Information preserving XML schema embedding

Citation for published version:
Fan, W & Bohannon, P 2008, 'Information preserving XML schema embedding', *ACM Transactions on Database Systems*, vol. 33, no. 1. https://doi.org/10.1145/1331904.1331908

Digital Object Identifier (DOI):
10.1145/1331904.1331908

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Early version, also known as pre-print

Published In:
ACM Transactions on Database Systems

Information Preserving XML Schema Embedding

WENFEI FAN
University of Edinburgh and Lucent Technologies
and
PHILIP BOHANNON
Yahoo! Research

A fundamental concern of data integration in an XML context is the ability to *embed* one or more source documents in a target document so that (a) the target document conforms to a target schema and (b) the information in the source documents is *preserved*. In this paper, information preservation for XML is formally studied, and the results of this study guide the definition of a novel notion of *schema embedding* between two XML DTD schemas represented as graphs. Schema embedding generalizes the conventional notion of graph similarity by allowing an edge in a source DTD schema to be mapped to a path in the target DTD. Instance-level embeddings can be derived from the schema embedding in a straightforward manner, such that conformance to a target schema and information preservation are guaranteed. We show that it is NP-complete to find an embedding between two DTD schemas. We also outline efficient heuristic algorithms to find candidate embeddings, which have proved effective by our experimental study. These yield the first systematic and effective approach to finding information preserving XML mappings.

Categories and Subject Descriptors: H.2.5 [**Database Management**]: Heterogeneous Databases—*Data translation*

General Terms: Algorithms, Design, Languages, Management, Performance, Theory

Additional Key Words and Phrases: Data transformation, information integration, information preservation, schema embedding, schema mapping, XML, XSLT

W. Fan is supported in part by ERSRC GR/S63205/01, ERSRC GR/T27433/01, BBSRC BB/D006473/1 and NSFC 60228006.

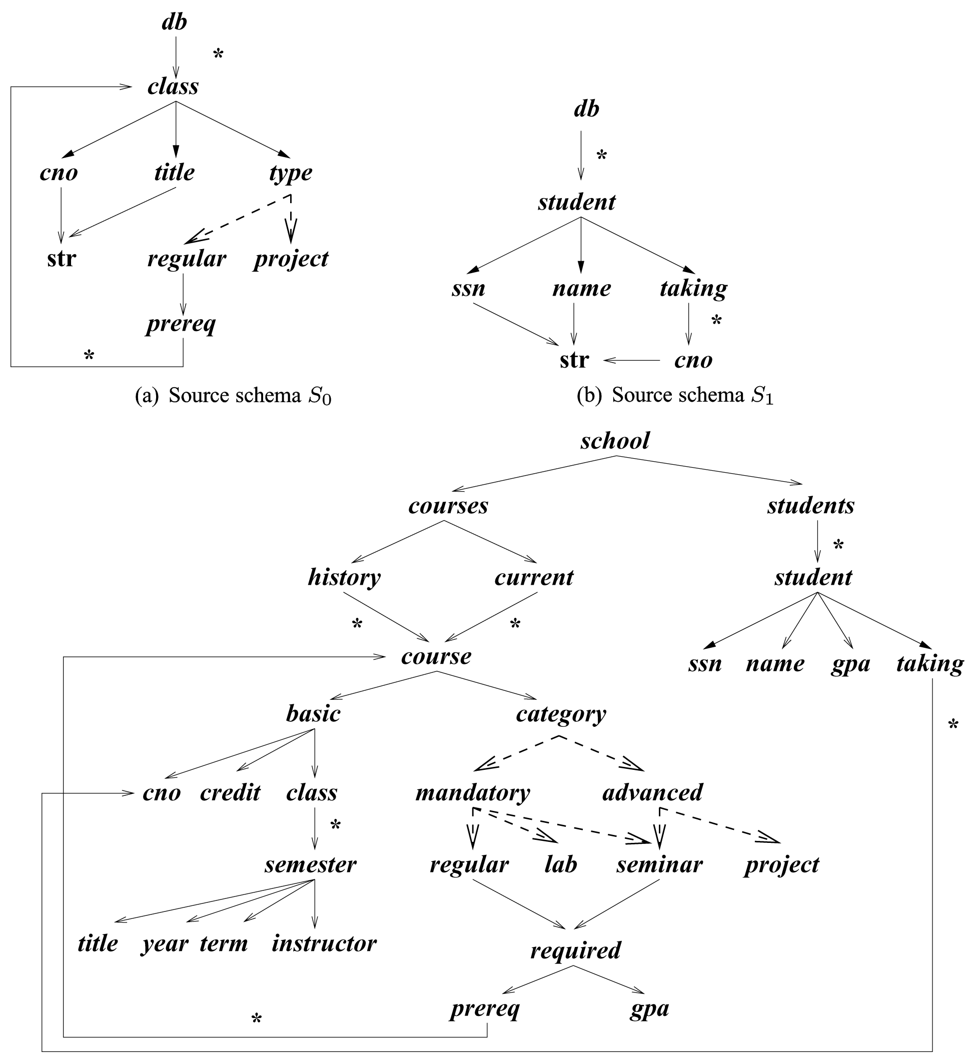
1. INTRODUCTION

A central technical issue for the exchange, migration and integration of XML data is to find mappings from documents of a source XML (DTD) schema to documents of a target schema. In practice such a mapping, referred to as an XML *mapping*, often needs to (1) guarantee *type-safety*, that is, the target document produced by the mapping should conform to the target schema; and (2) *preserve information*, that is, the target documents should not lose original information of the source data. Criteria for *information preservation* include: (1) *invertibility* [Hull 1986]: can one recover the source document from the target? and (2) *query preservation*: for a certain XML query language, can all queries on source documents in that language be answered on target documents by queries in the same language? We now illustrate these concepts with an example.

Example 1.1. Consider two source DTDs S_0, S_1 and a target DTD S represented as graphs in Figure 1 (we omit the str–PCDATA– child under *cno, credit, title, year, term, instructor, gpa* in Figure 1(c)). A document of S_0 contains information of *class*es taught at a school, and a document of S_1 contains *student* data of the school. The user wants to map the document of S_0 and the document of S_1 to a single instance of S, which is to collect data about *courses* and *students* of the school in the last five years. Here we use edges of different types to denote various constructs of a DTD, namely, *solid edges* for a concatenation type (a unique occurrence of each child), *dashed edges* for disjunction (one and only one child), and *star edges* (edges labeled '$*$') for Kleene star (zero or more child).

Here type safety requires that *school* documents produced by integrating instances of S_0 and S_1 are guaranteed to conform to the target DTD S. Invertibility asks for the ability to reconstruct the original *class* and *student* documents from an integrated *school* document, while query preservation requires the ability to answer XML queries expressed in a language $\mathcal{L}$ (e.g., XPath) and posed on *class* and *student* documents by equivalent queries that are expressed in the same language $\mathcal{L}$ but are posed on the *school* document.

Type safety is typically necessary since it concerns whether or not XML mappings make sense. In many applications one also wants XML mappings to be information preserving. For example, in data exchange between two schemas or in data migration from one schema to another [Lenzerini 2002], one often wants to reconstruct the source document and thus requires the XML mapping to be invertible. As observed in Fagin [2006], inverse mappings are also useful in developing new mappings via mapping composition. It is also common in practice that users care more about query answerability than the ability to restore the original documents. As an example, in a P2P system [Kementsietsidis et al. 2003; Halevy et al. 2004], for any query Q posed on a local document residing in one peer, one wants to be able to find the same query answer via a query Q' posed on another peer; furthermore, while Q' may be different from (yet equivalent to) Q, Q' and Q should be expressible in the *same* query language L, for example, XPath. In other words, the XML mappings from data at one peer to another peer should be query preserving. Note that in query rewriting it is also required that the rewritten query should be expressible in the same language



(a) Source schema S_0

(b) Source schema S_1

(c) Target schema S

Fig. 1. Example: source and target schemas.

as that of the original query [Halevy 2000; Lenzerini 2002]. For all the reasons that this requirement is important for query rewriting, it is also needed for query preservation. In particular, the language in which the original query Q is given may be only language supported by the system or is the language that the user wants to use. Furthermore, the user should not be penalized by paying the higher price for evaluating and optimizing queries in a richer language than that of the original query, for example, XQuery.

While one can certainly define XML mappings in a query language such as XQuery or XSLT, such queries may be large and complex, and as a result time-consuming to construct by hand. Worse still, type safety may be hard to check for mappings defined in XQuery or XSLT [Alon et al. 1995]. In practice one often wants a systematic methods to define XML mappings that are *guaranteed* to be type safe. When it comes to information preservation, a number of fundamental questions are open. Can one determine whether an XML mapping is information preserving? Is there an efficient method to find information preserving XML mappings, or better still, to define XML mappings that are guaranteed to be invertible and/or query preserving w.r.t. a popular XML query language, for example, XPath?

While type safety and information preservation are clearly desirable, an additional feature is the ability to map documents of DTDs that have *different structures*. A given source DTD may differ in structure from a desired target DTD. This is commonly encountered in data integration, where the target DTD needs to accommodate data from *multiple sources* and thus may not be similar to any of the sources in structure; in Figure 1, for example, the *school* DTD is quite different from the *class* and *student* DTDs.

Background. While information preservation has been studied for traditional database transformations [Abiteboul and Hull 1988; Hull 1986; Miller et al. 1993, 1994], to our knowledge, no previous work has considered it for XML mappings. In fact, a variety of tools and models *have* been proposed for finding XML mappings at schema- or instance-level [Doan et al. 2001; Madhavan et al. 2001; Melnik et al. 2002; Melnik et al. 2003; Miller et al. 2001; Milo and Zohar 1998]; however, none has addressed invertibility and query preservation for XML. Most tools either focus on *highly similar* structures, or adopt a strict graph similarity model like simulation [Abiteboul et al. 2000] to match structures, which is incapable of mapping DTDs with *different structures* such as those shown in Figure 1, and can ensure neither invertibility nor query preservation w.r.t. XML query languages. Another issue is that it is unclear that mappings found by some of these tools guarantee type safety when it comes to complex XML DTDs.

Contributions. In response to the practical need, we investigate fundamental questions associated with XML mappings. Furthermore, we propose a method for defining XML mappings that are guaranteed to be both type safe and information preserving. Here information preservation will be characterized by both invertibility and query preservation. More specifically, this work makes the following contributions.

First, as criteria for information preservation we revisit the notions of invertibility and query preservation Abiteboul and Hull 1988; Hull 1986; Miller et al. 1993, 1994] for XML mappings. While the two notions coincide for relational mappings w.r.t. relational calculus [Hull 1986], we show that they are in general different for XML mappings w.r.t. XML query languages. Furthermore, we show that it is undecidable to determine whether or not an XML mapping defined in a small fragment of XQuery or XSLT is information preserving.

Second, to cope with the undecidability result, we introduce an XML mapping framework based on a novel notion of schema embeddings. A *schema embedding* is a natural extension of graph similarity in which an edge in a source DTD schema may be mapped to a *path*, rather than a single edge, in a target DTD. For example, the source DTDs S_0 and S_1 of Figure 1 can both be embedded in

S, while there is no sensible mapping from them to S based on graph similarity. From a schema embedding, an instance-level XML mapping can be directly produced that has all the properties mentioned above. In particular, such mappings are *invertible*, *query preserving w.r.t. regular XPath* (an extension of XPath introduced in Marx [2004]), and *ensure type safety*. As with schema-mapping techniques for other data models, by automatically producing this mapping the user is saved from the burden of writing and type-checking a complex mapping query. Moreover, we show that the *inverse* and *query translation functions* for the mapping are efficient.

Third, we provide an algorithm to translate queries posed against a source schema into queries against the document as embedded in the target schema. In order to accomplish this translation in low-polynomial time, we introduce a mild augmentation of nondeterministic finite state automata to represent regular XPath queries; based on this notion we then develop a schema-directed translation algorithm producing an automaton for the target schema. While automaton may itself be translated into regular XPath, this translation subsumes the translation of finite-state automata to regular expressions, an EXPTIME-complete problem [Ehrenfeucht and Zeiger 1976].

