mediator concept, as known in the DB area [Wie92, LIO96, PAGM96], to facilitate the implementation of the above applications. The MIX mediator provides to the user or to the application an XML view of the XML data exported by one or more applications or repositories. The mediator administrator customizes the view to the user needs, i.e., the view selects, consolidates, and ranks information according to the user's preferences. The views are customized using the mediator's query and view definition language, called XMAS (XML Matching And Structuring).

Because of the great similarity of XML with semistructured data [PGMW95, BDHS96a, QRS+95] we started with an architecture that is reminiscent of TSIMMS [PAGM96], a mediator for semistructured data. However, unlike OEM1 (which is the semistructured data model used by TSIMMS), and other semistructured data models XML data are typically accompanied by a Document Type Definition (DTD) which describes the content and the structure of the objects (aka, elements in XML terminology) participating in a document. In this paper we focus on valid XML documents, i.e., documents that always have a DTD.

DTDs are considered to be a kind of schema of a document. However they are more versatile with respect to how much structure they impose on the document. At the very structured extreme of the "structuredness" spectrum they may impose structure comparable to the rigid structure of relational data. At the other extreme they may allow any object type to contain any other object type. And in the middle of the spectrum they impose structures that are less restrictive and permit more variation in the data than conventional schemas do.

We briefly discuss the benefits realized by the use of DTDs in an on-demand mediator. The main technical contribution of this paper is the development of an algorithm required in order to compute the view DTDs (and hence realize many of the DTD benefits.) The algorithm works for a limited class of XML queries/views. Finally we introduce a framework for measuring the quality of view DTDs. We believe that this framework will be used in the future by works that will use more complex view definition and query languages.

To illustrate the gains obtained by the DTD use we

---

1 OEM stands for Object Exchange Model.
walk thru the operation of the TSIMMIS mediator first (recall, TSIMMIS does not use DTDs) and the MIX mediator, which does use DTDs, next.

The TSIMMIS mediator and the Disadvantages of Living Without Some Structure. Wrappers conceptually export the source data translated into the semi-structured model OEM. The mediator exports an integrated view of the wrapper data, based on a view definition, provided by the mediator administrator. The view definition is expressed in the Mediator Specification Language (MSL). During runtime the mediator receives queries, which refer to the view objects and are expressed in MSL. It first combines the incoming query and the view into a query which refers directly to the source data (and not to the views anymore.) Then the optimizer finds a plan for executing the latter query by sending queries (also expressed in MSL) to the wrappers and combining their results in the mediator. The wrappers translate the queries they receive into queries understood by the sources.  

What makes this process challenging (and often inefficient) is that MSL specifications can be very “loose” on the amount of information they provide about the structures they integrate. The ability to work with “loose” specifications is a valuable feature when dealing with dynamic semi-structured sources. As a contrived example, MSL allows the mediator administrator to create a view that unions the structures exported by 100 sites, without having any information about the contents and the structure of the data exported by these sites.

There are two weak points in the above scenario. First, the user does not know the structure of the underlying data and this impedes his efforts to formulate reasonable queries. This is a serious problem in environments with dynamic and unknown information. The second problem is that the mediator may not have complete (or even any) knowledge of the metadata and structure of each source. This results to a heavy loss of performance. DTDs provide a solution to the above problems as discussed next.

The MIX mediator and the Advantages of Living with DTD-provided Structure. The MIX mediator employs DTDs to assist the user in information discovery, query formulation and to allow the query processor to derive more efficient plans. In particular, given the source DTDs and the view, the View DTD Inference module derives the view DTD. (There are more than one view DTDs. We explain below which one is the “best”.). The view DTD is passed to the DTD-based query interface which displays the structure of the view elements and also provides fill-in windows and menus that allow the user to place conditions on the elements[BGL+].

Once a query is formulated, with or without using the DTD-Based Query Interface, it is passed to the query processor. Then the query simplifier may employ the source DTDs to create a more efficient plan. Finally, note that mediators can be stacked on top of mediators [Wie92]. In this case it is important that the lower level mediators can derive and provide their view DTDs to the higher level ones.

Contributions

1. We develop a view DTD inference algorithm (see Section 4) for a limited class of XMAS queries/views. Note that it is easy to compute a loose DTD for a view but the query interface and the query processor need the ones that describe the view as precisely as possible. These most “precise” DTDs are captured by our formal criterion which is outlined next.

2. We introduce and formalize tightness as the criterion for judging the precision of a view DTD (see Section 3.1). In particular, we say that a DTD $d_1$ is tighter than a DTD $d_2$ if every document described by $d_1$ is also described by $d_2$. Given a view and the source DTDs the view inference algorithm attempts to derive the tightest DTD that.

---

2Indeed, decomposing and translating queries is further complicated because the sources, and consequently the wrappers, have limited query processing capabilities. However, this issue is orthogonal to the topic of this paper and will not be discussed any further.

3Furthermore, the view DTD can have other applications as well, besides the ones we develop in the XML mediator. For example, it may be used by a toolkit for generating XSL style sheets for presentation of the view.
contains all the documents that may appear as content of the view. We believe that the tightness criterion can be a benchmark for other, more powerful, view definition languages and view inference algorithms.

3. We provide simple examples where, unfortunately, even the tightest DTD describes structures that can never appear as the view's content, i.e., even the tightest DTD is not "tight enough". The view DTD inference algorithm derives an extended form of DTDs that typically does not have non-tightness problems.

We start with a mathematical abstraction of the XML model and the XMAS query language. Section 3 discusses the properties of view DTDs. Section 4 describes the view inference algorithms. We conclude with related work and future directions.

2. Model and Query Language Framework

We present next a mathematical abstraction of XML and DTDs. Without loss of generality on our results on DTDs we focus on XML documents that meet the following requirements:

1. Always have DTDs, i.e., we focus on valid documents.

2. Do not have attributes other than the ID attribute. Consequently, we do not include attribute type declarations in the DTDs. Furthermore, we make the simplifying assumption that all elements will have an ID.

