The Use and Computation of Specialized DTDs in the MIX Mediator System*

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

Abstract

The MIX mediator system provides to its users integrated XML views of data exported by XML sources or other MIX mediators. The system uses Document Type Definitions (DTDs) *to help the user formulate queries on the integrated view and to help the query optimizer deliver more efficient plans. However, the "manual" creation of DTDs is time-consuming and error-prone, given the size of real world DTDs. In light of this challenge we developed the* view DTD inference *module that automatically computes the view DTD from the view definition and the source DTDs. We present 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 inherent deficiencies that prevent the development of tight DTDs. We specify* "DTDs with specialization" *that resolve this problem. We extend the view inference algorithm with the ability to infer DTDs for view definitions that have element order constraints expressed using precedence relationships. Finally we describe DTD-based query optimization algorithms and we show their use and importance in the mediation context.*

*This work was supported by the NSF-IRI 9712239 grant and equipment donations from Intel Corp.

Note to the Reviewers. This document is an extended version of the ICDE 99 paper with title "Enhancing Semistructured Data Mediators with Document Type Definitions". The ICDE 99 paper introduced the concept of sound and tight DTDs. It also introduced specialized DTDs and motivated their importance in view DTD inference. Finally, it described an algorithm for finding sound and tight DTDs for navigational conjunctive queries.

The current document besides providing the detailed description of algorithms that were omitted from the conference version (in particular, the tightening and list inference algorithms) it also describes the following novel and unpublished findings and developments:

1. It describes the use of (specialized) DTDs by the query optimizer of the MIX mediator system.

2. It enhances the considered view/query language with the ability to express precedence relationships and, hence, check the order of elements. Consequently, the view DTD inference algorithm is extended to account for the order constraints. The algorithm extension is based on a corresponding extension of the concept of tagging.

For completeness puproses, this document also describes two related developments. In particular,

1. it describes the use of DTDs by the *Blended Browsing and Querying (BBQ)* query formulation and result browsing tool. The BBQ system has already appeared in the SIGMOD 99 demonstrations program [B^+99] and a paper [MP] on the UI aspects of BBQ has been submitted to the 5th Visual Databases conference.

2. it extends the view language with the ability to create and "group by" objects and it briefly describes the essentials of the corresponding view inference algorithm extension. This material is drawn from [PVb] and will be submitted to the PODS 00 conference.

1. Introduction

The MIX mediator provides to its clients an XML view of the XML data exported by one or more applications or repositories that have been wrapped to provide a logical XML view of their data (see Figure 1.) The views select, integrate, and rank information according to the user's preferences. The views are developed using the mediator's query and view definition language, called XMAS (*X*ML *M*atching *A*nd *S*tructuring), which is essentially an XML modification of languages such as Lorel and YATL [QRS+95, CDSS98].

The architecture of the MIX mediator closely resembles that of TSIMMIS [PAGM96], a mediator for semistructured data. However, unlike the Object Exchange Model (OEM), which is the semistructured data model used by TSIMMIS, and other semistructured data models, XML documents are typically accompanied by a *Document Type Definition (DTD)* which describes the content and the structure of the objects (called *elements* in XML terminology) participating in a document.[1] In effect, DTDs are the XML documents' schema. However they are more versatile with respect to how much structure they impose on the document. They can represent structures as rigid as relational tables or as loose as unstructured data. 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.

The topic of this paper is how the MIX mediator system exploits the opportunities provided by the presence of DTDs and how it automatically computes DTDs for the views.[2] In particular, we show how the optimizer of MIX exploits DTDs to produce more efficient queries and how DTDs drive MIX's QBE-style query interface. Then we develop an algorithm required in order to compute the view DTDs (and hence realize many of the DTD benefits.) 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 as well as richer "XML schema" frameworks.

To illustrate the gains obtained by DTD use we 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 semistructured 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 composed query which refers directly to the source data (and not to the views anymore.) Then the optimizer finds a plan for executing the composed 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.[3]

[1] In this paper we focus on *valid* XML documents, i.e., documents that always have a DTD.

[2] We do not consider issues in wrapping non-XML sources.

[3] Indeed, 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

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 semistructured 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.

However the lack of structure brings two deficiencies. 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 in a heavy loss of performance. For example, the mediator may send an "irrelevant" query to some source. A much more challenging problem is the processing of queries on (the very common in mediation) fusion views [PAGM96]; in this case the lack of structure knowledge forces the mediator to send to the sources unecessarily complex and expensive queries. 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 system passes the view DTDs to the *Blended Browsing and Querying* GUI that assists the user in information discovery and query formulation. Once a query is formulated, with or without using BBQ, it is passed to the query processor. There the query optimizer employs the source DTDs to derive more efficient plans. We emphasize an algorithm for minimizing the conjunctions that appear in the `WHERE` clause of the queries/views and we show its application in the optimization of fusion queries/views.

Having accepted that DTDs are useful the next issue is how could we automate their development? Their manual creation is error-prone and time-consuming. Facing this challenge we develop a view DTD inference module that, given the source DTDs and the view definition, derives the view DTD. (There are more than one view DTDs. We explain below which one is the "best" and why.)

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 developed and implemented a view DTD inference algorithm (see Section 4) for the class of queries where the `WHERE` clause consists of conjunctions of non-recursive path conditions and order constraining precedence relationships. Note that each individual path condition is allowed to have "disjunction" as in generalized path expressions [AQM+97]. The `SELECT` clause can simply pick source elements or it can can create and "group by" new elements.[4]

topic of this paper and will not be discussed any further.

[4]The view inference algorithm extension for handling the "group by" feature is drawn from [PVb].

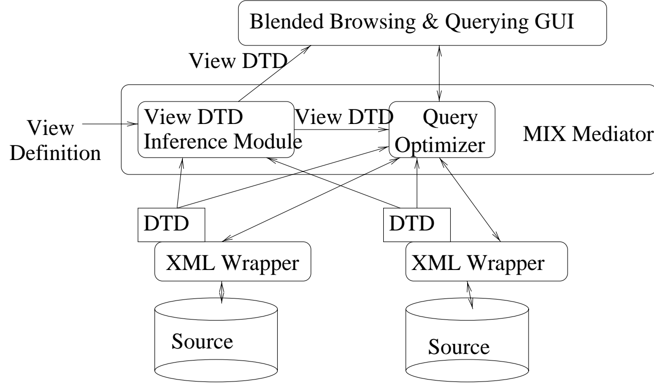


Figure 1. The MIX Mediator

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.[5] 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 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, XML schema formalisms and corresponding 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, called DTDs with specialization, that resolve many of the tightness problems of conventional DTDs.

4. We describe how the MIX mediator uses the constraints that DTDs impose on the XML data in order to optimize queries. First it prunes conjuncts that are unsatisfiable with respect to the source DTDs. Second, and technically more challenging, it employs a DTD-based conjunction minimization algorithm in order to simplify queries. We illustrate the benefits of this optimization in the case of queries on fusion views, i.e., views that construct objects that aggregate information from multiple sources [PAGM96].

We start with a mathematical abstraction of the XML model and the part of the XMAS query language that is used in this paper. Section 3 discusses the properties of view DTDs. Section 4 describes the view inference algorithms. Section 6 describes the use of DTDs in the BBQ system and in DTD-based query optimization. We conclude with related work.

[5] Furthermore, 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.

2. Model and Query Language Framework

We present next a mathematical abstraction of XML and DTDs. Without loss of generality 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. Consequently, we do not include attribute type declarations in the DTDs.[6]

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.

Definition 2.1 *[Element] An* element e *consists of a* name*, denoted as* $\mathrm{name}(e)$ *and* content*, denoted as* $\mathrm{content}(e)$*. The content is either a sequence of elements or a PCDATA value, i.e., a character string.*

A document is simply an element. (We slightly deviate from XML's definition, where the DTD is also "packaged" into the document.)

Definition 2.2 *[DTD] A DTD is a pair consisting of the top name* n_t *and a set of declarations* $\{\langle n : \mathrm{type}(n)\rangle\}_{n \in N}$*, where* N *is the set of names and* $\mathrm{type}(n)$ *is either a regular expression over* N *or* PCDATA.

We will often omit mentioning the top name and, by default, it will be the name declared first.