Fourth, we provide algorithms to compute schema embeddings. We show that it is NP-complete to find an embedding between two DTDs, even when the DTDs are nonrecursive. Thus practical algorithms for finding embeddings are necessarily heuristic. We have implemented our algorithms and conducted an experimental study based on mapping schemas taken from real-life and benchmark sources to copies of these schemas with varying amounts of introduced noise. These experiments verify the accuracy and efficiency of our heuristics on schemas up to a few hundred nodes in size, supporting the practical applicability of schema embedding. We omit the details of the algorithms and experimental results due to the lack of space, but we suggest the reader consult [Bohannon et al. 2005].

Schema embeddings are a promising tool for automatically computing information preserving XML mappings, and are particularly suited for common information integration cases where the target schema is more general and thus more complex than the source. To the best of our knowledge, this work is the first to study information preservation in the generic XML context, and it yields a systematic and effective approach to defining and finding type safe and information preserving XML mappings.

Organization. The remainder of the article is organized as follows. Section 2 reviews DTDs and XPath, and revisits invertibility and query preservation for XML mappings. Section 3 investigates basic properties of invertibility and query preservation, establishing equivalence, separation and complexity results for XML mappings. Section 4 defines the notion of schema embedding and shows that schema embedding guarantees information preservation and type safety. Section 5 shows that the problem of finding schema embedding is intractable, and outlines our algorithms for computing schema-embedding candidates. Related work is discussed in Section 6, followed by topics for future work in Section 7.

2. DTDS, XPATH, INFORMATION PRESERVATION

In this section we review DTDs and (regular) XPath, and revisit the notions of information preservation [Hull 1986; Miller et al. 1994] for XML.

2.1 DTDs

To simplify the discussion, consider DTDs of the form (E, P, r), where E is a finite set of *element types*; r is a distinguished type in E, called the *root type*; P defines the element types: and for each A in E, $P(A)$ is a regular expression of the form:

$$\alpha \quad ::= \quad \mathsf{str} \mid \epsilon \mid B_1, \ldots, B_n \mid B_1 + \cdots + B_n \mid B^*$$

where str denotes PCDATA, ϵ is the empty word, B is a type in E (referred to as a *child* of A), and '+', ',' and '$*$' denote *disjunction* (with $n > 1$), *concatenation* and the *Kleene star*, respectively. We refer to $A \rightarrow P(A)$ as the *production* of A. Note that this form of DTDs does not lose generality since any DTD S can be converted to S' of this form (in linear time) by introducing new element types, and (regular) XPath queries on S can be rewritten into equivalent (regular) XPath queries on S' in PTIME [Benedikt et al. 2005].

Schema Graphs. We represent a DTD S as a labeled graph G_S, referred to as the *graph* of S. For each element type A in S, there is a unique node labeled A in G_S, referred to as the A *node*. From the A-node there are edges to nodes representing child types in $P(A)$, determined by the production $A \rightarrow P(A)$ of A. There are three different types of edges indicating different DTD constructs. Specifically, if $P(A)$ is $B_1, \ldots, B_n$ then there is a *solid edge* from the A node to each B_i node, referred to as an AND edge; it is labeled with a position k if B_i is the k-th occurrence of a type B in $P(A)$ (the label can be omitted if B_i's are distinct). If $P(A)$ is $B_1 + \cdots + B_n$ then there is a *dashed edge* from the A node to each B_i node (w.l.o.g. assume that B_i's are distinct in disjunction), referred to as an OR edge. If $P(A)$ is B^*, then there is a *solid* edge with *a* "$*$" *label* from the A node to the B node, referred to as a STAR edge. When it is clear from the context, we shall use the DTD and its graph interchangeably, both referred to as S; similarly for A element type and A node.

A DTD is said to be *recursive* if and only if its graph is *cyclic*.

For example, Figure 1 shows graphs representing three DTDs, where Figures 1(a) and 1(c) depict recursive DTDs.

An XML *instance* T of a DTD S is an ordered, node-labeled tree that conforms to S. That is, (1) there is a unique node, the *root*, in T labeled with r; (2) each node in T is labeled either with an E type A, called an A *element*, or with str, called a *text node*; (3) each A element has a list of children of elements and text nodes such that their labels are a word in the regular language defined by $P(A)$; and (4) each text node carries a string value (PCDATA) and is a leaf. We denote by $\mathcal{I}(S)$ the set of all instances of S.

Two XML trees T_1 and T_2 are said to be *equal*, denoted by $T_1 = T_2$, if T_1 and T_2 are isomorphic by an isomorphism that is the identity on string values. More specifically, for a node n_1 in T_1 and a node n_2 in T_2, we say that n_1 and n_2 are equal, denoted by $n_1 = n_2$, if the following conditions are satisfied. (1) If both n_1 and n_2 are text nodes, then they carry the same string value. (2) If both n_1 and n_2 are elements, then they are both labeled with the same tag and moreover, their children are pairwise equal, *i.e.,* their children are lists $[v_1, \ldots, v_k]$ and $[u_1, \ldots, u_k]$, respectively, and $v_i = u_i$ for all $i \in [1, k]$. We say that $T_1 = T_2$ if $r_1 = r_2$, where r_1 and r_2 are the root nodes of T_1 and T_2, respectively. Intuitively, if T_1 and T_2 are equal then they have the identical structure carrying the same string values (observable values).

Assume a countably infinite set $\mathcal{U}$ of *node ids*. For each XML node v in T, we assume that v is associated with a distinct node id $id(v) \in \mathcal{U}$, and denote the set of all node ids in T by $\mathsf{dom}(T)$. Note that a text node is also associated with a node id and it carries PCDATA.

A DTD S is *consistent* if it has no useless element types, that is, each element type of S appears in $\mathcal{I}(S)$. For standard context-free grammars (CFGs) it has been well studied how to drop all useless types from a CFG [Hopcroft and Ullman 1979], in quadratic time. Along the same lines we can convert any DTD S to a consistent S' in $O(|S|^2)$ time such that $\mathcal{I}(S') = \mathcal{I}(S)$. Thus in the sequel we only consider consistent DTDs.

2.2 XPath and Regular XPath

We consider a class of *regular* XPath queries proposed and studied in Marx [2004], denoted by $\mathcal{X}_R$ and defined as follows:

$$
\begin{aligned}
p &::= \epsilon \mid A \mid p/\mathit{text}() \mid p/p \mid p \cup p \mid p^* \mid p[q], \\
q &::= p \mid p/\mathit{text}() = \text{`}c\text{'} \mid \mathit{position}() = k \mid \neg q \mid q \wedge q \mid q \vee q,
\end{aligned}
$$

where ϵ is the empty path (*self*), A is a label (element type), '$\cup$' is the *union* operator, '/' is the *child-axis*, and $*$ is the Kleene star; q is a *qualifier*, p is an $\mathcal{X}_R$ expressions, k is a natural number, c is a string constant, and $\neg, \wedge, \vee$ are the Boolean negation, conjunction and disjunction operators, respectively. We shall use a special qualifier true, which always holds and is definable in $\mathcal{X}_R$ (e.g., $[\mathsf{true}]$ can be defined as $[\epsilon]$).

An *XPath* [Clark and DeRose 1999] fragment of $\mathcal{X}_R$, denoted by $\mathcal{X}$, is defined by replacing p^* with $p//p$ in the definition above, where $//$ is the *descendant-or-self axis*.

A (regular) XPath query p is evaluated at a *context node* v in an XML tree T; its result, denoted by $v[\![p]\!]$, is the set consisting of (a) node ids in $\mathsf{dom}(T)$ of those nodes reachable via p from v, and (b) string values (PCDATA) if p contains a sub-query of the form $p'/\mathit{text}()$, e.g., when p is $A//C/\mathit{text}() \cup A/B$ [Clark and DeRose 1999; Marx 2004]. We use $p(T)$ to denote $v[\![p]\!]$ if v is the root node of T. While node ids are non-printable, commercial systems typically provide APIs that allow one to define a function f such that (a) it maps node ids in $\mathsf{dom}(T)$ to observable values and (b) it behaves the same as the identity function on string values (e.g., Xerces [Xerces and Xalan], Galax [Siméon and Fernandez]).

2.3 Invertibility and Query Preservation

For XML DTDS S_1 and S_2, a *(data) instance mapping* $\sigma_d : \mathcal{I}(S_1) \to \mathcal{I}(S_2)$ is *invertible* if there exists an inverse σ_d^{-1} of σ_d such that for any XML instance $T \in \mathcal{I}(S_1)$, $\sigma_d^{-1}(\sigma_d(T)) = T$, where $f(T)$ denotes the result of applying a function (or mapping, query) f to T. In other words, the composition $\sigma_d^{-1} \circ \sigma_d$ is equivalent to the identity mapping *id*, which maps an XML document to itself.

For an XML query language $\mathcal{L}$, a mapping σ_d is *query preserving w.r.t.* $\mathcal{L}$ if there exists a computable function $\mathsf{Tr} : \mathcal{L} \to \mathcal{L}$ such that for any XML query $Q \in \mathcal{L}$ and any $T \in \mathcal{I}(S_1)$, $Q(T) = \mathsf{Tr}(Q)(\sigma_d(T))$, i.e., $Q = \mathsf{Tr}(Q) \circ \sigma_d$.

In a nutshell, invertibility is the ability to recover the original source XML document from the target document; query preservation w.r.t. $\mathcal{L}$ indicates whether *all* queries of $\mathcal{L}$ on any source T of S_1 can be effectively answered over $\sigma_d(T)$, that is, the mapping σ_d does not lose information of T when $\mathcal{L}$ queries are concerned.

The notions of invertibility and query preservation are inspired by (calculus) *dominance* and *query dominance* that were proposed in Hull [1986] for relational mappings and later studied in Abiteboul and Hull [1988] and Miller et al. [1993, 1994]. In contrast to query dominance, query preservation is defined w.r.t. a given XML query language that does not necessarily support query composition. Invertibility is defined for XML mappings and it only requires σ_d^{-1} to be a partial function defined on $\sigma_d(\mathcal{I}(S_1))$.

We say that a mapping $\sigma_d : \mathcal{I}(S_1) \to \mathcal{I}(S_2)$ is *information preserving* w.r.t. $\mathcal{L}$ if it is both invertible and query preserving w.r.t. $\mathcal{L}$.

A subtle issue arises from query preservation w.r.t. regular XPath. As mentioned earlier, when a regular XPath query Q is evaluated on an XML tree T, its result $Q(T)$ may contain non-printable node ids. This is analogous to the oid-observability problem associated with implicit oids in object-oriented databases [Abiteboul et al. 1995]. We refine the semantics of query preservation *w.r.t.* regular XPath as follows. With an XML mapping $\sigma_d : S_1 \to S_2$ we associate a (partial) node id mapping $\mathsf{idM}()$ that, given an XML instance T of the source schema, maps $\mathsf{dom}(\sigma_d(T))$ to $\mathsf{dom}(T)$ and it is the identity mapping on string values (furthermore, $\mathsf{idM}()$ also recovers the original tag of the node v in T if $\mathsf{idM}()$ maps a node from $\sigma_d(T)$ to v; we omit this to simplify the discussion). We say that σ_d is query preserving w.r.t. regular XPath if there exists a computable function Tr such that for any regular XPath query Q and any $T \in \mathcal{I}(S_1)$, $Q(T) = \mathsf{idM}(\mathsf{Tr}(Q)(\sigma_d(T)))$, *i.e.,* $\mathsf{idM}()$ recovers the original query result $Q(T)$ from $\mathsf{Tr}(Q)(\sigma_d(T))$. This assures that $Q(T)$ and $\mathsf{idM}(\mathsf{Tr}(Q)(\sigma_d(T)))$ return the same set of node ids (carrying their original tags in T). Assuming a countably infinite set $\mathcal{V}$ of observable values, then for *any* function f that maps nodes ids in $\mathsf{dom}(T)$ to $\mathcal{V}$ and maps string values to themselves, $f(Q(T)) = f(\mathsf{Tr}(Q)(\sigma_d(T)))$; that is, they yield the same set of observable values no matter what printable values are associated with those node ids via user-defined mapping function and system-provided APIs.