3. Do not have empty elements. Note that we still allow elements with empty content (which, confusingly enough, are not the same with empty elements [BPSM].)

4. Do not have mixed content elements, i.e., we do not capture elements whose content mixes strings with elements.

5. Neglect physical aspects of XML, i.e., entities.

   Given these assumptions XML is formalized as follows.

   Notice that the IDREF attributes are also excluded from our study. However, this exclusion does not significantly limit our DTD related results since the DTD does not type the target of an IDREF attribute.

Definition 2.1 [Element] An element e is a triplet consisting of a name, denoted as name(e), a unique ID attribute, and content, denoted as content(e). The content is either a sequence of elements or a PCDATA value, i.e., a character string.

Definition 2.2 [DTD] A DTD is a set \{\langle n : type(n) \rangle \}_{n \in N}\, where N is the set of names and type(n) is either a regular expression over N or PCDATA.

We denote by \(\mathcal{L}(r)\) the regular language described by r.

Definition 2.3 We say that an element e satisfies a DTD D, denoted as \(e \models D\), if the following hold:

1. name(e) \(\in N\), where N is the set of element names.

2. if content(e) = \(e_1 \ldots e_m\) then name(e_1) \(\ldots\) name(e_m) \(\in \mathcal{L}(\text{name}(e))\) and \(e_i \models D\), \(1 \leq i \leq m\).

3. else if content(e) is a string then type(name(e)) = PCDATA.

Definition 2.4 [Valid XML Document] A valid XML document consists of a DTD D, a document type \(d_\text{root}\), and a (most probably nested) element d such that \(d_\text{root} = \text{name}(d)\), i.e., \(d_\text{root}\) is the name of the root element of the document, \(e \models D\).

Remark 1 We have omitted listing "ANY" [BPSM] as another kind of type. However, ANY is merely a macro for the regular expression \(n_1 \ldots n_k\) where \(N = \{n_1, \ldots, n_k\}\).

Regular Expression Notation Staying in sync with the XML specification we use the following notations in regular expressions:

- \(r_1, r_2\) stands for the concatenation of \(r_1\) and \(r_2\).

- \(r_1 | r_2\) stands for the union (occasionally mentioned as disjunction) of \(r_1\) and \(r_2\).

- \(r^*\) stands for the Kleene closure of \(r\).

- \(r^+\) stands for \(r, r^*\).

- \(r^?\) stands for \(r|e\).
2.1. Query Language

The part of the query/view definition language XMAS we use in this paper is a subset of the recently proposed XML-QL [DFT]. All semi-structured query languages have the functionality described by this subset. The same language is used for defining queries and views. The only difference between a query and a view is that a mediated view is assigned a URL thru which it will be accessed by queries.

Our view inference algorithm works with pick-element XMAS queries, i.e., queries whose SELECT clause has a single variable, called pick-variable, that binds to elements and the WHERE clause consists of a single containment condition that is applied to exactly one source. The only form of negation we allow is the ability to say that the id’s of two elements are different. The elements that bind to the pick-variable are grouped into the view document whose name preceeds the SELECT clause. The order in which they appear is the same with the order in which they appear in the document when we traverse the elements of the document in a depth-first left-to-right order. We illustrate the semantics of the query language with the following example.

(Q1) withJournals =
   SELECT P
   WHERE <department><name>CS</name>
   P:*[professor | gradStudent]
   <publication id=Pub1><journal></journal></journal></journal></journal>
   </journal>
   AND Pub1 != Pub2

The variable P binds to all professor or gradStudent elements that
1. are contained in a department element,
2. the department contains an element name whose content is a string CS
3. P contains two different publication elements that contain journal subelements.

Note that we use the notation V : (element)…(element) instead of XML-QL’s equivalent (element)…(element) ELEMENT AS V

We could as well have variables in the content or element name position. For example, instead of professor | gradStudent we could have a variable N that can bind to any name. Our view inference algorithm works for pick-element queries where in the element name position we may have a constant, or a disjunction of constants or a variable that does not appear in other places in the condition. For simplicity we replace each element name variable with a disjunction of all names in the source DTDs at a preprocessing stage.

3. View DTD Inference

In this section we provide algorithms for inferring the DTD of a view from the source DTDs and the view definition. Before we proceed to describing the view inference algorithm of our system we provide two formal criteria against which view DTD inference algorithms should be evaluated. Furthermore, Section 3.2 shows in a formal way that inherent DTD weaknesses decrease the “precision” of view DTDs. Our specialized DTDs (Section 3.3) do not suffer from such non-tightness problems.

3.1. Soundness and Tightness

We believe that the view DTD must satisfy two properties. The first one, called soundness, guarantees that every view document will be described by the view DTD.

Definition 3.1 A view DTD DV is sound if, given source DTDs D1,D2,⋯,Dn and a view definition V, for every tuple (d1,...,dn) of n documents such that d1 |= D1,d2 |= D2,⋯,dn |= Dn the view document V(d1,...,dn) satisfies DV.

From now on we will refer to sound view DTDs simply as view DTDs.

The second property, called tightness, is motivated by the fact that view DTDs may describe document structures that cannot appear in a view (see Example 3.1). We suggest that the view DTD inference algorithm selects the tightest view DTDs, which, intuitively, are the ones that describe the “fewest” documents that cannot appear in a view. This intuition is formalized by the following definitions.

Definition 3.2 A DTD D is tighter than a DTD D′ if every document satisfying D satisfies D′.

Definition 3.3 A type (n: r) is tighter than a type (n: r′) if L(r) ⊆ L(r′), i.e., every sequence of elements described by r is also described by r′.

Definition 3.4 A DTD DV is a tightest view DTD for given source DTDs D1,D2,⋯,Dn and a view definition V if there is no view DTD D′V such that D′V is tighter than DV.

For the class of pick-element queries the view inference algorithm can “tightly” the view DTD in three ways. First it includes in the view DTD only the types
for the names that may appear in the view documents. 
Second, it tightens the types of the names as illustrated in Examples 3.1 and 3.2. Finally, the order and cardinality of the output elements is discovered as illustrated in Example 3.2.