Definition 2.3 *We say that a document* d satisfies *a DTD* D*, denoted as* $d \models D$ *if the top name of the DTD is the name of the top-level element of* d *and for every (sub)element* e *of* d *the following hold:*

1. $\mathrm{name}(e) \in N$*, where* N *is the set of element names.*

2. if $\mathrm{content}(e) = e_1 \ldots e_m$ *then* $\mathrm{name}(e_1) \ldots \mathrm{name}(e_m) \in \mathcal{L}(\mathrm{type}(\mathrm{name}(e)))$ *and* $e_i \models D$, $1 \leq i \leq m$. *(We denote by* $\mathcal{L}(r)$ *the regular language over type names, that is described by* r*)*

3. else if $\mathrm{content}(e)$ *is a string then* $\mathrm{type}(\mathrm{name}(e)) = \mathrm{PCDATA}$.

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* $N = \{n_1, \ldots, n_k\}$.

[6] 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.

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 $r|\epsilon$.

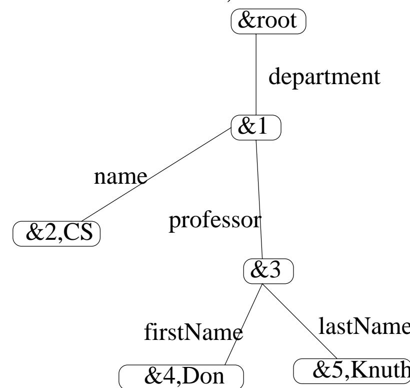
2.1. Query Language

The part of the query/view definition language XMAS we use in this paper is common to essentially all XML and semistructured data query languages [AQM+97, CDSS98, DFF+]. 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.

The presented view inference algorithm works with queries where the `WHERE` clause consists of a conjunction of non-recursive generalised path conditions and is applied to exactly one source. The view DTD inference generalization for multiple sources and value-based joins is straightforward. The only form of negation we consider is the ability to say that two elements are different. Besides path conditions we also allow conditions on the order of elements, which are expressed with the precedence predicates $<$ and $>$, whose arguments are element variables. For example, the predicate $X < Y$ requires that the binding of X appears before the binding of Y.

In its simplest form the `SELECT` clause has a single variable. We call such queries *pick-element* queries. The elements that bind to the pick-variable are grouped into the view document whose name precedes 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, i.e., the query preserves order.

The semantics of our language are based on viewing XML documents as labelled trees of the following form (which is identival to OEM):



We illustrate the syntax and semantics of the query language with the following examples. As it is immediately obvious the `WHERE` clause follows the syntax and semantics of path expressions in OQL and Lorel [AQM+97]. [7]

(Q1) withJournals =

```
SELECT P
WHERE root.department D, D.name=CS,
  D.(professor|gradStudent) P,
  P.publication Pub1, Pub1.journal,
  P.publication Pub2, Pub2.journal, Pub1 != Pub2
```

The variable `P` binds to all `professor` or `gradStudent` elements that

1. are contained in a `department` element `D`,
2. the department contains an element `name` whose string content binds to `CS`
3. `P` contains two different `publication` elements that contain `journal` subelements.

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.

The following query creates a "journalPerson" element for every element `P` with at least two journal publications. Each created "journalPerson" contains the last name and all journal publications of the specific person.

(Q2) withJournals =

```
SELECT <journalPerson>
          L FOR L
          Pub1 FOR Pub1
       </journalPerson> FOR P
WHERE root.department D, D.name=CS,
  D.(professor|gradStudent) P,
  P.lastName L,
  P.publication Pub1, Pub1.journal,
  P.publication Pub2, Pub2.journal, Pub1 != Pub2
```

The semantics are as follows: Derive all tuples of variable bindings that satisfy the `WHERE` clause. Then process the `SELECT` clause top-down; for every binding p of `P` create a `journalPerson` element. Then place in this `journalPerson` the list of all bindings l of `L`, for the specific p, and then the list of all `Pub1` bindings – again for the specific p.

[7] There is also an implementation where the syntax follows the more XML-like XML-QL syntax [DFF+].

As one can see the `FOR` feature is essentially a modification of the group-by mechanisms of OQL and YATL. In general, a variable has to be within the scope of a `FOR` statement that mentions its name. However, for simplicity one is allowed to write

```
SELECT <journalPerson>
           L
           Pub1
       </journalPerson> FOR P
```

in which case the query preprocessor turns it into "`L FOR L`" and "`Pub1 FOR Pub1`".

3. View DTDs

The view DTD inference module infers the DTD of a view from the source DTDs and the view definition. However, caution is needed when using the phrase "the DTD of the view" since a view has multiple possible DTDs. What the query processor and the query formulation GUI need are precise DTDs that, if possible, describe only the view element structures that are actually possible. We first provide two criteria, called *soundness* and *tightness*, that formalize the above idea. It turns out that inherent DTD weaknesses decrease the "precision" of view DTDs (see Section 3.2.) Our specialized DTDs (Section 3.3) resolve such non-tightness problems to a significant extent.

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 D_V is sound if, given source DTDs $D_1, D_2, \ldots, D_n$ and a view definition V, for every tuple $(d_1 \ldots d_n)$ of n documents such that $d_1 \models D_1, d_2 \models D_2, \ldots, d_n \models D_n$ the view document $V(d_1, \ldots, d_n)$ satisfies D_V.*

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 $\langle n : r \rangle$ is tighter than a type $\langle n : r' \rangle$ if $\mathcal{L}(r) \subseteq \mathcal{L}(r')$, i.e., every sequence of elements described by r is also described by r'.*

Definition 3.4 *A DTD D_V is a tightest view DTD for given source DTDs $D_1, D_2, \ldots, D_n$ and a view definition V if there is no view DTD D'_V such that D'_V is tighter than D_V.*

For the class of pick-element queries the view inference algorithm can "tighten" 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)

$$
\begin{aligned}
\{\langle \mathit{department}: \;& \mathit{name}, \mathit{professor}*, \mathit{gradStudent}*, \\
& \qquad\qquad\qquad\qquad \mathit{course}*\rangle \\
\langle \mathit{professor}: \;& \mathit{firstName}, \mathit{lastName}, \mathit{publication}*, \\
& \qquad\qquad\qquad\qquad \mathit{teaches}\rangle \\
\langle \mathit{gradStudent}: \;& \mathit{firstName}, \mathit{lastName}, \mathit{publication}*\rangle \\
\langle \mathit{publication}: \;& \mathit{title}, \mathit{author}*, (\mathit{journal}|\mathit{conference})\rangle\}
\end{aligned}
$$

Note that for simplicity we have omitted all *PCDATA* declarations, such as

$\langle firstName : PCDATA\rangle$

(Q3) withJournals =

```
SELECT P
WHERE root.department D,
  D.(professor|gradStudent) P,
  P.publication Pub1, Pub1.journal,
  P.publication Pub2, Pub2.journal, Pub1 != Pub2
```

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

$$\langle \mathit{withJournals} : (\mathit{professor}|\mathit{gradStudent})*\rangle$$

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 (Q3) 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 (DTD2) shows that *professors* appear before *gradStudents*.

(D2)

$$
\begin{aligned}
\{\langle \mathit{withJournals} : \;& \mathit{professor}*, \mathit{gradstudent}*\rangle \\
\langle \mathit{professor}: \;& \mathit{firstName}, \mathit{lastName}, \mathit{publication}, \\
& \qquad\qquad \mathit{publication}+, \mathit{teaches}\rangle \\
\langle \mathit{gradStudent}: \;& \mathit{firstName}, \mathit{lastName}, \mathit{publication}, \\
& \qquad\qquad\qquad\qquad \mathit{publication}+\rangle \\
\langle \mathit{publication}: \;& \mathit{title}, \mathit{author}*, (\mathit{journal}|\mathit{conference})\rangle\}
\end{aligned}
$$

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

EXAMPLE 3.2 Consider the query (Q4) 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*.

(Q4) publist =

```
SELECT P
WHERE root.department.(professor|gradStudent).publication P,
  P.journal
```

The view DTD is then:

$$\text{(D3)} \quad \{\langle \textit{publist} : \quad \textit{publication}* \rangle \quad \langle \textit{publication} : \quad \textit{title}, \textit{author}*, \textit{journal} \rangle \}$$

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.