Example 2.1. Figure 2 depicts a source schema S_1 (on the left) and a target schema S_2 (on the right). A mapping $\sigma_d : \mathcal{I}(S_1) \to \mathcal{I}(S_2)$ is indicated by the arrows, while the inverses of the arrows denote the node id mapping $\mathsf{idM}()$. Intuitively, given any $T \in \mathcal{I}(S_1)$, an instance $\sigma_d(T)$ is constructed such that its

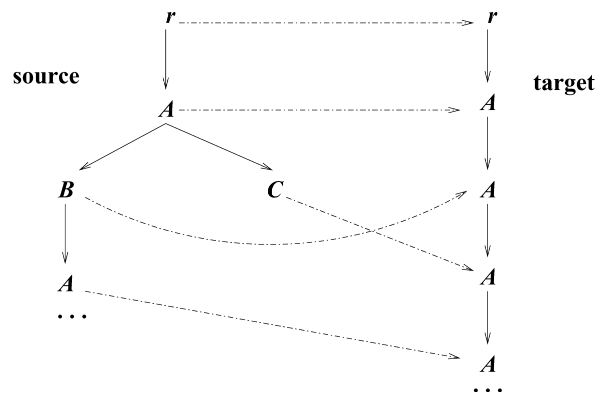


Fig. 2. An example XML mapping.

id mapping idM() maps (a) the id of the root r_2 of $\sigma_d(T)$ to the id of the root r_1 of T; (b) the id of the A child of r_2 to the A child of r_1; and inductively, (c) if the id of element v' in $\sigma_d(T)$ is mapped to the id of an A element v in T, then the ids of the child and the grandchild of v' are mapped to the ids of the B, C children of v, respectively, and the id of the great grandchild of v' is mapped to the id of the A child of the B child of v.

Consider an XPath query $Q = A/B$ posed at the root r_1 of T, and $Q' = A/A$ at the root r_2 of $\sigma_d(T)$. Then Q and Q' are equivalent *w.r.t.* idM(). Indeed, $Q(T)$ returns a single node id o, and $Q'(\sigma_d(T))$ returns a single node id o', where $\mathsf{idM}(o') = o$.

This semantics of query equivalence can be implemented in, for example, XSLT, as follows. Distinct ids can be generated for distinct nodes in T via, for example, the *generate-id()* function of XSLT, which returns a (printable) string value as the unique id of a node. As will be seen in Section 4.3, a mapping $\sigma_d()$ can be expressed in XSLT, which can be easily extended such that given T, $\sigma_d()$ also creates a unique id for each node in T and furthermore, associates the id of each node in T with the id of the corresponding node in $\sigma_d(T)$. This produces the definition of idM(). This can also be used to display ids in query result, that is, for each id in the query result, we can simply return its string representation generated by *generate-id()*, a printable string.

To simplify the discussion and due to the space constraint, in the sequel, we omit the construction of the id mapping idM() when it is clear from the definition of σ_d, and simply write $Q(T) = \mathsf{idM}(\mathsf{Tr}(Q)(\sigma_d(T)))$ as $Q(T) = \mathsf{Tr}(Q)(\sigma_d(T))$. By mapping a node v in T to a node v' in $\sigma_d(T)$, or copying v to v', we mean that we create a node v' such that idM() maps the id of v to the id of v'.

3. INFORMATION PRESERVATION

In this section we establish basic results for the separation and equivalence of the invertibility and query preservation of XML mappings, as well as the complexity of determining whether a given XML mapping is information preserving.

Invertibility and Query Preservation: Separation. It was shown [Hull 1986] that calculus dominance and query dominance are equivalent for relational mappings. In contrast, invertibility and query preservation do not necessarily coincide for XML mappings and query languages. Recall the class $\mathcal{X}$ of *XPath queries* defined in Section 2, which does not support query composition, identity mapping (from XML documents to XML documents), or the ability to navigate a recursive DTD based on certain patterns that are expressible in terms of the Kleene closure p^*.

THEOREM 3.1. *There exists an invertible* XML *mapping that is not query preserving w.r.t. $\mathcal{X}$; and there exists an* XML *mapping that is not invertible but is query preserving w.r.t. the class of $\mathcal{X}$ queries without position() qualifier.*

PROOF. The proof consists of two parts.

(1) We first show that invertibility does not entail query preservation w.r.t. *the XPath fragment $\mathcal{X}$*. Recall the source DTD S_1 and the target DTD S_2 shown in Figure 2:

$S_1 = (\{r, A, B, C\}, P_1, r)$, where P_1 consists of the following productions:
$r \to A, \quad A \to B, C, \quad B \to A + \epsilon, \quad C \to \epsilon,$

$S_2 = (\{r, A\}, P_2, r)$, where P_2 consists of:
$r \to A, \quad A \to A + \epsilon.$

The mapping $\sigma_d : \mathcal{I}(S_1) \to \mathcal{I}(S_2)$ given in Example 2.1 can be expressed by the function path() (see Section 4.1) from the edges of S_1 to paths of S_2 as follows:

$$\mathsf{path}(r, A) = A \qquad \mathsf{path}(A, B) = A \qquad \mathsf{path}(A, C) = A/A \qquad \mathsf{path}(B, A) = A/A$$

For instance, given a B node v_1 and its A child v_2 in T, $\mathsf{path}(B, A)$ maps the edge (v_1, v_2) in T to a path (v_1', v', v_2') in $\sigma_d(T)$, where v_1', v', v_2' are all labeled with A, and v_1', v_2' are mapped from v_1, v_2, respectively. That is, if v_1 in T is mapped to v_1' in $\sigma_d(T)$, then the child v_2 of v_1 in T is mapped to the node v_2' in $\sigma_d(T)$ that is reachable from v_1' by following $\mathsf{path}(B, A) = A/A$. In other words, $\mathsf{path}(B, A)$ is relative to node v_1'.

Obviously σ_d is invertible: one can restore the original T from $\sigma_d(T)$ by generating T inductively top-down starting from the root r_1 of T.

Now consider an $\mathcal{X}$ query $Q = //B$. An equivalent translation of Q over $\sigma_d(T)$ is to find all the elements in the A-chain of $\sigma_d(T)$ that are reachable from r_2 via A^{3k+2}. It is easy to prove by contradiction that A^{3k+2} is not expressible in $\mathcal{X}$, even with the *position()* qualifier. Thus σ_d is not query preserving w.r.t. $\mathcal{X}$.

(2) We next show that query preservation w.r.t. *the XPath fragment $\mathcal{X}$* without *position()* qualifiers does not entail invertibility. Consider a source DTD S_1:

$S_1 = (\{r, A\},\ P_1,\ r)$, where P_1 consists of:
$r \to A^*, \quad A \to \mathsf{str}.$

and assume that the target DTD S_2 is identical to S_1.

The mapping $\sigma_d : \mathcal{I}(S_1) \to \mathcal{I}(S_2)$ is such defined that for any $T \in \mathcal{I}(S_1)$, the root r_1 of T is mapped to the root r_2 of $\sigma_d(T)$, the A children of r_1

are mapped to the A children of r_2 such that there is a bijection from the A children of r_1 to the A children of r_2; however, the A-children of r_2 are ordered based on their string values (str).

All the $\mathcal{X}$ queries posed over $T \in \mathcal{I}(S_1)$ are equivalent to one of the following forms of $\mathcal{X}$ queries: ϵ, A, $A[q]$, where q is a Boolean formula defined in terms of atomic formulas of the form *text()* = *'c'*. Since the identity mapping from $\mathcal{X}$ to $\mathcal{X}$ yields equivalent queries over $\sigma_d(T)$ for these queries, the mapping σ_d is query preserving w.r.t. $\mathcal{X}$. However, σ_d is not invertible: one cannot recover the original order of the A elements of r_1 based on σ_d. $\square$

Invertibility and Query Preservation: Equivalence. We next identify sufficient conditions for the two to coincide: the definability of *the identity mapping* from XML documents to XML documents, and *query composability* (i.e., for any Q_1, Q_2 in $\mathcal{L}$, $Q_2 \circ Q_1$ is also in $\mathcal{L}$). Recall that the identity mapping is definable in neither regular XPath nor XPath, because (regular) XPath does not return XML trees.

THEOREM 3.2. *Let $\mathcal{L}$ be any* XML *query language and σ_d be a mapping: $\mathcal{I}(S_1) \to \mathcal{I}(S_2)$.*

—*If the identity mapping id is definable in $\mathcal{L}$ and σ_d is query preserving w.r.t. $\mathcal{L}$, then σ_d is invertible.*

—*If $\mathcal{L}$ is composable, σ_d is invertible and σ_d^{-1} is expressible in $\mathcal{L}$, then σ_d is query preserving w.r.t. $\mathcal{L}$.*

PROOF. We prove the two statements as follows.

(1) Suppose that σ_d is query preserving w.r.t. $\mathcal{L}$. Then there exists a computable function $\mathsf{Tr} : \mathcal{L} \to \mathcal{L}$ such that for any $Q \in \mathcal{L}$ and any $T \in \mathcal{I}(S_1)$, $Q(T) = \mathsf{Tr}(Q)(\sigma_d(T))$. Since id is in $\mathcal{L}$, we have $T = id(T) = \mathsf{Tr}(id)(\sigma_d(T))$ for any $T \in \mathcal{I}(S_1)$. That is, $\sigma_d^{-1} = \mathsf{Tr}(id)$, and thus σ_d is invertible.

(2) Suppose that $\mathcal{L}$ is composable, σ_d is invertible and σ_d^{-1} is in $\mathcal{L}$. Then define $\mathsf{Tr} : \mathcal{L} \to \mathcal{L}$ to be $\mathsf{Tr}(Q) = Q \circ \sigma_d^{-1}$ for any $Q \in \mathcal{L}$. Obviously Tr is computable, and $\mathsf{Tr}(Q)$ is in $\mathcal{L}$ since $\mathcal{L}$ is composable. Furthermore, for any $Q \in \mathcal{L}$ and $T \in \mathcal{I}(S_1)$, $Q(T) = Q(\sigma_d^{-1}(\sigma_d(T)) = \mathsf{Tr}(Q)(\sigma_d(T))$. Thus Tr is an effective query translation function for $\mathcal{L}$. $\square$

Recall the class $\mathcal{X}_R$ of *regular* XPath queries defined in Section 2. Although the identity mapping *id* is not definable in $\mathcal{X}_R$, we show below that query preservation w.r.t. $\mathcal{X}_R$ is a stronger property than invertibility: every node in a source document can be uniquely identified by an $\mathcal{X}_R$ query on the target document, and thus can be recovered.

THEOREM 3.3. *If an* XML *mapping σ_d is query preserving w.r.t. $\mathcal{X}_R$, then σ_d is invertible. Conversely, there exists σ_d that is invertible but is not query preserving w.r.t. $\mathcal{X}_R$.*

PROOF. Suppose that $\sigma_d : \mathcal{I}(S_1) \to \mathcal{I}(S_2)$ is query preserving w.r.t. $\mathcal{X}_R$. We show that σ_d is invertible by providing an algorithm for computing σ_d^{-1}. Given $\sigma_d(T)$, the algorithm recovers T as follows. It first creates the root r_1 of T, with id o given in $\mathsf{idM}(o, o')$, where o' is the id of the root r_2 of $\sigma_d(T)$. It then recursively expands T top-down as follows, until T cannot be expanded further.