**EXAMPLE 3.1** Consider the following subset of the department DTD and the query that retrieves professors or graduate students with at least two journal publications.

(D1) 
```
{<department : name, professor+, gradStudent+, course*>
  <professor : firstName, lastName, publication+, teaches>
  <gradStudent : firstName, lastName, publication+>
  <publication : title, author+, (journal|conference)>>
```

(Q2) withJournals = 
SELECT P
WHERE <department><name>CSC</name>
  P:<professor | gradStudent>
  <publication id=Pub1><journal></journal></journal>
  <publication id=Pub2><journal></journal></journal>
</journal> AND Pub1 != Pub2

A naive view inference algorithm may derive a view DTD by the following steps: First it adds the type definition

```
(withJournals : (professor|gradStudent)+)
```

in the DTD because P binds to elements named professor or gradStudent. Then it declares withJournals to be the document type, and eliminates all type definitions that correspond to names that are not referenced, directly or indirectly, by withJournals.

It is easy to see that such a DTD is not as tight as the following DTD (D2), which is actually the tightest DTD for the query (Q2) and the source DTD (D1). Notice that the professor and gradStudent types of DTD (D2) have been refined to reflect the constraint that the corresponding elements have at least two publications. Then the withJournals type of (D2) shows that professors appear before gradStudents.

(D2) 
```
{<withJournals : professor+, gradStudent+>
  <professor : firstName, lastName, publication+, teaches>
  <gradStudent : firstName, lastName, publication+>
  <publication : title, author+, (journal|conference)>>
```

The above example illustrated how a type can be refined by removing a "*" and forcing more than one instances of a name. Another very common case of refinement is *disjunction removal*, as illustrated by the following example.

**EXAMPLE 3.2** Consider the query (Q3) that operates on the source defined by DTD (D1) and collects all journal publications. It is clear that the disjunction (journal|conference) can be removed from the type definition of publication.

(Q3) publist = 
SELECT P
WHERE <department><name>CSC</name>
  P:<professor | gradStudent>
  <publication><journal>
</journal>/</journal>/</journal>

The view DTD is then:

(D3) 
```
{<publist : publication+>
  <publication : title, author+, journal)}
```

Notice that we could not remove the disjunction (journal|conference) from the DTD (D2) of Example 3.1 because the query retrieves many publications and, except for two of them, the other ones may be journals or proceedings. Hence we have to leave the definition of publication in the view DTD2 as is and lose the information that at least two publications of each professor/student in the view are journal publications. Such a loss of structural information is intrinsic in DTDs and is discussed next.

### 3.2. Structural Tightness

In many practical cases even the tightest view DTDs describe view document structures that cannot be produced by the view. For example, the DTD (D2) loses the information that at least two publications of each professor/student are in a journal. Consequently DTD (D2) describes documents with students having conference publications only - though it is clear from the view definition that a student with conference proceedings only can not appear in the view.

We formalize this information loss phenomenon by introducing the structural tightness property of view DTDs. We present the sources of structural non-tightness for the case of pick-element queries and provide the means to detect non-tightness of inferred DTDs.
Formalization of Structural Tightness Intuitively a view DTD is non-tight if it describes document "structures" that cannot be produced by the view. 5 First we formalize the notion of structural class. Intuitively, the structural class of a document excludes the string values of the document and thus abstracts its element name structure.

Definition 3.5 A structural class of documents is a set of documents such that for every two documents d1 and d2 in the class there is a mapping that maps
1. every string of d1 into a string of d2 and vice versa,
2. every ID of d1 into an ID of d2 and vice versa, and
3. if the mappings are applied to d1, d1 becomes identical to d2 and vice versa.

Definition 3.6 A structural class of documents satisfies a DTD D if the documents of the class satisfy D.

Notice that if one document of the class satisfies D then all documents of the class satisfy D. So in the above definition we could replace "the documents" with "a document".

Definition 3.7 Given a set of source DTDs D1, . . . , Dn and a view V, a DTD DV is structurally tight if
1. it is the tightest DTD of the view given the source DTDs,
2. for every structural class S that satisfies DV there is a view document I that satisfies S and there are also source documents I1, . . . , In, satisfying D1, . . . , Dn and I = V(I1, . . . , In).

Using the Definition 3.7 we characterize DTD (D2) as non-tight because there is a structural class, say the class S of withJournals documents that have one professor having no journal publications, that does not meet the second condition of the definition. In particular, S satisfies (D2), yet there is no possible valid source document I1 such that the view (Q2) when applied to I1 will result in a document that belongs to the structure S.

On the other hand DTD (D3) is tight according to Definition 3.7.

3.3. Specialized DTDs

Non-tightness reduces the "precision" of DTDs and also causes internal problems to our algorithms. To alleviate the non-tightness problems we developed the concept of specialized DTDs. Their important property is that there is a structurally tight specialized DTD for most views and source DTDs. Indeed, we conjecture that all pick element views without recursion have a structurally tight specialized view DTD. For views with recursive paths there are cases where there is no tight specialized DTD, simply because there is not even a tightest DTD (see remarks at the end of the section).

Definition 3.8 A specialized DTD (s-DTD) is a set
\[ \{ \langle n^i : type(n^i) \rangle \}_{n^i \in N^+} \]

where \( N^+ = \{ n^i | n^i \in N, i = 0, \ldots, spec(n) \} \) and spec(n) is a non-negative integer defined for all n ∈ N. The type is a regular expression over \( N^+ \) or it is PCDATA. The superscripts attached to the names are called tags and the regular expression type(\( n^i \)) is called a tagged regular expression.

We will need to convert the s-DTD to a regular DTD. For this purpose we define the image:

Definition 3.9 The image of a sequence \( \langle n^i_1 \ldots n^i_m \rangle \) of members of \( N^+ \) is the sequence \( \langle n_1 \ldots n_m \rangle \) of members of N (i.e., the image is the sequence after projecting out the superscripts.) Similarly the image of a tagged regular expression r is the regular expression r'th derived if we replace each name \( n^i \) of r with n.