To formalize this information loss phenomenon we will have to focus on the element name structure of XML documents, since it is the element names that are described by DTDs. We do so by introducing the *structure representatives* of XML documents. A structure representative encompasses the element name structure of an XML document but it discards its string content (i.e., the PCDATA).

Then we introduce the structural tightness property of view DTDs. Intuitively a view DTD is non-tight if it describes document structures that cannot be produced by the view.

Definition 3.5 *A* structure representative *of a document d is a document d' that is identical to d modulo the fact that every string has been replaced with the string* `dummy`*. We will also say that d satisfies d'.*

Obviously a structure representative d' of a document d satisfies a DTD D if and only if d satisfies D.

Definition 3.6 *Given a set of source DTDs $D_1, \ldots, D_n$ and a view V, a DTD D_V is structurally tight if*

1. *it is the tightest DTD of the view given the source DTDs,*
2. *for every structure representative d'_V that satisfies D_V there is a view document d_V that satisfies d'_V and there are also source documents $d_1, \ldots, d_n$, satisfying $D_1, \ldots, D_n$ such that $d_V = V(d_1, \ldots, d_n)$.*

Using the Definition 3.6 we characterize the DTD (D2) as non-tight because there is a structure representative d'_V, say the *withJournals* document that has one *professor* having no journal publications, that does not meet the second condition of the definition. In particular, d'_V satisfies the view DTD (D2), yet there is no possible valid source document d_1 such that the view (Q3) when applied to d_1 will result in a document that satisfies the structure d'_V.

On the other hand DTD (D3) is tight according to Definition 3.6

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, all pick element views have a structurally tight specialized view DTD. Note however, that 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 remark at the end of the section).

Definition 3.7 *A specialized DTD (s-DTD) is a set*

$$\{\langle n^i : type(n^i)\rangle\}_{n^i \in N^+}$$

where $N^+ = \{n^i | n \in N, i = 0, \ldots, spec(n)\}$ and $spec(n)$ is a non-negative integer defined for all $n \in N$. 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.

s-DTDs besides being tight descriptors of views are also very important as temporary results of our algorithms. Nevertheless, one may eventually want to stick with conventional DTDs. In this case we need to convert the s-DTD to a regular DTD. For this purpose we define the image:

Definition 3.8 *The image of a sequence $\langle n_1^{i_1} \ldots n_m^{i_m}\rangle$ 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' derived if we replace each name n^i of r with n.*

EXAMPLE 3.3 For instance the image of the tagged type $\langle title,\ author^1,\ author^2\rangle$ is just $\langle title, author, author\rangle$.

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

Definition 3.9 *An element e satisfies an s-DTD D if the following hold*

- $n \in N$, *where* $n = \mathrm{name}(e)$[8],
- *there is an* $i, 0 \leq i \leq spec(n)$ *such that*
  - *if* $content(e)$ *is a string then* $\mathrm{type}(n^i)$ *is PCDATA; or*
  - *if* $content(e) = e_1 \ldots e_m$ *then* $name(e_1) \ldots name(e_m) \in image(type(n^i))$, *and* $e_i \models D$, $1 \leq i \leq m$.

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(Q3) 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 $publication^1$ that defines journal papers only. Then we require each professor/gradStudent to have exactly two $publication^1$ objects and optionally other publications. The full specialized DTD is:

(D4)

$$
\begin{array}{ll}
\{\langle withJournals: & professor*, gradstudent*\rangle \\
\langle professor: & firstName, lastName, publication*, \\
 & \quad publication^1, publication*, \\
 & \quad publication^1, publication*, teaches\rangle \\
\langle gradStudent: & firstName, lastName, publication*, \\
 & \quad publication^1, publication*, \\
 & \quad publication^1, publication*\rangle \\
\langle publication: & title, author*, (journal|conference)\rangle\} \\
\langle publication^1: & title, author*, journal\rangle\}
\end{array}
$$

[8] This condition stayed the same with plain DTDs

Remarks on the Non-Existence of Tightest and Tight s-DTDs The pick-element views considered in this paper always have tight s-DTDs. However, once we extend beyond the pick-element class of this paper there are some bad news (at least from a theory point of view) regarding the existence of tightest and tight s-DTDs. As demonstrated below, pick-element views involving recursive paths may not have tight s-DTDs when evaluated on sources whose DTDs are recursive, in the sense that the element declaration of a name n may directly or indirectly involve n. Indeed, such views do not have a tightest view DTD either.

EXAMPLE 3.5 Consider the following DTD and the following pick-element query with recursive paths.

(D5) $\{\langle section : prolog, section*, conclusion\rangle\}$

(Q5)
```
startsAndEnds =
    SELECT X
    WHERE root.section*.X, X.(prolog|conclusion)
```

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".

(T6) $\langle startsAndEnds : (prolog|conclusion)*\rangle$

(T7) $\langle startsAndEnds : prolog, conclusion|$
$prolog, (prolog|conclusion)*, conclusion\rangle$

(T8) $\langle startsAndEnds : prolog, conclusion|$
$prolog, (prolog|conclusion)*, conclusion|$
$prolog, prolog, (prolog|conclusion)*,$
$conclusion, conclusion\rangle$

A second case of non-tightness has to do with maintaining constraints across lists constructed by the `SELECT` clause of non-pick-element queries. For example, consider the following query that creates a single `total` element containing a list consisting of the concatenation of all professor `lastName`'s and all `teaches`' elements.

```
namesAndClasses =
SELECT <total>
          L FOR L
          T FOR T
       </total>
WHERE root.department.professor P
       P.lastName L, P.teaches T
```

The following DTD is non-tight because it does not capture that the number of `lastName` and the number of `teaches` elements have to be equal.

(D5) $\{\langle namesAndClasses : total\rangle$
$\langle total : lastName*, teaches*\rangle\}$

How important will such non-tightness be in practice is difficult to assess, especially since approximations such as (D5) are doing a reasonable job of describing the view. For now, we may only comment that we have not seen realistic examples where the effective use of s-DTDs by the BBQ query formulator and the MIX mediator's optimizer has been affected by such forms of non-tightness.

4 Algorithms

We first show how to infer an s-DTD for the type of elements that bind to the pick-variable X in queries of the form (Q6). This will be an important building block of view inference since it shows how conditions refine the types on which they are applied. Section 4.4 describes the computation of the type of the view's top element and Section 5 completes the picture by generalizing to non-pick-element views.

(Q6) `SELECT` X `WHERE` `root.`$rootname.X, X : \langle conditions\ on\ X\ and\ its\ subobjects\rangle$

4.1. DTD Type Refinement

The DTD tightening algorithm of the inference module refines the types of the source DTD in order to make them always satisfy the element name conditions applied on the picked variable. 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 refine*(r, n) *of a regular expression* r *given a name* n *is the regular expression* r' *that describes all strings of* $\mathcal{L}(r)$ *that contain at least one instance of* n.

The algorithm that computes *refine*(r, n) uses the special operators '$\otimes$' and '$\|$' that extend the regular expression operators ',' and '|':

$$r_1 \otimes r_2 = \{ \begin{array}{l} fail,\ if\ r_1 = fail\ or\ r_2 = fail, \\ r_1, r_2,\ otherwise \end{array}$$

$$r_1 \| r_2 = \{ \begin{array}{l} fail,\ if\ r_1 = fail\ and\ r_2 = fail, \\ r_1,\ if\ r_1 \neq fail\ and\ r_2 = fail, \\ r_2,\ if\ r_1 = fail\ and\ r_2 \neq fail, \\ r_1 | r_2,\ otherwise \end{array}$$

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 $\mathit{refine}(r', n) \| \mathit{fail}$
  if $r = g*$ then return $g * \otimes \mathit{refine}(g, n) \otimes g*$
  if $r = r_1, r'$ then return $(\mathit{refine}(r_1, n) \otimes r') \|$
      $(r_1 \otimes \mathit{refine}(r', n))$
  if $r = r_1 | r'$ then return $\mathit{refine}(r_1, n) \| \mathit{refine}(r', n)$

EXAMPLE 4.1 Consider the DTD (D10) and the query (Q7)

(D10) $\{\langle \mathit{professor} : \ \mathit{name}, (\mathit{journal} | \mathit{conference})* \rangle\}$