To expand T, for each node v created for T, it recovers the children of v based on its type A, the production $A \to \alpha$ of A in S_1, and the query translation function $\mathsf{Tr} : \mathcal{X}_R \to \mathcal{X}_R$. To do so it makes use of a subclass of $\mathcal{X}_R$, referred to as $\mathcal{X}_R$ *paths*, which are of the form $\rho = \eta_1/\ldots/\eta_k$, where $k \geq 1$, η_i is of the form $A[q]$, and q is either true or a *position()* qualifier. By induction on the length of the path from r_1 to v, one can easily verify that there is a unique $\mathcal{X}_R$ path ρ such that $r_1[\![\rho]\!]$ is a singleton set $\{v\}$. The process of recovering the children of v is based on the structure of α:
(1) $\alpha = A_1, \ldots, A_n$. For each A_i, define an $\mathcal{X}_R$ query $Q_i = \mathsf{Tr}(\rho/A_i\ [position() = k])$, where k indicates the k-th occurrence of A_i element in α if it has multiple A_i elements. Note that evaluating $\rho/A_i[position() = k]$ at the root r_1 of T is equivalent to evaluating the query $A_i[position() = k]$ at the context node v in T, which would return the k-th A_i child of v. Since σ_d is query preserving, evaluating $Q_i(\sigma_d(T))$ at the root r_2 of $\sigma_d(T)$ yields the same answer as evaluating $\rho/A_i[position() = k]$ at the root r_1 in T. Let v_i be the single node returned by $Q_i(\sigma_d(T))$ at the root r_2. Copy $v_1, \ldots, v_n$ to T as the children of v, and for each $i \in [1, n]$, proceed to expand the subtree at v_i in the same way.
(2) $\alpha = A_1 + \cdots + A_n$. For each A_i, let $Q_i = \mathsf{Tr}(\rho/A_i)$. Evaluate $Q_i(\sigma_d(T))$ at the root r_2 of $\sigma_d(T)$ as in (1). Since σ_d is query preserving and T is an XML tree that conform to S_1, there exists one and only one $i \in [1, n]$ such that $Q_i(\sigma_d(T))$ returns a single note v_i (and the Q_j's return empty for $j \neq i$). Copy v_i to T as the only child of v and proceed to expand the subtree at v_i in the same way.
(3) $\alpha = B^*$. For each natural number k, evaluate $Q_k(\sigma_d(T))$ at the root r_2 of $\sigma_d(T)$ as in (1), where $Q_k = \mathsf{Tr}(\rho/B[position() = k])$, until it reaches a k_0 such that $Q_{k_0}(\sigma_d(T)) = \emptyset$. Since σ_d is query preserving and $\sigma_d(T)$ is mapped from an XML tree T that conforms to S_1, $Q_k(\sigma_d(T))$ at r_2 of $\sigma_d(T)$ yields the same answer as evaluating the query $B[position() = k]$ at the context node v in T, that is, there exists one and only one node v_k returned by $Q_k(\sigma_d(T))$ at r_2 for each $k < k_0$, and for any $k \geq k_0$, $Q_k(\sigma_d(T)) = \emptyset$. Copy v_k to T as the k-th child of v for all $k < k_0$, and proceed to expand the subtree at each v_k in the same way.
(4) $\alpha = \mathsf{str}$. Find the string value by evaluating $\mathsf{Tr}(\rho/text())$ at the root r_2 of $\sigma_d(T)$ as in (1).
(5) $\alpha = \epsilon$. Nothing needs be done here.

The children of v generated above have different identities from v because no distinct nodes in T have the same identity and σ_d is query preserving. This process terminates. Indeed, each step of the process expands T by copying distinct nodes from $\sigma_d(T)$, and $\sigma_d(T)$ is a (finite) XML tree. One can verify that $T = \sigma_d^{-1}(\sigma_d(T))$, i.e., the algorithm above indeed computes σ_d^{-1}. Thus σ_d^{-1} is computable and σ_d is invertible.

To show that invertibility does not necessarily lead to query preservation w.r.t. $\mathcal{X}_R$, recall the DTDS S_1 and S_2 defined in the proof of Theorem 3.1 (1). Consider a mapping $\sigma_d : \mathcal{I}(S_1) \to \mathcal{I}(S_2)$ such that for any $T \in \mathcal{I}(S_1)$, the root r_1 of T is mapped to the root r_2 of $\sigma_d(T)$, the A child of r_1 is mapped to the A child of r_2; and inductively, if an A element v in T is mapped to an A element v' in $\sigma_d(T)$, then the B, C children of v are also mapped to v', and the A child of the B node is mapped to the A child of v', such that the number of A nodes in T is the same as that in $\sigma_d(T)$. Obviously, σ_d is invertible: given any $\sigma_d(T)$ one can recover T

such that the number of A nodes in T is the same as that in $\sigma_d(T)$, and each A node in T has a B child followed by a C child. However, one cannot translate an $\mathcal{X}_R$ query $(A/(B \cup C))^*$ over S_1 to an equivalent $\mathcal{X}_R$ query over S_2. $\square$

Complexity. It is common to find XML mappings defined in XQuery or XSLT. A natural and important question is to decide whether or not an XML mapping in XQuery or XSLT is invertible or query preserving w.r.t. a query language $\mathcal{L}$. Unfortunately, this is impossible for XML mappings defined in any language that subsumes first-order logic (*FO*), such as XQuery, XSLT, even when $\mathcal{L}$ consists of projection queries only. Thus it is beyond reach in practice to answer the question for XQuery or XSLT mappings. That is, it is impossible to find a systematic method to determine an XML mapping defined in one of these languages is information preserving or not.

THEOREM 3.4. *It is undecidable to determine, given an* XML *mapping* σ_d *defined in any language subsuming FO, whether or not (a)* σ_d *is invertible; and (b)* σ_d *is query preserving w.r.t. projection queries.*

These negative results are not surprising: as indicated by the proof below, the undecidability results already hold for relational data transformations expressed in relational algebra (*FO*), and for relational projection queries. Similar undecidability results have been established for relational and object-oriented models [Hull 1986; Miller et al. 1994], and recently for mappings from XML to relations [Barbosa et al. 2005], although query preservation was not investigated there.

PROOF. It suffices to show that these problem are undecidable for relational mappings defined in relational algebra (RA). For if it holds, the undecidability carries over to XML mappings defined in *FO* since relational data can be coded in XML and RA queries can be expressed in *FO* over XML trees.

We verify the undecidability by reduction from the equivalence problem for RA queries. That is the problem to decide, given two RA queries $Q_1, Q_2 : \mathcal{R}_1 \to \mathcal{R}_2$, whether or not $Q_1 \equiv Q_2$, that is, whether or not for any relational database I of $\mathcal{R}_1$, $Q_1(I) = Q_2(I)$. This equivalence problem is undecidable [Abiteboul et al. 1995].

(a) We first show that the invertibility problem is undecidable. Given two RA queries $Q_1, Q_2 : \mathcal{R}_1 \to \mathcal{R}_2$, we define a RA mapping $V : \mathcal{R}_1 \times \mathcal{R}_2 \to \mathcal{R}_1 \times \mathcal{R}_2$, as follows:

$$V = \pi_{R_1} \times (\pi_{R_2} \cup \Delta(Q_1, Q_2)), \qquad \Delta(Q_1, Q_2) = (Q_1 \setminus Q_2) \cup (Q_2 \setminus Q_1).$$

Note that $Q_1 \equiv Q_2$ iff $\Delta(Q_1, Q_2) = \emptyset$, that is, when $\Delta(Q_1, Q_2)$ always returns an empty set.

We show that V is invertible iff $Q_1 \equiv Q_2$. If $Q_1 \equiv Q_2$, then $\Delta(Q_1, Q_2) = \emptyset$. Then V is the identity query and is certainly invertible. Conversely, if $Q_1 \not\equiv Q_2$, then there exists an instance I of $\mathcal{R}_1$ such that $\Delta(Q_1, Q_2)(I)$ is nonempty. Consider two distinct instances of $\mathcal{R}_1 \times \mathcal{R}_2$: $I_1 = (I, \Delta(Q_1, Q_2)(I))$ and $I_2 = (I, \emptyset)$. Since $V(I_1) = V(I_2) = (I, \Delta(Q_1, Q_2)(I))$, V is not injective and thus is not invertible (there exists no inverse function for V).

(b) We now show that query preservation for projection queries is undecidable. Given two RA queries $Q_1, Q_2 : \mathcal{R}_1 \to \mathcal{R}_2$, we use the same RA mapping V given above to show that V is query preserving w.r.t. a fixed query iff $Q_1 \equiv Q_2$. Consider a fixed query $Q = \pi_{R_2}$. First, suppose that $Q_1 \equiv Q_2$. Then one can define Tr such that $\mathsf{Tr}(Q) = Q$. This shows that V is query preserving w.r.t. Q. Conversely, suppose that $Q_1 \not\equiv Q_2$. Suppose, by contradiction, that there is a computable query translation function Tr such that $Q' = \mathsf{Tr}(Q)$. Recall I_1, I_2 given above. Obviously, $Q(I_1) \neq Q(I_2)$, while $Q'(V(I_1)) = Q'(V(I_2))$ since $V(I_1) = V(I_2)$. Thus either $Q(I_1) \neq Q'(V(I_1))$ or $Q(I_2) \neq Q'(V(I_2))$; that is, Tr does not translate Q to an equivalent query over the target, which contradicts the assumption above. Thus V is not query preserving w.r.t. Q. □

The undecidability suggests that we start with languages simpler than XQuery and XSLT when studying information preserving XML mappings. Indeed, understanding (regular) XPath query preservation is a necessary step toward a full treatment of XML mappings defined in XQuery or XSLT, in which XPath is embedded. The extension to regular XPath is particularly natural for studying XSLT, where the recursive semantics of rule processing makes it straightforward to express regular XPath queries in small stylesheets. Thus in the remainder of the paper we shall focus on (regular) XPath, both for defining XML mappings and for querying XML data.

4. SCHEMA EMBEDDINGS FOR XML

The negative results in Section 3 tell us that it is already hard to determine whether or not an XML mapping is information preserving, not to mention finding one. As remarked in Section 1, it is also nontrivial to check the type safety of XML mappings. This motivates us to look for a class of XML mappings that are *guaranteed* to be both information preserving and type safe.

We approach this problem by specifying XML mappings in terms of schema level embeddings, and providing an automated derivation of instance-level mappings from these embeddings. Our notion of *schema embeddings* is novel: it extends the conventional notion of graph similarity by allowing an edge in a source DTD to be mapped to a path in a target DTD. Intuitively, this allows a "smaller" DTD to be embedded in a "larger" one.

In this section we define XML schema embeddings, present an algorithm for deriving an instance-level mapping from a schema embedding, provide XSLT coding of these instance-level mappings as well as their inverse, and verify that the resulting mappings indeed ensure both type safety and information preservation.

4.1 Schema Level Embeddings

Consider a source XML DTD schema $S_1 = (E_1, P_1, r_1)$ and a target DTD $S_2 = (E_2, P_2, r_2)$. In a nutshell, a schema embedding σ is a pair of functions (λ, path) that maps each A type in E_1 to a $\lambda(A)$ type in E_2, and each edge (A, B) in S_1 to a unique path, $\mathsf{path}(A, B)$ in S_2, from $\lambda(A)$ to $\lambda(B)$. Intuitively, if an A node u in the source is mapped to a $\lambda(A)$ node v in the target, then we map the B child of u to a $\lambda(B)$ descendant of v identified by the unique $\mathsf{path}(A, B)$ emanating from v. The mapping must obey three correctness properties: (1) $\lambda(A)$ must be semantically compatible with A, *i.e.,* $\lambda(A)$ and A belong to the same "domain", (2) $\mathsf{path}(A, B)$ must have a "larger information capacity" than the edge (A, B)—for example, a STAR edge can only be mapped to a path with at least one STAR edge, and (3) the S_2 paths mapped from sibling edges in S_1 must be sufficiently distinct to allow information to be preserved. To formally define these correctness conditions, we first introduce some notation.

$\mathcal{X}_R$ Paths. An $\mathcal{X}_R$ *path* over a DTD $S = (E, P, r)$ is an $\mathcal{X}_R$ query of the form $\rho = \eta_1/\ldots/\eta_k$, where $k \geq 1$, η_i is of the form $A[q]$, and q is either true or a *position()* qualifier, such that ρ represents a label path in S, carrying all the position labels on the path. An $\mathcal{X}_R$ path is called an AND *path* (resp. OR *path*, and STAR *path*) if it is nonempty and consists of only solid or star edges (resp. of solid edges and at least one dashed edge, and of solid edges and at least one edge labeled $*$). Referring to Figure 1(c), for example, *basic/class/semester* is an AND path and a STAR path, and *mandatory/regular* is an OR path.