EXAMPLE 3.3 For instance the image of the tagged type \( \langle title, author^4, author \rangle \) is just \( \langle title, author, author \rangle \).

Finally we need the ability to check whether an XML object satisfies the specialized DTD:

Definition 3.10 An element e satisfies an s-DTD D if the following hold
1. \( n \in N \), where \( n = \text{name}(e) \),
2. there is an i, 0 ≤ i ≤ spec(n) such that
   - if content(e) is a string then type(\( n^i \)) is PCDATA;
   - if content(e) = \( e_1 \ldots e_m \) then name(e1) . . . name(em) ∈ image(type(\( n^i \)))

5Requiring a tight view DTD to describe view documents exclusively is a property that cannot be achieved in any non-trivial case.
6Proving tightness for specific views and DTDs is beyond the scope of this paper.

7This condition stayed the same with plain DTDs
To avoid cluttering DTDs with the superscript notation from now on we assume that \( n \) is an acceptable shortcut for \( n^0 \).

To illustrate the use of specialized DTDs we show how the DTD from Example 3.1 can be turned into a tight specialized DTD.

**EXAMPLE 3.4** Recall that the problem with the view DTD for Query (Q2) was that every professor or a gradStudent retrieved was required to have two journal publications, but DTDs cannot represent such constraints. With specialized DTDs we create a new type \( \text{publication}^{(1)} \) that defines journal papers only. Then we require each professor/gradStudent to have exactly two \( \text{publication}^{(1)} \) objects and optionally other publications. The full specialized DTD is:

\[
\begin{align*}
\langle \text{withJournals : professor}, \text{gradStudent} \rangle & \\
\langle \text{professor} : & \text{firstName}, \text{lastName}, \text{publication}^{*}, \text{publication}^{1}, \text{publication}^{*}, \text{teaches} \rangle \\
\langle \text{gradStudent} : & \text{firstName}, \text{lastName}, \text{publication}^{*}, \text{publication}^{1}, \text{publication}^{*} \rangle \\
\langle \text{publication} : & \text{title}, \text{author}^{*}, \text{journal} \rangle \\
\langle \text{publication}^{1} : & \text{title}, \text{author}^{*}, \text{journal} \rangle
\end{align*}
\]

**3.4. Remarks on the Non-Existence of Tightest and Tight DTDs**

We have seen that even simple views often do not have tight view DTDs. Some views involving recursive paths may not have tight s-DTDs either. Indeed, such views may not have a tightest view DTD either.

**EXAMPLE 3.5** For example, consider the following DTD and the following pick-element with recursive paths query.

\[
\begin{align*}
\langle \text{section} : & \text{prolog}, \text{sections}^{*}, \text{conclusion} \rangle
\end{align*}
\]

\[
\begin{align*}
\langle \text{startsAndEnds} = & \text{SELECT X} \\
& \text{WHERE } \langle \text{section}^{*} \rangle.X: \langle \text{prolog} \mid \text{conclusion} \rangle
\end{align*}
\]

We can keep increasing the tightness of the “startsAndEnds" type but it is impossible to come up with the tightest type. For example, the type (T6) is less tight than (T7) which, in turn, is less tight than the (T8), etc. The non-existence of a tightest type is due to the fact that there is no regular expression that can recognize the language consisting of all sequences with equal numbers of "prologs" and "conclusions".

Although in general case obtaining the tightest DTD or a tight s-DTD is not possible, we conjecture that for pick-element queries without recursive path expressions there is always a tightest DTD as well as a tight s-DTD.

**4. Algorithms**

In this section we describe how a tight specialized DTD is computed for pick-element queries without recursive path conditions. First we show how to infer an s-DTD for the type of elements that bind to a pick-variable \( X \) in queries of the form:

\[
\text{(Q5) SELECT X WHERE } X: \text{true condition}
\]

In Section 4.1 we describe how individual types are refined. In Section 4.2 we present the algorithm for computing the type of the elements that bind to \( X \) as well as the types of the sub-elements. Finally in Section 4.4 we complete the presentation by describing the computation of the type of the view's top element.

**4.1. DTD Type Refinement**

The DTD tightening algorithm of Section 4.2 recursively "tightens" each type of the initial DTD (DTD of the source before the application of the query) by means of the type refinement algorithm. We first provide a type refinement definition and examples. We assume that no two conditions in the query have the same name.

**Definition 4.1** The type refinement \( \text{refine}(r, n) \) of a regular expression \( r \) given a name \( n \) is the regular expression \( r' \) that describes all strings of \( L(r) \) that contain at least one instance of \( n \).

The algorithm that computes \( \text{refine}(r, n) \) uses the special operators \( \ominus \) and \( \mid \) that extend the regular expression operators \( + \) and \( * \):

\[
\begin{align*}
\text{fail, if } r_1 = \text{fail or } r_2 = \text{fail} \\
r_1 \ominus r_2 = \begin{cases} r_1 \mid r_2, & \text{otherwise} \end{cases}
\end{align*}
\]
\[ r_1 || r_2 = \begin{cases} 
\text{fail}, \text{if } r_1 = \text{fail} \text{ and } r_2 = \text{fail}, \\
r_1, \text{if } r_1 \neq \text{fail} \text{ and } r_2 = \text{fail}, \\
r_2, \text{if } r_1 = \text{fail} \text{ and } r_2 \neq \text{fail}, \\
r_1 || r_2, \text{otherwise} 
\end{cases} \]

**Type refinement algorithm for conditions involving different names:**

function refine(r, n)
  if r = n then return n
  if r = n' where n' is a name and n' \neq n then return fail
  if r = r' then return refine(r', n) || fail
  if r = g* then return g* \otimes refine(g, n) \otimes g*
  if r = r_1, r_2 then return (refine(r_1, n) \otimes r_2) \mid (r_1 \otimes refine(r_2, n))
  if r = r_1 || r_2 then return refine(r_1, n || refine(r_2, n))