(Q7) answer =
```
SELECT X
WHERE root.professor X, X.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.

$$
\begin{aligned}
& \mathit{refine}(`n, (j|c)*', j) \\
& = \mathit{refine}(n, j) \otimes (j|c)* \ \| \ n \otimes \mathit{refine}((j|c)*, j) \\
& = (\mathit{fail} \ \| \ n, \mathit{refine}((j|c)*, j)) \\
& = n, (j|c)* \otimes \ \mathit{refine}(j|c, j) \otimes (j|c)* \\
& = n, (j|c)* \otimes \ (\mathit{refine}(j, j) \ \| \ \mathit{refine}(c, j)) \otimes (j|c)* \\
& = n, (j|c)* \otimes \ (j \ \| \ \mathit{fail}) \otimes (j|c)* \\
& = n, (j|c)*, \ j, \ (j|c)*
\end{aligned}
$$

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* $\mathit{refine}(r, n^T)$ *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* $\mathit{image}(s_1), n, \mathit{image}(s_2)$ *is a member of* $\mathcal{L}(r)$.

Now, let us consider a `WHERE` clause that requires the variables X and Y to bind to elements named n, that are children of some p, where $\langle p : r \rangle$. In this case we'll create two tagged symbols n^X and n^Y correspondingly and call the tagged regular expression refinement routine twice; first as

$$r' = \mathit{refine}(r, n^X)$$

and then as

$$r'' = \mathit{refine}(r', n^Y)$$

If the query specifies that $X \neq Y$ then each string of $\mathcal{L}(r'')$ must contain exactly one n^X and exactly one n^Y. However, if the query does not specify that $X \neq Y$ $\mathcal{L}(r'')$ may either

1. contain exactly one n^X and exactly one n^Y, or
2. contain exactly one n^{XY}

Note that we have taken the freedom to use variable names and lists of variable names (instead of numbers) as tag symbols. This generalization obviously does not affect our s-DTD results.

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) $\quad$ $T \neq 0$
if $r = n$ $\quad$ *recall n is a shortcut for n^0*
  then return n^T
if $r = n'^{T'}$ where $n'^{T'}$ is a tagged name and $(n' \neq n \vee T' \neq 0 \vee T' \neq T)$
  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.

(Q8) answer =
```
SELECT X
WHERE root.professor X, X.journal J1, X.journal J2, J1 != J2
```

The tightening algorithm will tag the two instances of journal as $\mathit{journal}^{J_1}$ and $\mathit{journal}^{J_2}$ or, for brevity, $\mathit{journal}^1$ and $\mathit{journal}^2$. For brevity let us also use the first letters of the names. First it refines the type $n, (j|c)*$ (recall, this is a shorthand for $n^0, (j^0|c^0)*$ with j^1 and the result is further refined with j^2.

$$\begin{aligned}
&\mathit{refine}(`n, (j|c)*', j^1) \\
&= \mathit{refine}(n, j^1) \otimes (j|c)*) \parallel (n \otimes \mathit{refine}((j|c)*, j^1)) \\
&= (fail \parallel (n \otimes ((j|c) * \otimes \mathit{refine}((j|c)*, j^1) \otimes (j|c)*)) \\
&= n, (j|c) * \otimes (\mathit{refine}(j, j^1) \parallel \mathit{refine}(c, j^1)) \otimes (j|c)* \\
&= n, (j|c)*, j^1, (j|c)*
\end{aligned}$$

$$\begin{aligned}
&\mathit{refine}(`n, (j|c)*, j^1, (j|c)*', j^2) \\
&= (\mathit{refine}(n, j^2) \otimes (j|c)*, j^1, (j|c)*) \parallel \\
&(n \otimes refine(`(j|c)*, j^1, (j|c)*', j^2)) \\
&= fail \parallel n \otimes (\mathit{refine}((j|c)*, j^2) \otimes j^1, (j|c)* \parallel \\
&(j|c) * \otimes \mathit{refine}(`j^1, (j|c)*', j^2)) \\
&= n, ((j|c) * \otimes \mathit{refine}((j|c), j^2), j^1, (j|c)*) \parallel \\
&(j|c)*, (\mathit{refine}(j^1, j^2) \otimes (j|c)* \parallel j^1 \otimes \mathit{refine}(j|c)*))) \\
&\ldots \\
&= (n, (j|c)*, j^2, (j|c)*, j^1, (j|c)*) \mid \\
&(n, (j|c)*, j^1, (j|c)*, j^2, (j|c)*)
\end{aligned}$$

One can easily generalize *refine*() to take as input a disjunction of names $n_1^T, \ldots, n_m^T$ instead of a single name n^T.

Order Constraints The tagged regular expressions described above provide a simple solution to the issue of incorporating order constraints.

Let us assume two variables X and Y binding to children of p, where $\langle p : r \rangle$, and also consider the constraint $X < Y$. First r is refined using X and Y. The next goal is to produce an r'' such that

$$\mathcal{L}(r'') = \{s | s \in \mathcal{L}(r') \ and \ n^X \ appears \ before \ n^Y \ in \ s\}$$

Let us call *sieve()* the procedure that constructs r''. Its implementation is based on the following observation about the structure of r': The tagged regular expressions produced from the *refine()* procedure never have a tagged name within the scope of a Kleene closure. Indeed, those expressions are always of the form

$$(r_{1,0}, n_{1,1}^{T_{1,1}}, r_{1,1}, \ldots, n_{1,m_1}^{T_{1,m_1}}, r_{1,m_1}) | \ldots | (r_{k,0}, n_{k,1}^{T_{k,1}}, r_{k,1}, \ldots, n_{k,m_k}^{T_{k,m_k}}, r_{k,m_k})$$

and none of the $r_{i,j}$ contains a tagged symbol. Given this format it is straightforward to implement *sieve()* as follows:

function *sieve*(r,$X < Y$)
  if $r = r_1 | \ldots | r_m$ then
    return $sieve(r_1, X < Y) \| \ldots \| sieve(r_m, X < Y)$
  else if $r = r_0, n_1^{T_1}, r_1, \ldots, n_m^{T_m}, r_m$ then
    if there is an $n_i^{T_i}$ whose tag contains X and

an $n_j^{T_j}$ whose tag contains Y and $i < j$ then

return r

else

return fail

EXAMPLE 4.3 Consider the following DTD and query.

(D11) $\{\langle professor : \ name, (journal|conference)(journal|conference)\rangle\}$

(Q9) answer =

```
SELECT X
WHERE root.professor X, X.journal J, X.conference C, J<C
```

The refinement of the `professor` type will result into

$$(n, j^J, c^C)|(n, c^C, j^J)$$

Then the *sieve()* will reduce it into

$$\begin{aligned}
&sieve(((n, j^J, c^C)|(n, c^C, j^J)), J < C) \\
&= sieve((n, j^J, c^C), J < C)\|sieve((n, c^C, j^J), J < C) \\
&= n, j^J, c^C
\end{aligned}$$

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 (Q6). 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.

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 3.8) and then to merge all images that have the same name. We also want to inform the user that a merging has occured, since merging inadvertently introduces non-tightness.

The algorithm is given below:

```
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, type(r), e, d, {})
             pull all names that occur untagged in the
                 resulting s-DTD

    procedure Tighten(n, t, c, d, d')
                    n is the name to be tightened
                      (n 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 list of conditions
      d' ← Tighten(n, t, c_1, d, d')
      for each condition c_i, i >1
          if Tighten(n, dtd[n], c_i, d, d') does not fail
             d' ← Tighten(n, dtd[n], c_i, d, dtd)
          else fail
      return d'

    else c is a name c_1^T with tag T, followed by a list of
                     conditions rest
      d'[n^T] ← refine(t, c_1^T)
              if Tighten(c_1, d[c_1], rest, d, d') succeeds
                 return Tighten(c_1, d[c_1], rest, d, d')
              else fail

    procedure pull(name n, tightened s-DTD d',
                                          original DTD d)
    d'[n^0] ← d[n]
    for every name n' that occurs untagged in d'[n^0]
      pull(n', d', d)
```

Figure 2. The Tightening Algorithm

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 $\langle n^T : type(n^T) \rangle$ of d
  if d' contains the type definition $\langle n : type(n) \rangle$
    replace $\langle n : type(n) \rangle$ with
$$\langle n : type(n) | image(type(n^T)) \rangle$$
    signal the merge
  else
    insert in d' $\langle n : image(type(n^T))) \rangle$

We illustrate next how the above algorithm can convert an s-DTD into a tightest DTD.
EXAMPLE 4.4 Consider the DTD (D4) from Example 3.4. Merging will collapse the *publication* and *publication*^{1} definitions into a single definition and remove the tags from all type definitions.[9] At this point the view inference module will inform the user of non-tightness. The regular DTD after the merge is:

(D12)
$\{\langle withJournals : professor*, gradstudent* \rangle$
$\langle professor :$ $firstName, lastName, publication*,$
$publication, publication*, publication,$
$publication*, teaches \rangle$
$\langle gradStudent :$ $firstName, lastName, publication*,$
$publication, publication*, publication,$
$publication* \rangle$
$\langle publication :$ $(title, author*, (journal | conference)) \;|$
$(title, author*, journal) \rangle \}$

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 A.

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 preceeding 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:

[9] 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 *paublication*^{1}.

$$
\begin{array}{l}
l_k = \text{SELECT } L_k \\
\quad \text{WHERE} \\
\quad L_0 : \langle d_0 \rangle \; L_1 : \langle d_{1,1} \rangle \; \ldots L_k : \langle d_{k,1} \rangle \, condition_{k,1} \langle / \rangle \\
\qquad\qquad\qquad\qquad \langle d_{k,2} \rangle \, condition_{k,2} \langle / \rangle \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad \langle d_{k,i_k} \rangle \, condition_{k,i_k} \langle / \rangle \\
\qquad\qquad\qquad\qquad \langle / \rangle \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad \langle d1, 2 \rangle \, condition_{1,2} \langle / \rangle \\
\qquad\qquad \vdots \\
\qquad\qquad \langle d_{1,i_1} \rangle \, condition_{1,i_1} \langle / \rangle \\
\qquad\qquad \langle / \rangle
\end{array}
$$

In the first step the algorithm computes the type of $l_0 =$ `SELECT` L_0 `WHERE`... as follows:

1. If the condition is unsatisfiable with respect to the DTD then the type is $\langle l_0 : \epsilon \rangle$.
2. If the condition is valid with respect to the DTD then the type is $\langle l_0 : d_t \rangle$, where d_t is the document type. Apparently d_t must be d_0 or one of the names appearing in the disjunction d_0.
3. If the condition is satisfiable with respect to the DTD, as is the case in the running example, then the type is $\langle l_0 : d_t ? \rangle$.

A condition c is valid w.r.t. to the DTD d if c is satisfied by every document that satisfies d. The tightening algorithm is extended to compute the validity of the tree condition c with respect to the DTD d as follows:

- The refinement algorithm is modified to compute validity of a regular expression re w.r.t. the name n:
  - if re is a sequence $s_1 \ldots s_n$, return "valid" if any of the s_i is "valid"
  - if re is a disjunction $d_1 \ldots d_n$, return "valid" if any of the d_i is "valid"
  - if re is $s*$, return "invalid"
  - if re is name n, return "valid"
- If every refinement performed by the tightening algorithm results in "valid", then the condition c is valid wrt the DTD d.

Satisfiability of a condition w.r.t. the DTD is established by the tightening algorithm. The condition is satisfiable, if the tightened DTD is nonempty.

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_t is the one-level extension (see below) of the type of l_i according to the tightened DTD.[10]

[10]Note that the one-level extension step of the algorithm makes it inappropriate for queries with recursive path expressions.

$$
\begin{array}{l}
l_{i+1} = \text{SELECT } L_{i+1} \\
\quad \text{WHERE} \\
\quad \langle d_t \rangle\ L_{i+1} :\langle d_{i+1,1} \rangle \ \ldots L_k :\langle d_{k,1} \rangle\, condition_{k,1} \langle / \rangle \\
\qquad\qquad\qquad\qquad\qquad \langle d_{k,2} \rangle\, condition_{k,2} \langle / \rangle \\
\qquad\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad\qquad\qquad \langle d_{k,i_k} \rangle\, condition_{k,i_k} \langle / \rangle \\
\qquad\qquad\qquad\qquad\qquad \langle / \rangle \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad \langle d_{i+1,2} \rangle\, condition_{i+1,2} \langle / \rangle \\
\qquad\qquad \vdots \\
\qquad\qquad \langle d_{i+1,i_{i+1}} \rangle\, condition_{i+1,i_{i+1}} \langle / \rangle \\
\qquad\qquad \langle / \rangle
\end{array}
$$

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

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 A.

EXAMPLE 4.5 Consider the query (Q14) that operates on a source with the DTD (D13) and picks all titles and authors of student publications. We have introduced the variables `D` and `G` for the sake of explaining the algorithm.

(D13)

$$
\begin{array}{ll}
\{\langle department : & name, professor+, gradStudent+, \\
 & \qquad\qquad course*\rangle \\
\langle professor : & firstName, lastName, publication+, \\
 & \qquad\qquad teaches\rangle \\
\langle gradStudent : & firstName, lastName, publication*\rangle \\
\langle publication : & title, author*, (journal|conference)\rangle\}
\end{array}
$$

(Q14) papers = SELECT P
WHERE D:<department>
G:<gradStudent>
X:<publication>
P:<title | author> </></></>

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 tightening algorithm which declares the condition satisfiable[11] and consequently the type of l_0 becomes *department*?.

In the next step the list inference algorithm works with the dummy query (Q15) and the hypothetical type $\langle d_t : x(department?)\rangle$ or equivalently $\langle d_t : (name, professor+, gradStudent+, course*)?\rangle$.

[11]It cannot be valid because publications are optional for graduate students.

(Q15) l_1 = SELECT G
WHERE <d_t>
G:<gradStudent>
X:<publication>
<title | author> </></></>

Projecting the `gradstudent` condition on the type of d_t we get (note we keep only the first letters of the names)

$$\begin{aligned} & project('(n, p+, g+, c*)?', g) \\ & = (project(n, g), project(p, g)+, project(g, g)+, \\ & \quad project(c, g)*)? \ = g* \end{aligned}$$

Then, by considering the query

(Q16) l_2 = SELECT P
WHERE <d_t>
X: <publication>
P: <title | author> </> </>

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.

$$\begin{aligned} & project(`(t, a*, (j|c))*', t|a) \\ & = (project(t, t|a), project(a, t|a)*, (project(j, t|a)| \\ & \quad project(c, t|a)))* = (t, a*)* \end{aligned}$$

5 Beyond pick-element views

For the sake of completeness we summarize the extension of view DTD inference to include `FOR`, as has been described in [PVb]. The key point in inferring DTDs for views involving `FOR` is to work top-down on the `SELECT` clause structure. For example, in the case of query (Q2) the algorithm will first derive the type of `withjournals` by working on a query that has the hypothetical head

```
withJournals = SELECT P
```

After it derives the type of `withJournals` for the above it will appropriately change the names appearing in the type to `journalPerson`. Then, in order to derive the type of `journalPerson` the algorithm will essentially perform inference for the following query where the "professor" has been instantianted to a specifi object id `root_prof` that operates as the head of a dummy source.