An $\mathcal{X}_R$ path ρ_1 is called a *prefix* of another $\mathcal{X}_R$ path ρ_2 if $\rho_2 = \rho_1/\eta_j/\ldots/\eta_k$.

Schema Element Similarity. A *similarity matrix* for S_1 and S_2 is an $|E_1|\times|E_2|$ matrix att of numbers in the range [0, 1]. For any $A \in E_1$ and $B \in E_2$, $\mathsf{att}(A, B)$ indicates the suitability of mapping A to B as determined by human domain experts or computed by an existing *schema matching* algorithm, e.g., [Athitsos et al. 2005; Doan et al. 2001; Li and Clifton 2000]. We note that most previous work on mapping construction has assumed an accurate set of attribute correspondences; that is, $\mathsf{att}(A, B) \in \{0, 1\}$. Supporting non-Boolean quality measures accrues several advantages with minimal increased complexity, since a candidate embedding can be computed based on a machine-generated similarity measure. First, the algorithm can be used in "best-effort" applications, and second, attribute matches that participate in information-preserving mappings can be preferred over others, even if resulting matching will be hand-checked by a domain expert. Leveraging this, in the next section we shall formalize the problem of finding a schema embedding as an optimization problem as commonly encountered in "best-effort" applications.

Type Mapping. A *type mapping* λ from S_1 to S_2 is a (total) function from E_1 to E_2; in particular, it maps the root of S_1 to the root of S_2, i.e., $\lambda(r_1) = r_2$. A type mapping λ is *valid* w.r.t. a similarity matrix att if for any $A \in E_1$, $\mathsf{att}(A, \lambda(A)) > 0$. Note that in general one could define validity in terms of a threshold θ, *i.e.,* λ is valid if $\mathsf{att}(A, \lambda(A)) > \theta$; we assume $\theta = 0$ in this paper to simplify the discussion.

Path Mapping. A *path mapping* from S_1 to S_2, denoted by $\sigma : S_1 \to S_2$, is a pair (λ, path), where λ is a type mapping and path is a function that maps each edge (A, B) in S_1 to an $\mathcal{X}_R$ path, $\mathsf{path}(A, B)$, that is from $\lambda(A)$ to $\lambda(B)$ in S_2.

For a particular element type A in E_1, we say that σ is *valid* for A if the following conditions hold, referred to as the *path type* condition and the *prefix-free* condition on $\mathsf{path}(A, B)$, based on the production $A \to P_1(A)$ in S_1:

—if $P_1(A) = B_1, \ldots, B_l$, then for each i, $\mathsf{path}(A, B_i)$ is an AND path from $\lambda(A)$ to $\lambda(B_i)$ that is *not a prefix* of $\mathsf{path}(A, B_j)$ for any $j \neq i$;

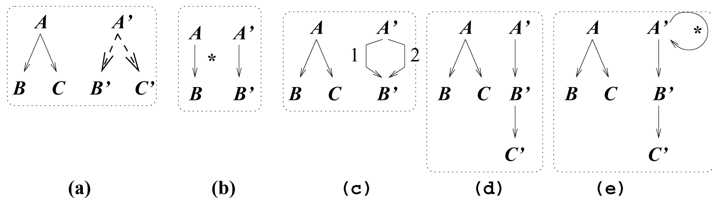


Fig. 3. Path mappings for DTDs.

—if $P_1(A) = B_1 + \cdots + B_l$, then for each i, $\mathsf{path}(A, B_i)$ is an OR path from $\lambda(A)$ to $\lambda(B_i)$ that is not a prefix of $\mathsf{path}(A, B_j)$ for any $j \neq i$ [1];

—if $P_1(A) = B^*$, then $\mathsf{path}(A, B_i)$ is a STAR path;

—if $P_1(A) = \mathsf{str}$, then $\mathsf{path}(A, \mathsf{str})$ is an AND path ending with *text()*.

The following example illustrates why these conditions are necessary to impose for deriving an instance-level mapping from σ.

Example 4.1. In Figure 3 a number of simple schema mapping examples are presented to illustrate the validity conditions for schema embeddings. The figure shows five scenarios, labeled "a" through "e." Each scenario consists of a source and a target DTD, with the schema graph of the source DTD on the left and the target on the right. Except for Figure 3(c), types in the source are mapped (by the type mapping λ) to types with the same name in the target, for example, A is mapped to A'. In Figure 3(c), two source types are mapped to one target type, in that both $\lambda(B) = B'$ and $\lambda(C) = B'$.

For Figure 3(a), there is no valid path mapping from the source DTD to the target, since $\mathsf{path}(A, B)$ and $\mathsf{path}(A, C)$ violate the path type condition; intuitively, B and C must coexist in a source document while only one of B' and C' exists in the target. Similarly, for Figure 3(b), the source cannot be mapped to the target since the target cannot accommodate possibly multiple B elements in the source. For Figure 3(c), a valid embedding is $\mathsf{path}(A, B) = B'[position() = 1]$ and $\mathsf{path}(A, C) = B'[position() = 2]$. For Figure 3(d), there is no valid embedding since $\mathsf{path}(A, B)$ is a prefix of $\mathsf{path}(A, C)$, violating the prefix-free condition. For Figure 3(e), a valid embedding is $\mathsf{path}(A, B) = A'/B'$ (by unfolding the cycle once) and $\mathsf{path}(A, C) = B'/C'$.

Finally, we define XML schema embeddings as follows.

Schema Embedding. A *schema embedding* from S_1 to S_2 w.r.t. a similarity matrix att is a path mapping $\sigma = (\lambda, \mathsf{path})$ from S_1 to S_2 such that λ is valid w.r.t. att, and σ is valid for every element A in E_1.

Note that schema embedding takes into account the semantics of the source and target schemas by means of the similarity matrix, on top of the syntactic and structural correspondences between the two schemas.

[1] Abusing our normal form of DTDs, an optional type B can be specified as, for example, $A \to B + \epsilon$; here $\mathsf{path}(A, B)$ simply needs to be an OR path since ϵ is not an element type and thus $\mathsf{path}(A, \epsilon)$ is undefined.

Example 4.2. Assume a similarity matrix att such that $\text{att}(X, X') = 1$ for all X in the DTD S_0 of Figure 1(a) and X' in S of Figure 1(c). Here the similarity matrix att imposes no restrictions: any name in the source can be mapped to any name in the target; thus the embedding here is decided solely on the DTD structures. The source DTD S_0 can be embedded in the target S via $\sigma_1 = (\lambda_1, \text{path}_1)$ defined as follows:

λ_1(db) = school, λ_1(class) = course, λ_1(type) = category,
$\lambda_1(A)$ = A /* A: cno, title, regular, project, prereq, str */

path_1(db, class) = courses/current/course
path_1(class, cno) = basic/cno
path_1(class, title) = basic/class/semester[position()=1]/title
path_1(class, type) = category
path_1(type, regular) = mandatory/regular
path_1(type, project) = advanced/project
path_1(regular, prereq) = required/prereq
path_1(prereq, class) = course
$\text{path}_1(A$, str) = *text()* /* A for cno, title */

Note that $\text{path}_1(A, B)$ is a path in S denoting how to reach $\lambda_1(B)$ from $\lambda_1(A)$; that is, the path is *relative to* $\lambda_1(A)$ rather than starting from the root. For example, path_1*(type, project)* indicates how to reach *project* from a *category* context node in S, where *category* is mapped from *type* in S_0 by λ_1.

In contrast, one *cannot* map S_0 to S by graph similarity, which requires that node A in the source is mapped (similar) to B in the target only if all *children* of A are mapped (similar) to *children* of B. In other words, graph similarity maps an edge in the source to an edge in the target.

The definition of schema embedding can be extended to support further restructuring "across hierarchies" such that a child B of a source type A is not necessarily mapped to a descendant of $\lambda(A)$ in the target; this can be achieved via, for example, upward modality in $\text{path}(A, B)$. It is also possible that an AND edge does not have to be mapped to an AND path. We focus on the main idea of schema embeddings in this paper and defer the extension to a later study.

Embedding Quality. There are many possible metrics. In this paper we consider only a simple one: the quality of a schema embedding $\sigma = (\lambda, \text{path})$ w.r.t. att is the sum of $\text{att}(A, \lambda(A))$ for $A \in E_1$, and we say that σ is *invalid* if λ is invalid w.r.t. att. We refer to this metric as $\text{qual}(\sigma, \text{att})$.

4.2 Instance Level Mapping

For a valid schema embedding $\sigma = (\lambda, \text{path})$ from S_1 to S_2, we give its semantics by defining a (data) instance-level mapping $\sigma_d : \mathcal{I}(S_1) \to \mathcal{I}(S_2)$, referred to as the XML *mapping of* σ. We define σ_d by presenting an algorithm that, given an instance T_1 of S_1, computes an instance $T_2 = \sigma_d(T_1)$ of S_2.

Before presenting the full algorithm, we introduce two notions, the minimum default instance of a target schema tag and the mapping fragment of a source node. In a nutshell, the mapping fragment of a source node v is the subtree in the target T_2 that is "mapped" from the subtree of v in T_1. The minimum default instance of a target schema tag A is a fixed default instance of A, which will be added to T_2 by the instance-level mapping σ_d such that T_2 is guaranteed to conform to S_2.

Minimum Default Instances. For a DTD $S_1 = (E_1, P_1, r_1)$ and a particular element type $A \in E_1$, we *fix* a default instance of A. Recall from Section 2 that we consider consistent DTDs only; as a result, instances of A exist. Among these instances of A one can *fix* an arbitrary one and treat it as the default instance. In other words, the default instance is a constant property of a single schema. Below we present how we fix a default instance.

Assume a fixed order on the types (XML tags) of E_1, and a fixed string value #s. We compute the *minimum default instance* of A, denoted by $\mathsf{mindef}(A)$, inductively based on the definition of A as follows. We associate a variable $\mathsf{rank(A)}$ with A, having an initial value 1. (1) If $P_1(A)$ is str, then $\mathsf{mindef}(A)$ is a node with label A carrying a str child with #s as its value, and we set $\mathsf{rank(A)} = 0$. (2) If $P_1(A)$ is B^*, then $\mathsf{mindef}(A)$ is a single A-node without any children, and we set $\mathsf{rank(A)} = 0$. We then repeat the following process until for all $A \in E_1$, $\mathsf{rank(A)} = 0$. For each $A \in E_1$ with $\mathsf{rank(A)} = 1$, (3) if $P_1(A)$ is $B_1, \ldots, B_n$ and $\mathsf{rank(B_i)} = 0$ for all $i \in [1, n]$, we let $\mathsf{mindef}(A)$ be a node with label A and children consisting of $\mathsf{mindef}(B_1), \ldots, \mathsf{mindef}(B_n)$; or (4) if $P_1(A)$ is $B_1 + \cdots + B_n$ and one of B_i has $\mathsf{rank(B_i)} = 0$, we let $\mathsf{mindef}(A)$ be $\mathsf{mindef}(B_j)$ such that B_j is the smallest among all B_i's with $\mathsf{rank(B_i)} = 0$ *w.r.t.* the order on the types in E_1. In both cases we set $\mathsf{rank(A)} = 0$. Since S_1 is consistent, in each iteration there must exist some A satisfying (3) or (4) above. Upon the termination of the process we have $\mathsf{mindef}(A)$ defined for all $A \in E_1$.