**EXAMPLE 4.1** Consider the DTD (D9) and the query (Q6)

\[(D9) \quad \{ (\text{professor} : \text{name}, (\text{journal} | \text{conference})*) \}\]

(Q6) answer =

```
SELECT X
WHERE X:<professor><journal></><</>
```

The tightening algorithm invokes the refinement algorithm above to enforce that the type definition of professor will make the existence of a journal necessary. The following steps illustrate how the algorithm decomposes the refinement of a sequence, of a loop, or of a disjunction into a composition of the refinements of the constituents of the sequence, the loop or the disjunction. Let us call name, journal, and conference by their first letter.

\[
\begin{align*}
\text{refine}(n, (j|c)*, j) &= \text{refine}(n, j) \otimes (j|c)* \mid n \otimes \text{refine}((j|c)*, j) \\
&= (\text{fail} \mid n, \text{refine}(j|c)*, j)) \\
&= n, (j|c)* \otimes \text{refine}(j, j) \otimes (j|c)* \\
&= n, (j|c)* \otimes (j \mid \text{fail}) \otimes (j|c)* \\
&= n, (j|c)*, j, (j|c)*
\end{align*}
\]

**Type Refinement When Conditions on Elements with the Same Name** When a tree condition requires the existence of two or more different elements with the same name the tightening algorithm has to work with specialized DTDs in order to derive the correct result. We extend below the type refinement definition to tagged regular expressions. (Recall, the type definitions of specialized DTDs are based on tagged regular expressions.)

**Definition 4.2** The type refinement refine(r, n_T) of a tagged regular expression r given a tagged name n_T is the tagged regular expression r' that describes all sequences s where

1. \( s \) is of the form \( s_1, n_T, s_2 \) and
2. the sequence \( \text{image}(s_1, n, \text{image}(s_2)) \) is a member of \( \mathcal{L}(r) \).

The algorithm for the refinement of tagged regular expressions differs from the algorithm of Section 4.1 in the base case (the first two lines)

function refine(r, n_T) \( T \neq 0 \)
  if r = n recall \( n \) is a shortcut for \( n^0 \)
  then return \( n_T \)
  if r = n_T where \( n_T \) is a tagged name and \( n_T \neq n \lor T' \neq 0 \lor T' \neq T \)
  then return fail
  the rest is the same with the algorithm of Section 4.1

**EXAMPLE 4.2** Consider again the DTD of Example 4.1 but now assume that the query requests the existence of two different journal publications.

(Q7) answer =

```
SELECT X
WHERE X:<professor> <journal id=J1><</>
      <journal id=J2><</>
AND J1 != J2
```

The tightening algorithm will tag the two instances of journal as \( j_1^0, j_2^0 \) and \( j_1^0, j_2^0 \). For brevity let us again use the first letters of the names. First it refines the type \( n, (j|c)* \) (recall, this is a shorthand for \( n^0, (j|c)^0* \) with \( j_1 \)) and the result is further refined with \( j_2^2 \).

\[
\begin{align*}
\text{refine}(n, (j|c)*, j_1) &= \text{refine}(n, (j|c)*, j_1^1) \mid (n \otimes \text{refine}((j|c)*, j_1^1)) \\
&= (\text{fail} \mid n, \text{refine}(j|c)*, j_1^1)) \\
&= n, (j|c)* \otimes (j \parallel \text{refine}(j, j_1^1)) \parallel (j|c)* \\
&= n, (j|c)*, j_1^1, (j|c)*
\end{align*}
\]

\[
\begin{align*}
\text{refine}(n, (j|c)*, j_2^2) &= \text{refine}(n, (j|c)*, j_2^2) \mid (n \otimes \text{refine}((j|c)*, j_2^2)) \\
&= (\text{fail} \mid n \otimes \text{refine}((j|c)*, j_2^2)) \mid (j|c)* \parallel (j|c)* \parallel (j|c)* \parallel (j|c)* \\
&= (n, (j|c)*, j_1^1, (j|c)*) \mid (n, (j|c)*, j_2^2, (j|c)*) \mid (n, (j|c)*, j_2^2, (j|c)*) \mid (n, (j|c)*, j_2^2, (j|c)*)
\end{align*}
\]

...
4.2. Tightening Algorithm

We discuss now how to combine the individual type refinements discussed in Section 4.1 into an algorithm that computes the \(s\)-DTD for queries of the form (Q5). The tightening algorithm (see Figure 2) starts with an empty \(s\)-DTD and adds refined types to it up by traversing the tree constraints and refining types of the original DTD. When two different tree constraints refine the same DTD type, we store the union of the content of the refinements. After the algorithm terminates we insert type definitions of types from the original DTD that occur in the content of the tightened \(s\)-DTD and were left unrefined. For simplicity, we assume that no two sibling conditions can bind to the same element.

Note that the tightening algorithm has a useful side effect. Given the tree condition \(c\) and the source DTD \(d\) it decides whether the condition is

- **valid**, i.e., \(c\) will be satisfied by every document that satisfies \(d\).
- **satisfiable**, i.e., \(c\) will be satisfied by some documents that satisfy \(d\).
- **unsatisfiable**, i.e., there is no document satisfying both \(c\) and \(d\). In this case the view DTD describes an empty answer.

4.3. Converting \(s\)-DTDs to DTDs

Once we have obtained a tightened \(s\)-DTD we may need to convert it into a regular DTD. The regular DTDs do not support tagged types, so we need to do the following: We first need to obtain the images of all types of the \(s\)-DTD (see Definition 3.9) and then to merge all images that have the same name. We also want to inform the user that a merging has occurred, since merging inadvertently introduces non-tightness.