```
journalPersonL = SELECT L
                 WHERE root_prof.lastName L,
                       root_prof.publication Pub1, Pub1.journal
                       root_prof.publication Pub2, Pub2.journal, Pub1 != Pub2
```

Similarly a type is derived for the `Pub1` list and the two types are concatenated in the type of `journalPerson`.

6 Use of DTDs in the MIX System

The MIX system utilizes DTDs to assist the user in information discovery and query formulation and to assist the query processor in deriving more efficient plans. In particular, the inferred view DTD is passed to the *Blended Browsing and Querying (BBQ)* Graphical User Interface (GUI), which displays the DTD structure of the view and allows one to intuitively formulate queries on the view and even on the results of previous queries. The BBQ interface is described in Section 6.1. We describe only aspects that show the use of DTDs in querying. The reader will find in `http://www.db.ucsd.edu/publications/bbq.pdf` an extensive discussion of the UI issues and the blending of querying and browsing in the BBQ interface.[12]

Section 6.2 describes the ways in which DTD knowledge speeds up query processing. Emphasis is given on the novel DTD-based simplification algorithm.

6.1 The BBQ User Interface: DTD-driven Query Formulation

The graphical user interface BBQ allows the construction of queries in an intuitive way, based on the DTD of the mediator view. Figure 3 depicts the main windows of the first version of BBQ which is available via `http://www.db.ucsd.edu/Projects/MIX/BBQ_User_Interface.html`.[13] The *condition* window, on the left, assists the formulation of the `WHERE` clause. The *answer* window, on the right, assists the formulation of the `SELECT` clause. The results returned by the MIX mediator are displayed in a separate window and can be navigated in a way that reminds navigating a Windows directory structure. Indeed, as Figure 3 shows, the query formulation windows, which displays the DTD information, also subscribe to the Windows directory navigation scheme. Hence browsing and querying of XML data are smoothly integrated.

We illustrate the main features by means of a typical mediation application which we refer to as the home buyer's scenario and is accessible at `http://www.npaci.edu/DICE/MIX/home_buyer/`.[14] In the home buyer scenario the user queries a mediator that has fused information from multiple real-estate related sites. For example, he queries for

> houses with 3 bedrooms, 2 baths, and price between $250K and $350K. Group the answers by region and for each home also show the nearby schools.

[12] This document has been submitted for publication to the 5th Visual Database Conference.

[13] This version was demonstrated in SIGMOD 99.

[14] The interested reader can check the AMICO demo `http://www.npaci.edu/DICE/AMICO/Demo/` for a "massive" application of DTD-based browsing.

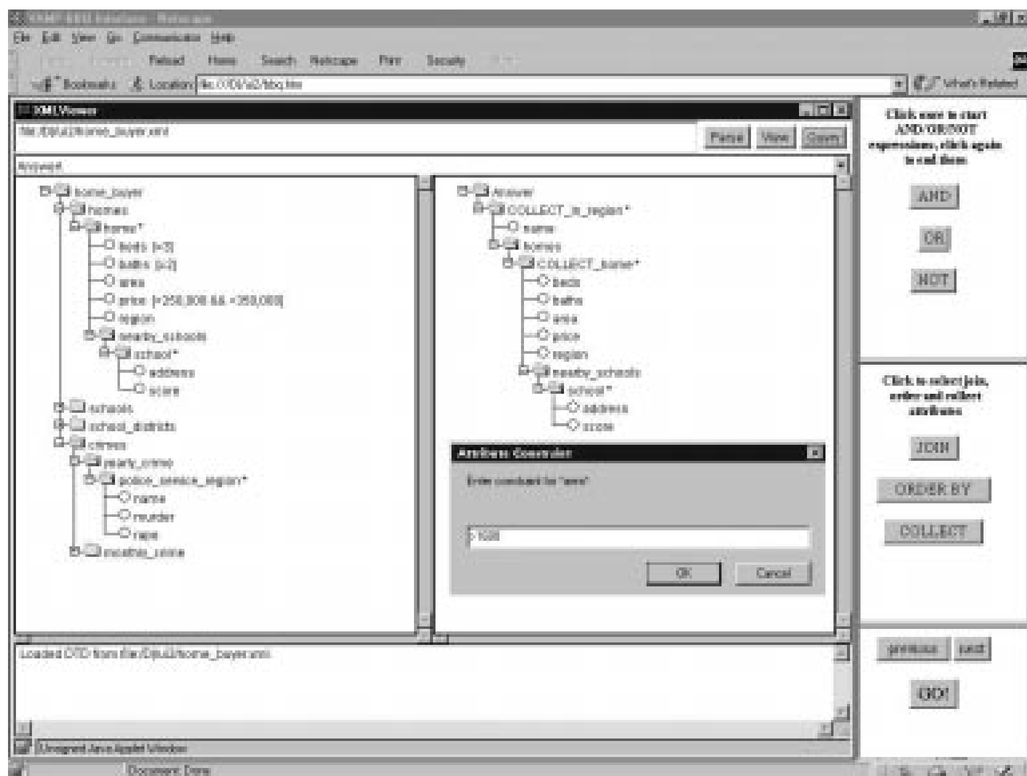


Figure 3. The DTD-driven BBQ GUI

Initially the condition window shows the root element `home_buyer` os the mediator view. Now the user can successively click on subelements of `home_buyer` thereby exposing as much of the DTD structure as necessary. By checking an element the user specifies that the corresponding element must be present in the data. Additional constraints can be attached to attributes, e.g., `>1600` for the `area` attribute. Joins can be expressed by linking the corresponding attributes. Here we can relate home and crime data by joining on the region.

Once the condition has been defined the query output is constructed. To this end the user drags subtrees from the condition window and drops them into the answer window. Additionally the output may be restructured and new elements may be created (e.g., the name `region`).

A newer version of BBQ makes apparent to the user even more structural details of the DTD; for example, which elements are "disjuncted", which ones are optional etc.

6.2 Query Optimization

The MIX mediator optimizes queries using the constraints that DTDs impose on the XML data. First it prunes conjuncts that are unsatisfiable with respect to the source DTDs.[15]

Second, it employs a DTD-based conjunct simplification. We illustrate the benefits of this optimization in the case of queries on fusion views, i.e., views that construct objects that aggregate information from multiple sources [PAGM96].

6.2.1 Warm Up

As we discussed in the list inference Section, the satifiability checking procedure, let us call it $sat(c, d)$, can decide whether the condition conjunct c is unsatisfiable when evaluated on documents that satisfy the DTD d. The optimizer uses sat to remove unsatisfiable conjuncts from a query plan.

EXAMPLE 6.1 Consider the following view (V10) and the query (Q11) applied on the view. We took the freedom to use two XMAS constructs we have not explained insofar; `UNION` has the obvious meaning while "_" stands for the disjunction of all element names.

```
(V10) people = SELECT P
               WHERE root.department.professor P
               UNION
               SELECT G
               WHERE root.department.gradStudent G


(Q11) answer = SELECT T
               WHERE root.people._ T, T.teaches
```

[15] Admittedly, the user queries in MIX have small chances of being unsatisfiable since the BBQ system leads the user in formulating satisfiable queries. However, the composition of the query with the view very likely leads to unsatisfiable conjuncts, as we will see in the examples.

The MIX mediator composes the query with the view and creates the following composition (Q12). Then the optimizer matches each conjunct against the source DTD and deduces, using *sat*, that the second union operand of (Q12) is unsatisfiable and hence it can be eliminated.

```
(V12) answer = SELECT P
               WHERE root.department.professor P, P.teaches
               UNION
               SELECT G
               WHERE root.department.gradStudent G, G.teaches
```

It is interesting that DTDs can prune queries that cannot be pruned with dataguides [GW97] or graph schemas [BDFS97]. In particular, dataguides and graph schemas encode only the existing paths of a semistructured database. For example, they can encode that there is a path $x.y$ and a path $x.z$ but they cannot encode whether elements named x always have both y and z children versus having only y or only z, or occasionally having both of them and occasionally just one.

6.2.2 The DTD-based Simplification and its use in Fusion Queries

The mediator query plans often contain conditions that are redundant given specific DTDs. The DTD-based conjunct simplification described next combines a novel *DTD-based unification* of variables (and corresponding paths) along with *conjunction minimization* in order to remove such redundant conditions. The essence of the optimization is similar to the one achieved by chasing functional dependencies; namely, the algorithm uses constraints – in this case the structural constraints expressed by the DTD – in order to unify variables or map variables to constants and, eventually, decide that some conditions are redundant.

Such an optimization is particularly useful when processing queries on *fusion* views [PAGM96], i.e., views that integrate multiple objects, which refer to the same real-world entity, into one fused view object. We first present an example where a fusion mediator benefits from the DTD-based simplification. Then we describe the novel part of the algorithm, namely the DTD-based unification part. (We only give literature references for the minimization part.)

EXAMPLE 6.2 The following fusion view collects publication information and fuses papers with the same title into a single object. For simplicity, we assume that the view accesses only one source – as opposed to the more typical case where data from multiple sources are fused together. If we had to fuse more than one sources we would first create an intermediate view that is the union of all sources and then we would proceed with fusing the entries using the view below.