Example 4.3. Given the XML mapping σ_d of the embedding defined in Example 4.2, some example values of mindef might be:

```
                      <student>
                          <ssn> #s </ssn>
mindef(student) =         <name> #s </name>
                          <gpa> #s </gpa>
                          <taking> </taking>
                      </student>

mindef(prereq) =      <prereq> </prereq>

                       <category>
                           <advanced>
mindef(category) =             <project> #s </project>
                           </advanced>
                       </category>
```

Production Fragments. For a source node v with type A in S_1, we define $t = \mathsf{pfrag}_A(v)$ to be an XML fragment, referred to as the *production fragment* of v with respect to σ_d. The root r_t of t is a node with label $\lambda(A)$ and carrying the same *node identity* as v. In addition, a set of "hot" leaf nodes $\mathsf{hleaf}(t)$ is defined,

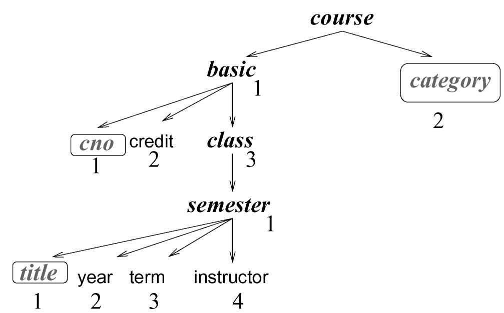


Fig. 4. The production fragment of *class* in Example 4.2.

and with each such leaf, h, is associated a node from T_1, $\mathsf{src}(h)$. Finally, during the production of t, *position* information is associated with each node $v \in t$ by $\mathsf{pos}(v)$ (with the first child of a node numbered "1"). For example, nodes of type *class* in Example 4.2 will have production fragments, as shown in Figure 4. Here $\mathsf{hleaf}(t) = \{category, cno, title\}$, shown in rectangles in the figure. Other nodes correspond to the minimum definition of *str* nodes. Finally, $\mathsf{pos}(v)$ for each node is shown as a number under the node.

Constructing Production Fragments. We now give the details of computing the XML fragment $t = \mathsf{pfrag}_A(v)$, for a source node v of type A. First, the root r_t is created and set as the root of t, and $\mathsf{pos}(r_t)$ is set to 1. Second, for each child v' of v in T_1, we add $\rho = \mathsf{path}(A, B)$ to t, where v' is of type B. In general, this is accomplished by dividing ρ into a prefix ρ_1 and a suffix ρ_2, such that ρ_1 is the longest prefix of ρ that matches a path ρ' in t. Nodes $u_1, \ldots, u_k$ are created for each of the k steps in ρ_2 ($|\rho_2| = k$), and $\mathsf{pos}(u_i)$ is set to one of a) j where the predicate [$\mathsf{position}() = j$] appeared on the corresponding step of ρ_2 or b) j where the parent of u_i is of type C in S_2, u_i is of type C'_j, and $P_2(C)$ is of the form $C'_1, \ldots, C'_j, \ldots, C'_m$, and c) if neither (a) nor (b) holds, j where v' is the j-th child of v. Node u_1 is then added as a child of u_0, which is the lowest node t matching ρ'. Finally, u_k is added to $\mathsf{hleaf}(t)$, and $\mathsf{src}(u_k)$ is set to v'. (Note that u_0 cannot be in $\mathsf{hleaf}(t)$ due to the prefix-free property of valid schema embeddings.)

Once this process completes, the embedded nodes needed are present, but ordering may be incorrect and some required nodes may be missing. To deal with this, if any node u in t but not in $\mathsf{hleaf}(t)$ requires a child u' of type C at position i, but no such u' exists with $\mathsf{pos}(u') = i$, then a copy of $\mathsf{mindef}(C)$ is added as a child of u, and $\mathsf{pos}(m)$ where m is the root of the copy of $\mathsf{mindef}(C)$ is set to i. Note that such a child can be required for a parent u that is associated with AND node in the schema, or a parent that is associated with a STAR node if some child u'' with $\mathsf{pos}(u'') > i$. Finally, the children of nodes in t not copied from some mindef are sorted into the pos order.

Instance Construction Algorithm. A construction algorithm, $\mathsf{InstMap}$, for σ_d is shown in Figure 5. In a nutshell, $\mathsf{InstMap}$ constructs T_2 in a top down fashion, by repeatedly replacing a hot node with the appropriate production fragment. In the process, the node id mapping $\mathsf{idM}()$ is generated at line 6 to map the ids of the nodes in T_2 to the ids of the corresponding nodes in T_1. Note that a node

function InstMap (T_1)

Input: XML tree T_1 conforming to DTD S_1
Output: $\sigma_d(T_1)$

1. initialize T_2 to be a single root node r_2;
2. $H := \{r_2\}$; *// set of hot nodes*
3. **while** $H \neq \emptyset$ **do**
4. select some h in T_2 from H where $\mathsf{src}(h)$ is a node of type B;
5. replace h in T_2 with $\mathsf{pfrag}_B(\mathsf{src}(h))$;
6. $\mathsf{idM}(r) = \mathsf{src}(h)$ where r is the root of $\mathsf{pfrag}_B(\mathsf{src}(h))$;
7. $H := (H \cup \mathsf{hleaf}(t)) - \{h\}$;
8. **return** T_2;

Fig. 5. Algorithm InstMap.

in T_1 is in H exactly once, and thus the algorithm trivially terminates in time linearly proportional to the size of the larger of T_1 and T_2.

Example 4.4. Consider the XML mapping σ_d of the embedding defined in Example 4.2. Given an instance T_1 of S_0 of Figure 1(a) and σ_d, algorithm InstMap generates a tree T_2 of S of Figure 1(c) as follows: InstMap first creates the root *school* of T_2, as a copy of the root *db* of T_1, and marks *school* as a "hot" node by adding it to H at line 2. The first time through the loop at lines 4–7, the new root is chosen as h, and replaced with the production fragment of the *db* node from T_1. This fragment is rooted at a *school node*, and has a *history* child and a *current child*. Since *history* is not involved in any $\mathsf{path}_i(db,B)$ for any potential child of db, the minimum default instance of the *history* node is used in this production fragment. Since *history* has an outgoing STAR edge, this minimum default instance is in fact a single *history* node. For the *current* node, the construction of the production fragment is more complicated. In fact, it produces a child of current with label *course* for each *class* child of the original root of T_1. Since these nodes terminate a path in the mapping, they are in the hleaf set for the production fragment just added to the tree, and thus they are added to H at line 7 (and *db* is removed). The algorithm continues by selecting one of the newly-created *course* nodes as h and replacing it with the production fragment of the *class* node in T_1. Note that at line 6 the node id mapping idM() is generated accordingly to map the ids of the nodes in T_2 to the ids of the corresponding nodes in T_1, in this case mapping the newly created *course* node back to the source *class node*. The new production fragment for *class* follows the form shown in Figure 4, and includes a *basic* subtree with *cno* as a "hot node," *credit* as a minimum default instance and a subtree under *class* of a single *semester*. Under this node, *title* is again added to H as a "hot node," while *year*, *term* and *professor* are filled in with their respective minimum default instances. The algorithm continues until the entire tree is created.

Correctness. We next show that σ_d is *well defined*. That is, given any T_1 in $\mathcal{I}(S_1)$, $\sigma_d(T_1)$ is an XML tree and moreover, it is type safe; that is, it conforms to S_2. This is nontrivial due to the interaction between different paths defined for disjunction types in the schema mapping σ, among other things. Consider, for example, path*(type, regular)* in Example 4.2. The path requires the existence of a *regular* child under a *mandatory* element m, which is in turn a child under a *category* element c in an instance of S. Thus it rules out the possibility of adding an *advanced* child under c or a *lab* child under m, perhaps requested by a *conflicting path* in σ. However, Theorem 4.1 shows that the *prefix-free* condition in the definition of valid path functions ensures that conflicting paths do not exist. Theorem 4.1 also shows that σ_d is *injective*: it maps distinct nodes in T_1 to distinct nodes in $\sigma_d(T_1)$, a property necessary for information preservation.

THEOREM 4.1. *The* XML *mapping* σ_d *of a valid schema embedding* $\sigma : S_1 \to S_2$ *is well defined and injective.*

PROOF. The proof consists of three parts. We first show that δ maps distinct $\mathcal{X}_R$ paths in S_1 from r_1 to distinct $\mathcal{X}_R$ paths in S_2 from r_2. Then, using this we show that σ_d is injective. Based on this we finally show that σ_d is well defined.

(1) We first define a function δ that maps $\mathcal{X}_R$ paths from the root r_1 in S_1 to $\mathcal{X}_R$ paths from the root r_2 in S_2. Given an $\mathcal{X}_R$ path $\rho = A_1[q_1]/\ldots/A_k[q_k]$ in S_1 from r_1, $\delta(\rho)$ is defined to be $\mathsf{path}(r_1, A_1)[q_1]/\ldots/\mathsf{path}(A_{k-1}, A_k)[q_k]$, an $\mathcal{X}_R$ path in S_2 from r_2, by substituting $\mathsf{path}(A_i, A_{i+1})$ for each A_{i+1} in ρ.

    We show that δ maps distinct $\mathcal{X}_R$ paths in S_1 from r_1 to distinct $\mathcal{X}_R$ paths in S_2 from r_2. Let ρ_1, ρ_2 be distinct $\mathcal{X}_R$ paths from r_1 in S_1. Consider the following two cases. First, ρ_1 is a prefix of ρ_2. That is, $\rho_2 = \rho_1/\rho$ where ρ is nonempty since ρ_1 and ρ_2 are distinct. Then ρ is mapped to a nonempty $\mathcal{X}_R$ path in S_2 by the definition of σ, and thus $\delta(\rho_1) \neq \delta(\rho_2)$; similarly if ρ_2 is a prefix of ρ_1. Second, neither is ρ_1 a prefix of ρ_2 nor ρ_2 is a prefix of ρ_1. Then there exist $\rho, \rho'_1, A[q], B_1[q_1]$ and $B_2[q_2]$ such that $\rho_1 = \rho/A[q]/B_1[q_1]/\rho'_1$, $\rho_2 = \rho/A[q]/B_2[q_2]/\rho'_2$, and $B_1[q_1], B_2[q_2]$ are the first labels that differ in ρ_1 and ρ_2. Then B_1, B_2 are child types of A, A is either a concatenation type or a disjunction type, and moreover, either B_1, B_2 are distinct labels, or q_1, q_2 indicate different positions of the same label. By the definition of schema embedding, neither is $\mathsf{path}(A, B_1)$ a prefix of $\mathsf{path}(A, B_2)$ nor the other way around. That is, $\mathsf{path}(A, B_1) = \varrho/\eta_1/\varrho_1$ and $\mathsf{path}(A, B_2) = \varrho/\eta_2/\varrho_2$ such that η_1 and η_2 are distinct. Thus $\delta(\rho_1) \neq \delta(\rho_2)$.

(2) From (1) and the definition of σ_d it follows that σ_d is injective. Indeed, any node in an XML tree is uniquely determined by an $\mathcal{X}_R$ path from the root. Thus by the definition of σ_d, any node v in $T \in \mathcal{I}(S_1)$ is mapped to a distinct node in $\sigma_d(T)$. Indeed, this obviously holds if the parent of v is of a concatenation or disjunction type or str; and moreover, if the type of the parent v' of v is defined with a Kleene star, the children of v' are mapped to distinct nodes preserving the original order, by the definition of σ_d.