The algorithm is given below:

Algorithm Merge

**INPUT:** an \(s\)-DTD \(d\)

**OUTPUT:** \(d'\) - the DTD in which the specialized types of \(d\) are merged

\[
d' \leftarrow \{ \}
\]

- for each type definition \(\langle n^T : \text{type}(n^T) \rangle\) of \(d\)
  - if \(d'\) contains the type definition \(\langle n : \text{type}(n) \rangle\)
    - replace \(\langle n : \text{type}(n) \rangle\) with \(\langle n : \text{type}(n) \rangle | \text{image}(\text{type}(n^T))\) signal the merge
  - else
    - insert in \(d'\) \(\langle n : \text{image}(\text{type}(n^T)) \rangle\)

Algorithm Tighten

**INPUT:** the DTD name \(r\)

\(e\), a tagged tree condition

\(d\), the DTD to be tightened

**OUTPUT:** \(d'\) - the tightened \(s\)-DTD that includes a refined type \(r\)

**METHOD:** run procedure Tighten\((r, \text{type}(r), c, d, \{\})\)

- pull all names that occur untagged in the resulting \(s\)-DTD

\[
\text{procedure Tighten}(n, t, c, d, d')
\]

- \(n\) is the name to be tightened
  - \((\text{may be root})\)
  - \(t\) is the type of \(n\)
  - \(c\) is a simple path or tree condition
  - \(d\) is the original DTD
  - \(d'\) is the current refined DTD

- if \(c\) is a list of conditions
  - \(d' \leftarrow \text{Tighten}(n, t, c_1, d, d')\)
  - for each condition \(c_i, i > 1\)
    - if \(\text{Tighten}(n, \text{dd}[n], c_i, d, d')\) does not fail
      - \(d' \leftarrow \text{Tighten}(n, \text{dd}[n], c_i, d, d')\)
    - else fail
  - return \(d'\)

- else \(c\) is a name \(c^T\) with tag \(T\), followed by a list of conditions \(\text{rest}\)

\[
d'[n^T] \leftarrow \text{refine}(t, c^T)
\]

- if the refinement included an elimination of a disjunct or a refinement of a star expression, indicate that the condition is not satisfied by all instances

  - if \(\text{Tighten}(c_1, d[c_1], \text{rest}, d, d')\) succeeds
    - return \(\text{Tighten}(c_1, d[c_1], \text{rest}, d, d')\)
  - else fail

\[
\text{procedure pull}(\text{name}\ n, \text{tightened } s\text{-DTD} \ d', \\text{original DTD} \ d')
\]

\[
d'[n^0] \leftarrow d[n]
\]

- for every name \(n'\) that occurs untagged in \(d'[n^0]\)
  - pull\((n', d', d)\)
We illustrate next how the above algorithm can convert an s-DTD into a tightest DTD.

**EXAMPLE 4.3** Consider the DTD (D4) from Example 3.4. Merging will collapse the *publication* and *publication*\(^4\) definitions into a single definition and remove the tags from all type definitions.\(^8\) At this point the view inference module will inform the user of non-tightness. The regular DTD after the merge is:

(D10)

\[
\begin{align*}
\langle \text{withJournals} : \text{professor}, \text{gradstudent} \rangle \\
\langle \text{professor} : \text{firstName}, \text{lastName}, \text{publication}, \text{publication}, \text{publication}, \text{publication}, \text{teaches} \rangle \\
\langle \text{gradStudent} : \text{firstName}, \text{lastName}, \text{publication}, \text{publication}, \text{publication}, \text{teaches} \rangle \\
\langle \text{publication} : \text{title}, \text{author}, \text{journal}, \text{conference} \rangle \\
\text{(title, author, journal)} \}
\end{align*}
\]

The resulting DTD can be simplified to the DTD (D2) in Example 3.1

**4.4. Result List Type Inference**

The tightening algorithm shows us how to compute the type of the elements that bind to the pick-variable of a pick-element query. Recall from Example 3.1 that finding the names of the elements that bind to the pick-variable and their types is not enough. In this section we complete the view inference by outlining the list-type inference algorithm that discovers the type of the top-level element of the view. The pseudo-code of the algorithm can be found in Appendix B.

The result list type inference algorithm works incrementally on the path ending at the pick variable. It introduces variables at every point in the path preceding the pick-variable and computes the result list type of each one of them by using the type of the previous list type. In particular, assume a query with a tree condition of the following form:

\[
l_k = \text{SELECT } L_k \\
\text{WHERE} \\
L_0 : (d_0) L_1 : (d_{1,1}) \ldots L_k : (d_{k,1}) \text{condition}_{k,1} / \\
(d_{k,2}) \text{condition}_{k,2} / \\
\vdots \\
(d_{k,i_k}) \text{condition}_{k,i_k} / \\
\langle / \rangle
\]

\(\vdots\)

\[
\langle d_{i+1,2} \rangle \text{condition}_{i+1,2} / \\
\vdots \\
\langle d_{i+1,i+1} \rangle \text{condition}_{i+1,i+1} / \\
\langle / \rangle
\]

\[
\langle / \rangle
\]

In the first step the algorithm computes the type of

\[
L_0 = \text{SELECT } L_0 \text{ WHERE } \ldots \text{ by invoking the tightening algorithm.}
\]

1. If the tightening algorithm declares that the condition is unsatisfiable with respect to the DTD then the type is \(\langle L_0 : e \rangle\).

2. If the tightening algorithm declares that the condition is valid with respect to the DTD then the type is \(\langle L_0 : d_k \rangle\), where \(d_k\) is the document type. Apparently \(d_k\) must be \(d_0\) or one of the names appearing in the disjunction \(d_0\).

3. If the tightening algorithm declares that the condition is satisfiable with respect to the DTD, as is the case in the running example, then the type is \(\langle L_0 : d_i \rangle\).