```
(V13) fused_view = SELECT <fused_paper>
                              S
                          <fused_paper> GROUPBY {T}
```

```
WHERE root.department.professor.publication P, P._ S, P.title=T
```

The variable P binds to specific publication subobjects of a source conforming to the DTD (D1). The variable S binds to any subobject of P and T binds to the value of the title subobject. Then the GROUPBY clause creates one fused_paper for every title binding of T. This fused_paper contains all bindings of S that correspond to this T.

Notice that the view (V13) is insensitive to the potential semistructured-ness of the publications; the variable S carries to the view every publication subobject. Indeed, the source may add new fields into the publications without disrupting the operation of the view. This ability to integrate without complete knowledge of schema has been recognized as an important feature in many mediation and semistructured data projects [AQM+97, PAGM96, DFF+]. However,views that integrate without complete knowledge of schema cause inneficiency in query processing, as it is shown below.

EXAMPLE 6.3 Now consider the following query (Q14) that retrieves all papers with a title that contains the word "database".

```
(Q14) answer =SELECT F
             WHERE root.fused_view.fused_paper F,
                   F.title=T, T LIKE '%database%'
```

The mediator composes the view (V13) and the query (Q14) into the following query (Q15) that refers directly to the sources. The query finds the titles T of publications that have a title D containing "database".

```
(Q15) answer = SELECT <fused_paper>
                          S
                      <fused_paper> GROUPBY {T}
      WHERE root.department.professor.publication P, P._ S, P.title=T
            root.department.professor.publication P', P'.title=T,
            P'.title=D, D LIKE '%database%'
```

Notice that since the mediator has not inspected the DTD yet it does not know that each paper has exactly one title – the one that contains "database". Hence the mediator is forced to perform the needless join on T (i.e., P.title=T=P'.title)[16] while it is sufficient to simply select the documents P where the title contains the word "database".

The inneficiency of the above example is removed by the cooperation of our DTD-based unification algorithm with a minimization algorithm, as follows.

EXAMPLE 6.4 First the DTD-based unification *unifies* T and D because P' has only one title according to the DTD. After the unification the query becomes

[16] This problem had been spotted in [PAGM96] as well. This is the first time a solution to it is proposed.

```
(Q16) composed_query = SELECT <fused_paper>
                                  S
                              <fused_paper> GROUPBY {T}
      WHERE root.publication P, P._ S, P.title=T
            root.publication P', P'.title=T, T LIKE '%database%'
```

Next the mediator minimizes the query by removing the conditions