(3) We next show that σ_d is type safe; that is, for any $T \in \mathcal{I}(S_1)$, $\sigma_d(T)$ conforms to S_2. One possible violation of S_2 may occur when there exists an A-node u in $\sigma_d(T)$ such that A is a disjunction type $A \to B_1 + \cdots + B_k$, and $\sigma_d(T)$ forces the presences of both B_i and B_j children of u. Then there must be two nodes v_1, v_2 in T identified by $\mathcal{X}_R$ paths ρ_1, ρ_2 from the root of T such that v_1, v_2 are mapped to the B_i, B_j children of u, respectively; furthermore,

$\rho_1 = \rho/A'[q]/\rho'_1$, $\rho_2 = \rho/A'[q]/\rho'_2$, and v_1, v_2 have the lowest common ancestor v that is identified by $\rho/A'[q]$ and is mapped to either u or an ancestor u' of u by σ_d. If v is mapped to u then v must have a disjunction type by the definition of schema embedding, and thus v_1, v_2 cannot coexist. This contradicts the assumption. Now assume that v is mapped to u'. Consider the following cases of the production of A', where A' is the element type of v. (i) Obviously if $A' \to \epsilon$ or $A' \to \mathsf{str}$, it is impossible for v to have descendants v_1 and v_2. (ii) If $A' \to B'_1, \ldots, B'_n$, then both $\mathsf{path}(A', B'_i)$ and $\mathsf{path}(A', B'_j)$ must be suffixes of the same $\mathcal{X}_R$ path that is mapped from $\rho/A'[q]$, ending with A, since otherwise it would violate the path type condition given that A is a disjunction type. However, this contradicts the prefix-free condition of schema embedding since either $\mathsf{path}(A', B'_i)$ is a prefix of $\mathsf{path}(A', B_j)$ or the other way around. (iii) If $A' \to B'_1 + \cdots + B'_n$, then either v_1, v_2 cannot exist at the same time, or v cannot be the lowest common ancestor of v_1 and v_2, since v has one and only one child. Again this leads to contradiction. (iv) If $A' \to B^*$, then by the definition of σ_d, v_1, v_2 cannot be mapped to nodes that have a common ancestor u without violating the path type condition, given that A has a disjunction type. This again contradicts the assumption. Putting these together, the violation of S_2 by OR paths cannot happen. Similarly, it can be verified that violations cannot be caused by AND and STAR paths either. $\square$

4.3 Instance Level Mapping with XSLT

While the given procedure clearly demonstrates that instance-level schema embeddings can be computed in a straightforward and efficient manner, it requires an implementation effort on the part of the user. In fact, the top-down rule-driven style of computation on which instance-level embedding is based is quite compatible with the widely used XML Stylesheet Language for Transformations (XSLT) [Clark 1999]. In this section, we sketch the construction of XSLT stylesheets to effect both σ_d and σ_d^{-1}, the construction of the original document. To simplify the discussion we omit the concrete construction of node id mapping $\mathsf{idM}()$, which, as remarked earlier, can be easily incorporated.

XSLT Overview. We now present a (somewhat simplified) model of XSLT stylesheet processing. In our model, an XSLT stylesheet X is a set of template rules $\{r_i\}$. Each rule, $r_i \in X$, is a 3-tuple ($\mathsf{match}(r_i)$, $\mathsf{mode}(r_i)$, $\mathsf{output}(r_i)$), where $\mathsf{match}(r_i)$ is the *match pattern* of r_i, $\mathsf{mode}(r_i)$ is the *mode* of r_i, and $\mathsf{output}(r_i)$ is the *output-tree fragment* of r_i. The match pattern of a template rule, $\mathsf{match}(r_i)$, is a *pattern* [Clark 1999] and is essentially a subset of XPath expressions containing only *child*, *descendant*, and *attribute* axes. The mode of a rule, $\mathsf{mode}(r_i)$, is a symbol that allows rules to be partitioned; that is, rule invocations must match in mode as well as match pattern. If there is no mode attribute, the XSLT processor will set it to a default value.

The output tree fragment for a rule, $\mathsf{output}(r_i)$, controls the structure of the rule's output. Essentially, $\mathsf{output}(r_i)$ is a well-formed fragment of XML, except that some leaf-nodes of the fragment may be *apply-template* nodes. We use $\mathsf{apply}(r_i)$ to denote the set of apply-template nodes in the output fragment of

r_i. An *apply-templates node*, a_j, is a 2-tuple of the form $(\mathsf{select}(a_j), \mathsf{mode}(a_j))$, where $\mathsf{select}(a_j)$ is the *select expression* of a_j, and $\mathsf{mode}(a_j)$ is the *mode* of a_j.

XSLT Processing Model. We will now informally describe the processing model of XSLT; for a formal discussion see Wadler [2000]. An XSLT stylesheet is executed against a *source document* T_s to produce a *target document* T_t. Document processing revolves around a set C of source nodes in T_s, referred to as *context nodes*. Associated with each node $c \in C$ is a node $t(c)$ in the (partially constructed) T_t. Processing of T_s with XSLT stylesheet X proceeds in general by selecting a node c from C and a processing rule $r_i \in x$ that *matches* c, then replacing $t(c)$ in T_t with a copy o_i of $\mathsf{output}(r_i)$. Each apply-template node $a_j \in \mathsf{output}(r_i)$ is then visited, and the XPath expression $\mathsf{select}(a_j)$ evaluated on T_s, using c as the start node, to yield a sequence of source nodes, R_j. For each node u in R_j (in order), a new dummy node $t(u)$ is created, forming a sequence of dummy target nodes, G_j. The apply-template node a_j in o_i is then replaced by the sequence G_j, and R_j is added to C. This process continues until C is empty.

An XSLT Template for σ_d^{-1}. Consider computing $\sigma_d^{-1}(T)$ for a source DTD $S_1 = (E_1, P_1, r_1)$, target DTD $S_2 = (E_2, P_2, r_2)$, mapping $\sigma = (\lambda, \mathsf{path})$, and a particular element type $C = \lambda(A)$, where $A \in E_1$ and $C \in E_2$. We define $\mathsf{invt}(C)$ to be one or more XSLT templates. The idea is to define a stylesheet comprised of templates for each element in the *image* of S_1 under λ, such that when the stylesheet is run on an instance produced by the instance-level mapping function σ_d applied to a tree, T_1, it will compute the inverse, σ_d^{-1}, recovering T_1. We now give the definition of each such template. For each such template $r_C = \mathsf{invt}(C)$, recall that we must define $(\mathsf{match}(r_C), \mathsf{mode}(r_C), \mathsf{output}(r_C))$. In this construction $\mathsf{match}(r_C)$ will be C, and $\mathsf{mode}(r_C)$ will always be MDATA, a fixed mode picked for the templates in σ_d^{-1}. The output tree fragment, $\mathsf{output}(r_C)$, always has a root element e_r, labeled A, where $C = \lambda(A)$. The sequence of children of this node differs based on the form of $P_1(A)$, as follows:

(1) $P_1(A)$ is $B_1, \ldots, B_n$, then the children of e_r will be a sequence of apply-template nodes, $a_1, \ldots, a_n$. For a_i, $\mathsf{select}(a_i) = \mathsf{path}(A, B_i)$.

(2) $P_1(A)$ is $B_1 + \cdots + B_n$, then instead of a single template for invt of C, we create n templates, $r_{C_1}, \ldots r_{C_n}$. Each r_{C_i} has as the match condition $\mathsf{match}(r_{C_i}) = C[\mathsf{path}(A, B_i)]$, which will match any C node with $\mathsf{path}(A, B_i)$ as an outgoing path. Each of these rules has for its output tree a node e_r with label A, and e_r will have a single child node that is an apply-template node, a, with $\mathsf{select}(a) = \mathsf{path}(A, B_i)$.

(3) $P_1(A)$ is B^*, then a single rule r_C will be created. The root node of $\mathsf{output}(r_C)$ will be labeled A, and it will have a single child node that is an apply-template node, a. The mode of a will be MDATA, and $\mathsf{select}(a) = \mathsf{path}(A, B)$. Note that on the target document, this selection path may return multiple nodes.

(4) $P_1(A)$ is str, then a single rule r_C will be created. The root node of $\mathsf{output}(r_C)$ will be labeled A, and it will have a single child node that is an apply-template node, a. The mode of a will be MDATA, and $\mathsf{select}(a) = \mathsf{text}()$.

Note that text() is an XPath function that returns true when applied to a str node.

Finally, we add a template that matches a text node and generates an output tree that is a copy of that node.

Example 4.5. Consider the target type *course* in Example 4.2 mapped to source type *class* under σ_d^{-1}. The XSLT template generated for *course* in the implementation of σ_d^{-1} would be:

```
<xsl:template match="course" >
   <class>
      <xsl:apply-templates select="basic/cno" />
      <xsl:apply-templates select="class/semester/title" />
      <xsl:apply-templates select="category" />
   </class>
</xsl:template>
```

Note that this template follows closely the tree shown in Figure 4, with each node in hleaf replaced with an apply-templates node. However, the form of production fragments is not as simple with other nodes, and we now show the two templates that are generated for *category*, since it is mapped to *type* which is a disjunctive node with two children in S_0:

```
<xsl:template match="category[mandatory/regular]" >
   <category>
      <xsl:apply-templates select="mandetory/regular" />
   </category>
</xsl:template>
<xsl:template match="category[advanced/project]" >
   <category>
      <xsl:apply-templates select="advanced/project" />
   </category>
</xsl:template>
```

Before defining an XSLT template for the instance-level mapping, σ_d, we introduce template rules to generate default values for elements:

Minimum Default Templates. For a DTD $S_1 = (E_1, P_1, r_1)$, and a particular element type $A \in E_1$, we define $\mathsf{mint}(A)$ to be the *minimum default template*, an XSLT template r_A with $\mathsf{match}(r_A) = \epsilon$, which will match any node, with $\mathsf{mode}(r_A) = \text{MDATA}$, and with $\mathsf{output}(r_A) = \mathsf{mindef}(r_A)$.

An XSLT Template for σ_d. We now outline the construction of an XSLT stylesheet to compute σ_d. The general idea is to follow the construction algorithm very closely, by providing one or more XSLT template rules for each production in S_1. The form of these rules takes advantage of regularities in the form taken by $\mathsf{pfrag}_A(v)$ depending on the type of the production, $P_1(A)$. In particular, the body of each rule will be one form taken by $\mathsf{pfrag}_A(v)$, and each "hot" node in this form will be replaced by an apply-templates node. We now describe the production of $\mathsf{pfrag}_A(v)$ in each case:

(1) $P_1(A)$ is $B_1, \ldots, B_n$. Note that, except for the node identity of the root node, $\mathsf{pfrag}_A(v)$ is a constant tree, say t_A, w.r.t. v. Accordingly, an XSLT template rule r_A can be constructed with $\mathsf{match}(r_A) =$ 'A' and $\mathsf{output}(r_A) = t_A$ modified by substituting an apply-templates node in $\mathsf{output}(r_A)$ for each node in $\mathsf{hleaf}(t_A)$. In particular, for $u \in \mathsf{hleaf}(t_A)$, we construct an apply template node a_u, with $\mathsf{select}(a_u) = B_i$ if the type of $\mathsf{src}(u)$ would be B_i in an instantiation of this fragment. (Default values for $\mathsf{mode}(r_i)$ are used unless specified otherwise.)

(2) $P_1(A)$ is $B_1 + \cdots + B_n$. In this case, n template rules, $r_{A,1}, \ldots, r_{A,n}$ can be constructed. The i-th template rule, $r_{A,i}$, is constructed as follows. First, we ensure that the variant is only fired when the correct child appears in T_1 by setting $\mathsf{match}(r_{A,i}) = A[B_i]$. The output fragment $\mathsf{output}(r_{A,i})$ is then based on the single path $\mathsf{path}(A, B_i)$, and the single leaf node becomes the only apply-templates node, $a_{A,i}$, of $r_{A,i}$. For this applytemplates node, $\mathsf{select}(a_{A,i}) = B_i$.

(3) $P_1(A)$ is B^*. Since the number of children of v may vary, pfrag_A cannot be precomputed in general. Note however, that by the definition of a valid path function, $\rho = \mathsf{path}(A, B)$ is of the form $\lambda(A)/C_1/\ldots/C_k/C_{k+1}/\ldots/C_n/\lambda(B)$, where C_k is the first type defined in terms of Kleene star in P_2, i.e., $P_2(C_k) = C_{k+1}^*$ and no element type on $\lambda(A)/C_1/\ldots/C_k$, except C_k, has production of this form. Accordingly, pfrag_A will have a constant prefix corresponding to $\lambda(A)/C_1/\ldots/C_k$ regardless of the number of children of v, and this fact can be used to construct a pair of XSLT template rules $r_{A,p}$ (p for "prefix") and $r_{A,s}$ (s for "suffix") for $P_1(A)$ as follows: For $r_{A,p}$, $\mathsf{match}(r_{A,p}) = A$, and the body $\mathsf{output}(r_{A,p})$ is a tree corresponding to the path $\lambda(A)/C_1/\ldots/C_k$. A single apply-templates node $a_{A,p}$ is added as a child of C_k, with $\mathsf{select}(a_{A,p}) = B$, and unlike the other rules for a new mode, $\mathsf{mode}(a_{A,p}) = M_A$. The second template rule, $r_{A,s}$ has that $\mathsf{match}(r_{A,s}) = B$, $\mathsf{mode}(r_{A,s}) = M_A$, and $\mathsf{output}(r_{A,s})$ is a tree corresponding to the path $C_{k+1}/\ldots/C_n$ (note it is the responsibility of the next rule to generate the $\lambda(B)$ node). As a child of the final C_n element in tree, a single apply-templates node $a_{A,s}$ is added, with $\mathsf{select}(a_{A,s}) = \mathsf{B}$. Each node v of type B in the source document will be processed by the rest of the stylesheet, and the result placed as a subtree, as required.