In each of the subsequent steps the algorithm computes the type of \(L_{i+1}, i = 0, \ldots, k - 1\) for the following query assuming that the type of the document type \(d_i\) is the one-level extension (see below) of the type of \(L_{i+1}\) according to the DTD.\(^9\)

\[
l_{i+1} = \text{SELECT } L_{i+1} \\
\text{WHERE} \\
\langle d_i \rangle L_{i+1} : (d_{i+1,1}) \ldots L_k : (d_{k,1}) \text{condition}_{k,1} / \\
(d_{k,2}) \text{condition}_{k,2} / \\
\vdots \\
(d_{k,i_k}) \text{condition}_{k,i_k} / \\
\langle / \rangle
\]

\[
\langle / \rangle
\]

**Definition 4.3** The one-level extension \(x(d)\) of a regular expression \(r\) according to a DTD \(d\) is the regular expression derived by replacing every name in \(r\) with its type.

\(^8\)Following the tightening algorithm step by step we can see that three specializations of *publication* are introduced. The third one, named *publication\(^2\)*, has essentially the same type with *publication\(^1\).*

\(^9\)Note that the one-level extension step of the algorithm makes it inappropriate for queries with recursive path expressions.
Then the specialized type of \( l_{i+1} \) is computed by projecting on the type of \( l_{i+1} \) the condition (or conditions) of the \( i + 1 \) level.

Let us illustrate projection and list inference with the following example. The complete algorithm can be found in [PV99].

**EXAMPLE 4.4** Consider the query (Q12) that operates on a source with the DTD (D11) and picks all titles and authors of student publications. We have introduced the variables \( D \) and \( G \) for the sake of explaining the algorithm.

(D11) \[
\{\langle \text{department} : \text{name}, \text{professor}+, \text{gradStudent}+, \text{courses}\rangle \\
\langle \text{professor} : \text{firstName}, \text{lastName}, \text{publication}+, \text{teaches}\rangle \\
\langle \text{gradStudent} : \text{firstName}, \text{lastName}, \text{publication}\rangle \\
\langle \text{publication} : \text{title}, \text{author}*, \text{(journal|conference)}\rangle \}
\]

(Q12) \[
\text{papers} = \text{SELECT P} \\
\text{WHERE D:}\langle \text{department}\rangle \\
\quad G:\langle \text{gradStudent}\rangle \\
\quad X:\langle \text{publication}\rangle \\
\quad P: \langle \text{title} \mid \text{author}\rangle 
\]

The algorithm first constructs a query that picks \( D \) into a result \( l_0 \) and computes the type of \( l_0 \). To do so it calls the specialization algorithm which declares the condition satisfiable\(^{10}\) and consequently the type of \( l_0 \) becomes \text{department}?.

In the next step the list inference algorithm works with the dummy query (Q13) and the hypothetical type \( \langle d_t : x(\text{department}?) \rangle \) or equivalently \( \langle d_t : (\text{name}, \text{professor}+, \text{gradStudent}+, \text{courses})? \rangle \).

(Q13) \[
l_t = \text{SELECT G} \\
\text{WHERE } \langle d_t\rangle \\
\quad G:\langle \text{gradStudent}\rangle \\
\quad X:\langle \text{publication}\rangle \\
\quad \langle \text{title} \mid \text{author}\rangle 
\]

Projecting the \text{gradStudent} condition on the type of \( d_t \) we get (note we keep only the first letters of the names)

\[
\text{project}((n, p+, g+, c*)?, g) \\
= (\text{project}(n, g), \text{project}(p, g)+, \text{project}(g, g)+, \text{project}(c, g)) = g*
\]

Then, by considering the query

\[ l_2 = \text{SELECT P} \]
\[ \text{WHERE } \langle d_t\rangle \]
\[ X: \langle \text{publication}\rangle \]
\[ P: \langle \text{title} \mid \text{author}\rangle \]

where the type of \( d_t \) is \( x(g*) = (f, l, p+)* \). Doing the projection of publication on this type we get \( \langle l_2 : p* \rangle \). Finally, we project the disjunction "title or author" on \( x(p*) = (t, a*, (j|c))\) and this gives us the correct result.

\[
\text{project}((t, a*, (j|c))w', t|a) \\
= (\text{project}(t, t|a), \text{project}(a, t|a)*, (\text{project}(j, t|a)\]
\[
\text{project}(c, t|a)))* = (t, a*) *
\]

5 Related Work

Our work with DTDs is closely related to problems in semistructured databases. In this section we describe the related work.

[GW97] introduces dataguides as OEM objects and studies problems of inference of dataguides from data and their use in query formulation and optimization.

The dataguides differ from DTDs in two important aspects, they do not capture constraints on order and cardinality and they do not capture constraints on the siblings. In this respect they are less powerful than the DTDs. However dataguides do not require the same type name to define the same type, so in this respect dataguides are similar to s-DTDs.

[BDPS97] defines graph schemas and studies their properties. The graph schemas are similar to dataguides but can include unary formulas on their edges. They discover that graph schemas are closed under application of UnQI queries [BDH99]. As in the case of dataguides, graph schemas cannot capture order, cardinality and constraints on the siblings.

[FS08] studies the problem of optimizing path expression with the aid of graph schemas. They introduce a query language that includes a limited form of tree conditions and paths. For this language they present algorithms for exact optimization of path queries. They define an optimal query as a query that returns a minimal answer. And they present algorithms for rewriting path queries into equivalent queries using state extents. They also present a polynomial approximation to the rewriting algorithm. Some of their results are applicable to our query language and DTDs.

[NUWC97] studies the inference of dataguides from data and approximations to dataguides. They introduce a concept of a representative object that allows one to compute a continuation of an object by a path.
expression. They then discuss various implementations of representative objects and their approximations and mention the utility of RO's in query optimization. In comparison to DTDs, RO's have the same shortcomings as Graph Schemas.

References


A. A Quick Presentation of the Logical Structure of XML Documents

This appendix summarizes the logical aspects of XML. It is included for the purpose of comparing the abstraction of XML by semistructured data against the actual logical part of the XML standard [].