```
root.publication P', P'.title=T
```

The minimization of semistructured queries is based on the obvious direct application of containment algorithms, as in [Ull89]. [PVa] covers containment of semistructured queries and we do not discuss it any further here.[17]

Specialized DTD-based Variable Unification The DTD-based unification finds which variables of the conditions always have the same bindings, given a specialized DTD. The algorithm performs a special evaluation of the path conditions on a labeled graph that represents the schema's structure. We first describe this graph and then we proceed with the evaluation.

The labeled *schema tree* $s(r)$ of a regular expression r is constructed as follows. Note that for brevity we describe first the construction of schema trees and schema graphs where the labels are on the nodes. At the very last step we move the label of each node n to all the edges that are incoming to n.

- if r is the tagged symbol n^T then $s(r) = n^T$
- if r is of the form $r_1, \ldots, r_n, n > 1$ then $s(r) = \mathbf{SEQ}(s(r_1), \ldots, s(r_n))$
- if r is of the form $r_1 | \ldots | r_n, n > 1$ then $s(r) = \mathbf{ALT}(s(r_1), \ldots, s(r_n))$
- if r is of the form $r'?$ then $s(r) = s(r')$[18]
- if r is of the form $r*$ then $s(r) = \mathbf{STAR}(s(r'))$

Essentially, $s(r)$ is the parse tree of the regular expression r.

Next, the *simple schema graph* $s(d)$ of a specialized DTD d is a rooted labeled graph constructed as follows:

- For every definition $\langle n^T : r \rangle$ include in the graph the schema tree $s(r)$.
- For every leaf node n^T of a schema tree draw an edge from n^T to the schema tree corresponding to $\langle n^T : r \rangle$.

[17] Indeed [PVa] covers more general cases than what is needed above.

[18] The fact that the presence of r' is optional does not affect our algorithms.

- Include a node labeled *root*(d) and draw an edge from it to the schema tree corresponding to $\langle root(d) : r \rangle$.

Then, given a list of path conditions

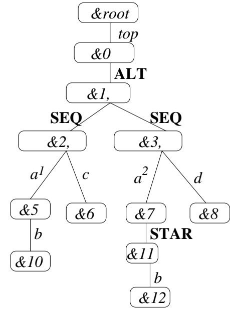
$$root.p_1 X_1, \ldots, [root|X_j].p_n X_n$$

we construct the *extended schema graph* by replicating every subtree of a **STAR** node (another) $n-1$ times.

Finally, we introduce a node with id *root*, we draw an edge from *root* to the top name of DTD, which is labeled *root*(d), and finally we move the label l of each node n to all the edges that are incoming to n.

EXAMPLE 6.5 The specialized DTD (D17) results into the schema graph (G18). Note that we have also denoted the node id's, starting with `&` in order to be able to refer to the nodes in the rest of the example.

(D17)$\langle top : a^1, c?|a^2, d \rangle$
$\langle a^1 : b \rangle$
$\langle a^2 : b* \rangle$

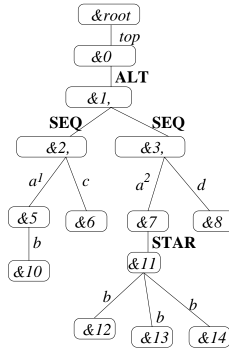


(G18)

Given the the list of conditions (i.e., the `WHERE` clause)

$$root.a.b\ X, root.a.b\ Y, root.c\ C$$

we derive the extended schema graph (G19), where the subtree &12 of **STAR** is replicated another two times, as &13 and &14.



(G19)

The DTD-based unification algorithm decides whether two variables of the conditions can be unified because for every database and every valid valuation of the list of path conditions they always bind to the same node. The unification uses the extended schema graph, which, in a sense, stands as a representative of all possible databases. Two variables X and Y are unified if every valuation of the paths on the schema graph results on X and Y binding to the same node of the schema graph. However, this idea will have to be extended to deal with the special meaning of the **ALT** nodes.

EXAMPLE 6.6 Consider the conditions

(C20) $root.a.b\ X, root.a.b\ Y, root.c\ C$

evaluated on databases conforming on the DTD (D17). Clearly X and Y can be unified because the presence of the c condition forces the a object to have exactly one b subobject – the one that binds to both X and Y.

The unification algorithm reaches this result by matching the path conditions with the extended schema graph and producing the list of valid bindings. In our example, the condition $root.c\ C$ matches with the path &0, &1, &2, &6 and the conditions $root.a.b\ X$ and $root.a.b\ Y$ match with &0, &1, &2, &5, &10.

X	Y	C
&10	&10	&6

The above list of bindings indicates that X and Y always bind to the same node. Note that the binding ($X \mapsto$ &12, $Y \mapsto$ &13, $C \mapsto$ &6) is not valid because it uses two nodes (namely the &6 and the &12) that rest on different subtrees of the same **ALT** node.

Since X and Y always bind to the same node of the schema graph we deduce that we can unify X and Y and hence the list of conditions (C20) can be reduced into

$$root.a.b\ X, root.a.b\ X, root.c\ C$$

and after the obvious minimization step into

$$root.a.b\ X, root.c\ C$$

The following example illustrates how the replication of the schema trees that hang from **STAR** nodes prevents us from unifying variables that bind to different instances of identical element structures.

EXAMPLE 6.7 Consider the following condition list applied on a source conforming to the DTD (D17).

$$root.a.b\ X, root.a.b\ Y, root.d\ D$$

Intuitively, X and Y must not be unified because they can bind to different subobjects of a nodes that conform to the a^2 definition. This intuition is captured by running the conditions on the corresponding schema graph (SG19): There is the valuation ($X \mapsto$ &12, $Y \mapsto$ &13, $D \mapsto$ &8) that maps X and Y to different nodes.

Conjecture Given a specialized DTD d and a list of path conditions

$$root.p_1 X_1, \ldots, [root|X_j].p_n X_n$$

two variables X_i and X_j can be unified if and only if they bind to the same node for every valid binding of the conditions of the extended schema graph.

[[[First we prove that if there is a binding b such that the variables X_i and X_j bind to different nodes of the extended schema graph then there is a database D conforming to d such that when the conditions are evaluated on D X_i and X_j bind to different nodes of D; and hence, we cannot unify X_i and X_j. Constructing such a database D is straightforward: First prune all the subtrees of **ALT** nodes that have not been used in the binding. Then remove the **SEQ**, **ALT** and **STAR** nodes by connecting their parents with their children. Finally turn the graph into a tree by expanding the cycles. Since our path conditions do not involve recursion we need to expand each cycle at most m times where m is the size of the largest path expression. So, insofar we have proved that if there is a binding that maps two variables X_i and X_j to two different nodes of the extended schema graph then X_i and X_j cannot be unified. Or, equivalently, if X_i and X_j can be unified then X_i and X_j have the same bindings when evaluated on the extended schema graph. <==

Now, we prove the reverse direction; namely, if X_i and X_j have the same bindings on the extended schema graph of d then they can be unified, i.e., they will have the same bindings on every database D conforming to d.]]]

7 Related Work

Our work on the inference and use of DTDs is closely related to three areas of industrial and academic research and development. First it relates to works in mining and using descriptions of semistructured databases. Second it relates to recent works on DTD inference and conformance. Finally it relates to recent industrial developments in XML "schemas".

Mining and Using Descriptions of Semistructured Databases

- [GW97] introduces dataguides as OEM objects and studies problems of mining dataguides from data. It also describes the use of dataguides in query formulation and in optimization. The main use in optimization is the discovery of unsatisfiable paths. 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. (Section 6.2 describes the difference between dataguides and DTDs in the elimination of unsatisfiable conditions.) However dataguides do not require the same type name to define the same type, so in this respect dataguides are similar to s-DTDs.

- [BDFS97] defines graph schemas and studies their properties. The graph schemas are similar to dataguides but can include unary formulas on their edges. [BDFS97] specifies that graph schemas are closed under application of UnQl queries [BDHS96]. As in the case of dataguides, graph schemas cannot capture order, cardinality and constraints on the siblings.

- [FS98] 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 when evaluated on the non-constrained database. They present algorithms for rewriting path queries into equivalent queries using state extents and they also present a polynomial approximation to the rewriting algorithm.
- [NUWC] 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.

XML "schemas" The shortcomings of DTDs in describing semistructured data have been recently addressed by both the industrial and the academic community. In particular, [BM99] provides a semistructured data "schema" formalism that encompasses specialized DTDs. This formalism was developed in parallel with specialized DTDs in the conference version of this paper [PV99]. Furthermore, the standard recommendation *XSchemas* [B$^+$] recently introduced a form of specialization by allowing an element x to be associated with different content types depending on its context. To the best of our knowledge, [PV99] was the first to highlight the importance of specialized DTDs in mediation.

DTD Inference and Conformance [MS99] solves the view conformance problem for a limited class of queries. The inputs to the conformance algorithm are a source "schema", a view "schema", and a view definition. The goal of the algorithm is to decide whether the view actually conforms to the provided view schema.

[PVb] has generalized our DTD inference results in three directions. First it considers recursive path expressions. Second, order conditions are expressed using horizontal regular path expressions. Note that horizontal path expressions specify a total order of an element's subobjects – unlike the precedence relationships used in this paper that introduce a partial order. Third, it provides inference for queries involving `GROUPBY`.

[PVb] shows that for the class of pick-element queries that it considers there is always a tight *context-free* specialized DTD, where a context-free DTD is one where the context of a specialized type is described using a context-free language – as opposed to the regular ones we've been using in MIX. Furthermore, it shows that one may not be able to deliver a tight specialized DTD for queries involving `GROUPBY`. However, recently [MSV] has shown that conformance can be decided even for queries involving `GROUPBY`.

Acknowledgements We thank Vincent Chu, who has built the original BBQ version, and Kevin Munroe who has built the current version [[[ ]]] . We also thank SDSC's Amarnath <== Gupta and Bertram Ludaescher who have built the home buyer and the AMICO applications using the MIX mediator system. Victor Vianu has provided critical insights on the definition of specialized DTDs and the generalization of the algorithm to non-selection queries. Finally,

we thank the entire MIX team and the students of UCSD's CSE291 class for interesting discussions.

References

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The LOREL query language for semistructured data. *International Journal on Digital Libraries*, 1(1), 1997.

[B+] D. Beech et al. XML schema part 1: Structures. Available at `http://www.w3.org/TR/xmlschema-1/`.

[B+99] C. Baru et al. Xml-based information mediation with mix, 1999.

[BDFS97] P. Buneman, S. Davidson, M. Fernandez, and D. Suciu. Adding structure to unstructured data. In *Proc. of the International Conference on Database Theory*, 1997.

[BDHS96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. ACM SIGMOD*, 1996.

[BM99] Catriel Beeri and Tova Milo. Schemas for integration and translation of structured and semi-structured data. In *Int'l. Conf. on Database Theory*, 1999.

[BPSM] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation. Latest version available at `http://www.w3.org.TR/REC-xml`.

[CDSS98] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion! In *Proc. ACM SIGMOD Conf.*, 1998.

[DFF+] A. Deutch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. Submission to W3C. Latest version available at `http://www.w3.org/TR/NOTE-xml-ql`.

[FS98] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proc. of the International Conference on Data Engineering*, 1998.

[GW97] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proc. VLDB*, 1997.

[MP] K. Munroe and Y. Papakonstantinou. BBQ: A visual interface for integrated browsing and querying of XML. Available at `http://www.db.ucsd.edu/publications/BBQ.pdf`.

[MS99] Tova Milo and Dan Suciu. Type inference for queries on semistructured data. In *Proc. PODS Conf.*, 1999.

[MSV] T. Milo, D. Suciu, and V. Vianu. Typechecking for xml transformers. Unpublished.

[NUWC] S. Nestorov, J. Ullman, J. Wiener, and S. Chawathe. Representative objects: Concise representations of semistructured, hierarchial data. In *Proceedings of 13th International Conference on Data Engineering*.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proc. VLDB Conf.*, 1996.

[PVa] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data.

[PVb] Y. Papakonstantinou and V. Vianu. Dtd inference for views of XML data.

[PV99] Y. Papakonstantinou and P. Velikhov. Enhancing semistructured data mediators with document type definitions. In *Proc. ICDE Conf.*, 1999.

[QRS+95] D. Quass, A. Rajaraman, S. Sagiv, J. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *Proc. DOOD*, pages 319–44, 1995.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies*. Computer Science Press, New York, NY, 1989.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

A The List inference algorithm

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

function InferList(v, p, c, t, d)
  if c is tree condition $c_1 \ldots c_n$
    for each c_i such that c_i is not p_1
      if Tighten(null, t, c_i, d, {}) results in fail
        return ϵ
      else if Tighten results in *satisfiable*
        t,d ← Tighten(null, t, c_i, d, {})
      else if Tighten results in *valid*
        t,d ← Tighten(null, t, c_i, d, {})

  if Tighten(null, t, p_1, d, {}) results in fail
    return ϵ
  t ← refine(t, p_1)
      if refine fails, return ϵ
  t ← project(t, p_1, d)
  if all specializations resulted in *valid*
      t ← substitute $d'[p_1]$ for p_1 in t
  else
      t ← substitute $(d'[p_1])?)$ for p_1 in t
  return InferList(v, $p_1 \ldots p_n$, c/p_1, v, t, d)

function project(t, $n(T)$, d)
  if $t = n$ or $t = n(T)$
    return $n(T)$
  if $t = n(T')$ and $T' \neq T$
    if Tighten(null, t, $n(T)$, d, {}) fails

```
    return ϵ
  else
    return n(T)
if t is (g)∗
  return ( project(g, n(T), d) )*
if t is r_1, ..., r_m
  return project(r_1, n(T), d), project((r_2, ..., r_m), n(T), d)
if t is r_1|...|r_m
  return project(r_1, n(T), d) | project((r_2|...|r_m), n(T), d)
```