A document consists of an element (usually containing other elements) and possibly a Document Type Definition (DTD) providing information about the structure of elements in the document. Each element has a name, a list of attribute/value pairs, and possibly content consisting of text and/or other elements. Elements without content are called empty. An element may have an id, specified by an ID attribute (see below). Attributes may also reference other elements using ids or lists of ids. A valid document is a document satisfying the following:

- no two elements in the document have the same id;
- each id occurring as an attribute value is the id of some element in the document; and
- the document contains a DTD and conforms to it (as explained below).

Thus, a document must have a DTD in order to be valid.

We next elaborate on the structure of elements. There are two kinds of elements:

1. EMPTY elements consist of a name and a list of attribute name/attribute value pairs. No attribute may appear twice in the list. Section A.4 presents the possible types of the attributes.

2. Non-empty elements consist of a name, a list of attribute name/attribute value pairs, and content. The content may be a sequence of elements (element content), a string (character content) or a sequence of elements and strings (mixed content).
Note that an element whose content is an empty list of elements is not an \textit{EMPTY} element. In OEM terms, an \textit{EMPTY} element corresponds to an atomic object while a non-empty element with empty content corresponds to a list object with an empty list of subobjects.

\subsection{A.1 Attribute Types}

XML supports the following attribute types.

1. \textit{ID}: the value of an \textit{ID} attribute must be unique within the document. There can be at most one \textit{ID} attribute in an element and there is a habit to name this attribute \textit{ID} (same name as the type).

2. \textit{IDREF}: the value of an \textit{IDREF} attribute must be the value of an \textit{ID} attribute occurring in the same document. Note that it is not possible to specify the name of the referenced element.

3. \textit{IDREFS}: a list of \textit{IDREF} values.

4. \textit{ENTITY}: the value is some identifier, say \texttt{img} that the DTD associates with some physical storage, say an image file \texttt{image.png}. Incidentally, note that the definition of “valid” and “well-formed” breaks down here: you have to have a DTD for the document to be even “well-formed” if you have entities but we’ll neglect this in the development of XML’s logical model.

5. \textit{ENTITIES}: a list of \textit{ENTITY} values.

6. \textit{Enumerated Types}

7. \textit{CDATA}: string for all practical purposes.

8. A bunch of other atomic types may come later by the committee.

\subsection{A.2 Document Type Definitions (DTDs)}

A valid document must conform to its DTD. A DTD specifies, for each element name that may appear in the document, an \textit{element type declaration} and an \textit{attribute list declaration}.

\textbf{Element Type Declaration} An \textit{element type declaration} describes for each element name the content of the elements, or specifies that the elements have no content (i.e., are \textit{EMPTY}). If the elements are not \textit{EMPTY}, the content may be as follows:

1. \textit{Element content}. The order and number of the children element is given by a \textit{regular expression} involving element names, choice lists, sequences, +, *, and ?.

2. \textit{Mixed content}. A sequence of elements and character data (#PCDATA). In this case the only allowed element type declaration is

   \[(PCDATA|\text{elem}_1|\ldots|\text{elem}_n)\]

3. \textit{Character Content}.

Note that all element names must have been declared even if they only appear as children of an element whose content type is \textit{ANY}.

\textbf{Attribute List Declaration} This specifies for each element name a list of attributes and their types (see description of the possible attribute types in Section A.1). Each attribute may be \textit{REQUIRED} or \textit{IMPLIED}. Furthermore it may have a default value or even be \textit{FIXED} to a specific value mentioned in the declaration.

\section{B The List inference algorithm}

\textbf{Algorithm InferList}  
\hspace*{0.5cm} INPUT: a variable name \texttt{v}  
\hspace*{0.5cm} a path \texttt{p = p_1 \ldots p_n}, variable \texttt{v} occurs on this path  
\hspace*{0.5cm} a tree condition with disjuncts and tags, \texttt{c}  
\hspace*{0.5cm} a DTD type \texttt{t}  
\hspace*{0.5cm} a DTD \texttt{d}  
\hspace*{0.5cm} OUTPUT: A type describing the possible lists of values of \texttt{v}  
\hspace*{0.5cm} METHOD: run function \texttt{InferList(v, p, c, t, d)}

function \texttt{InferList(v, p, c, t, d)}

if \texttt{c} is tree condition \texttt{c_1 \ldots c_n}

for each \texttt{c_i} such that \texttt{c_i} is not \texttt{p_1}

if \texttt{Spec(null, t, c_i, d, \{\})} results in fail

return \texttt{\epsilon}

else if \texttt{Spec results in satisfiable}

\hspace*{1cm} \texttt{t.d} \leftarrow \texttt{Spec(null, t, c_i, d, \{\})}

else if \texttt{Spec results in valid}

\hspace*{1cm} \texttt{t.d} \leftarrow \texttt{Spec(null, t, c_i, d, \{\})}

if \texttt{Spec(null, t, p_1, d, \{\})} results in fail

\hspace*{1cm} return \texttt{\epsilon}

\hspace*{1cm} \texttt{t} \leftarrow \texttt{refine(t, p_1)}

\hspace*{1cm} if refine fails, return \texttt{\epsilon}

\hspace*{1cm} \texttt{t} \leftarrow \texttt{project(t, p_1, d)}

if all specializations resulted in valid

\hspace*{1cm} \texttt{t} \leftarrow \texttt{substitute(d[p_1])} for \texttt{p_1} in \texttt{t}

else

\hspace*{1cm} \texttt{t} \leftarrow \texttt{substitute((d[p_1])?) for \texttt{p_1} in \texttt{t}}

return \texttt{InferList(v, p_1 \ldots p_n, c/p_1, v, t, d)}
function project( t, n(T), d )
    if r = n or r = n(T)
        return n(T)
    if r = n(T) and T' ≠ T (using could match semantics)
        if Spec(null, r, n(T), d, { }) fails
            return ε
        else
            return n(T)
    if r is (g)*
        return ( project(g, n(T), d ) )*
    if r is r_1, ..., r_m
        return project(r_1, n(T), d), project((r_2, ..., r_m), n(T), d)
    if r is r_1 | ... | r_m
        return project(r_1, n(T), d) | project((r_2 | ... | r_m), n(T), d)