(4) $P_1(A)$ is str. The treatment is the same as (1) except the last node of $\mathsf{path}(A, \mathsf{str})$ in T_2 is a text node holding the same value as the text node in T_1.

Example 4.6. Consider the source type *class* in Example 4.2 mapped to target type *course* under σ_d. The XSLT template generated to effect σ_d for *course* would be:

```
<xsl:template match="class" >
   <course>
      <basic>
         <xsl:apply-templates select="cno" />
         <credit> #s </credit>
         <class>
```

```
        <semester>
          <xsl:apply-templates select="title" />
          <year> #s </year>
          <term> #s </term>
          <instructor> #s </instructor>
        </semester>
      </class>
    </basic>
    <xsl:apply-templates select="type" />
  </course>
</xsl:template>
```

Two templates are generated for *type* since it is a disjunctive node with two children:

```
<xsl:template match="type[regular]" >
  <category>
    <mandatory>
      <apply-templates select="regular" />
    </mandatory>
  </category>
</xsl:template>

<xsl:template match="type[project]" >
  <category>
    <advanced>
      <apply-templates select="project" />
    </advanced>
  </category>
</xsl:template>
```

Finally, the prefix and suffix templates for *db* illustrate the handling of *star* edges. Note that these templates use mode to ensure that no spurious matches are made.

```
<xsl:template match="db" > <!-- prefix template >
  <school>
    <courses>
      <current>
        <apply-templates mode="M-db" select="class" />
      </current>
    </courses>
  </school>
</xsl:template>

<xsl:template match="class" mode="M-db" > <!-- suffix template >
  <xsl:apply-templates select="." />
</xsl:template>
```

4.4 Translation of Regular XPath Queries

In this section, we introduce techniques for translating $\mathcal{X}_R$ queries expressed against the source schema S_1 into queries against the target schema S_2. In particular, if Q is an $\mathcal{X}_R$ query on S_1, and σ is an embedding of S_1 into S_2 then we would like to find a query Q' such that $Q'(\sigma_d(T)) = Q(T)$ for any instance T of S_1. We define next a query translation function Tr with just this property. The approach we take is to translate Q into an automaton representation, and use this representation as the basis for the translation.

To motivate the use of an automaton framework for query translation, consider an apparently simpler translation based directly on the parse tree of Q. Recall that $\mathcal{X}_R$ queries use only the "child::" axis of XPath, albeit possibly in the context of a Kleene-star. An appealing idea is to replace a step "child::A" in a query with $\mathsf{path}(B, A)$ where B is A's parent in S_1. However, this simple translation is not generally correct. First, tags in a DTD do not have a unique parent, and A might appear in the production of a number of elements, say B and C. In this case, simple edge substitution is not sufficient to translate $(B \cup C)/A$, since in general $\mathsf{path}(B, A) \neq \mathsf{path}(C, A)$. Second, consider translating a query $(A_1 \cup A_2 \cup A_3)/(B_1 \cup B_2 \cup B_3)$. Since each $\mathsf{path}(A_i, B_j)$ is distinct, and may or may not be defined, it is clear that nine paths may need to be matched when this query is translated to S_2, and thus simple textual substitutions will not effectively map this query across σ. Simply substituting $\mathsf{path}(A, B)$ for A may lead to incorrect translation.

Finally, we note that keeping the output in automata form is important to the overall complexity of query translation. If the translated query $\mathsf{Tr}(Q)$ is explicitly represented as an $\mathcal{X}_R$ query, in the worst case it is of exponential size. Indeed, the query translation problem subsumes the problem for translating finite state automata to regular expressions, which is known to be exponential-time complete [Ehrenfeucht and Zeiger 1976]. To overcome these difficulties, we adopt a simple automaton (graph) representation of the translated queries, along the same lines as solutions for traditional path problems [Tarjan 1981]. This allows us to translate $\mathcal{X}_R$ queries on S_1 to equivalent queries, represented as automata, against S_2 as embedded by σ in low polynomial time.

We now define a simple automaton representation of $\mathcal{X}_R$ queries. We refer to such an automaton as an *annotated nondeterministic finite state automaton* (ANFA). As will be seen shortly, for each $\mathcal{X}_R$ query Q posed on instances T of S_1, there exists an ANFA characterizing the equivalent $\mathcal{X}_R$ query $\mathsf{Tr}(Q)$ on corresponding target documents $\sigma_d(T)$; furthermore, the size of the ANFA is bounded by $O(|Q|\,|\sigma|\,|S_1|)$. Leveraging ANFA's, we will then provide the definition of Tr.

Automaton Representation of $\mathcal{X}_R$ Queries. We represent an $\mathcal{X}_R$ query Q in terms of a mild extension of non-deterministic finite state automaton (NFA), by annotating states in a NFA with ANFA's that represent qualifiers in Q. Formally, the ANFA representing Q is defined to be $\mathcal{M}_Q = (M, \nu)$, where ν and M are given as follows. (i) ν is a mapping, referred to as *the annotation* of $\mathcal{M}_Q$, from a set of names $\{X_i \mid i \in [1, m]\}$ to a set of ANFA's $\{M_i \mid i \in [1, m]\}$, and M_i is an ANFA representing a subqualifier. Here a subquery of Q denotes a descendant in Q's parse tree, and a subqualifier denotes a descendant in the parse tree of

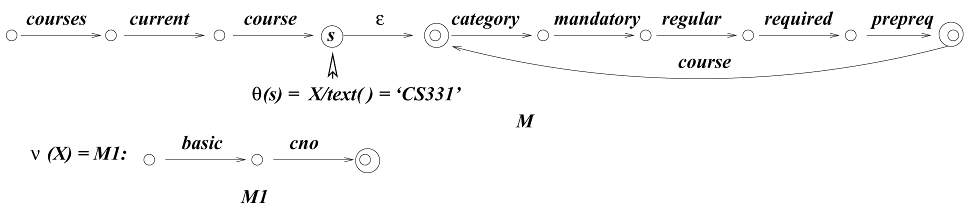


Fig. 6. An ANFA representing the $\mathcal{X}_R$ query Q'.

a qualifier of Q. Note that m is the number of subqualifiers in Q, and thus is no larger than the size $|Q|$ of Q. (ii) $M = (K, \Sigma, \delta, s, F, \theta)$, where K, Σ, δ, s, F are the states, alphabet (the labels in Q), transition function, start state and final states as in the standard NFA definition; and θ is a partial mapping from K to "qualifiers" defined as

$$q_x ::= X \;\mid\; X/\mathit{text}() = \text{`}c\text{'} \;\mid\; \mathit{position}() = k \;\mid\; \neg X \;\mid\; X \wedge X \;\mid\; X \vee X,$$

where X is a name. That is, θ annotates a state in M with a qualifier q_x.

Example 4.7. Consider a query Q' = *courses/current/course[basic/cno/text()='CS331']/(category/mandatory/regular/required/prereq/course)**. This $\mathcal{X}_R$ query is represented by the ANFA $\mathcal{M}$ shown in Figure 6, where $\mathcal{M} = (M, \nu)$ and $\nu(X) = \mathcal{M}1$.

More specifically, we define M_Q based on the structure of Q as follows.

(a) If Q is ϵ, then $M = (\{s\}, \Sigma, \delta, s, \{s\}, \theta)$, where $\delta(s, \epsilon)$ is the only defined transition (ϵ-transition). The mappings $\theta()$ and $\nu()$ are undefined.

(b) If Q is a label B, then $M = (\{s, f\}, \Sigma, \delta, s, \{f\}, \theta)$, where $\delta(s, B) = \{f\}$ is the only defined transition. The mappings $\theta()$ and $\nu()$ are undefined.

(c) If Q is $Q_1 \cup Q_2$, assume that $\mathcal{M}_{Q_i} = (M_{Q_i}, \nu_{Q_i})$ is the ANFA representing Q_i for $i \in [1, 2]$. We define M_Q as the union of M_{Q_1} and M_{Q_2} as for standard NFA's [Yu 1996], and $\theta()$ as the union of $\theta_1()$ in M_{Q_1} and $\theta_2()$ in M_{Q_2}; similarly for $\nu()$. We assume w.l.o.g. that the name X_q of a subqualifier q is uniquely identified by q in both ν_{Q_1} and ν_{Q_2}.

As for standard NFA's, M_Q can be defined for Q_1/Q_2 and Q_1^* in terms of the concatenation and Kleene closure of ANFA's, respectively.

Similarly, $p/\mathit{text}()$ is a special case of Q_1/Q_2 in which Q_2 is represented by an ANFA with a single transition defined by str.

(d) If Q is $p[q]$, assume that $\mathcal{M}_p = (M_p, \nu_p)$ and $\mathcal{M}_q = (M_q, \nu_q)$ are the ANFA's representing p and q, respectively, where $M_p = (K_p, \Sigma, \delta_p, s, F_p, \theta_p)$. We define M_Q to be $(K_p, \Sigma, \delta_p, s, F_p, \theta)$, where $\theta()$ is an extension of $\theta_p()$ by letting $\theta(f) = X_q$, for all $f \in F_p$, and $\nu()$ be an extension of the union of ν_p and ν_q by letting $\nu(X_q) = \mathcal{M}_q$.

The remaining cases cover the translation of a qualifier (Boolean expression) q in an $\mathcal{X}_R$ query (e.g., $p[q]$) to an ANFA form. Recall from Section 2.2 the definition of qualifiers. The translation of q is inductive based on the structure of q, in which the ANFA for a subexpression of q may be constructed by one of the cases (a)-(d) and the cases below.

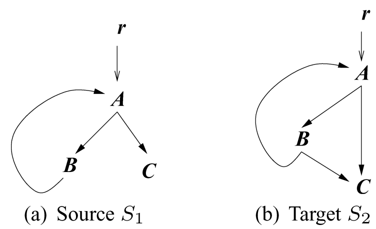


Fig. 7. Problematic DTDs for Query Translation.

(e) If q is p, assume that $\mathcal{M}_p = (M_p, \nu_p)$ is the ANFA representing p. Then M is defined as in (a), except that $\theta(s) = X_p$ and $\nu(X_p) = \mathcal{M}_p$.
(f) If q is $p/text() = c$, assume that $\mathcal{M}_p = (M_p, \nu_p)$ is the ANFA representing p. Then M_q is defined as in case (a), except that $\theta(s) = [X_p/text() = c]$ and $\nu(X_p) = \mathcal{M}_p$.
(g) If q is $position() = k$, then M is defined as in case (a), except that $\theta(s) = [position() = k]$.
(h) If q is $\neg q_1$, assume that $\mathcal{M}_{q_1} = (M_{q_1}, \nu_{q_1})$ is the ANFA representing q_1. Then M is defined as in case (a), except that $\theta(s) = [\neg X_{q_1}]$, and $\nu()$ is an extension of ν_q by letting $\nu(X_{q_1}) = \mathcal{M}_{q_1}$.
(i) If q is $q_1 \wedge q_2$, assume that $\mathcal{M}_{q_i} = (M_{q_i}, \nu_{q_i})$ is the ANFA representing q_i for $i \in [1, 2]$. We define M_q as in case (a), except that $\theta(s) = [X_{q_1} \wedge X_{q_2}]$, and $\nu()$ is an extension of the union of ν_{q_1} and ν_{q_2} by letting $\nu(X_{q_i}) = \mathcal{M}_{q_i}$. Similarly for the case when q is $q_1 \vee q_2$.

Along the same lines as the translation from standard NFA's to regular expressions [Yu 1996], one can restore an $\mathcal{X}_R$ query Q from its ANFA representation $\mathcal{M}_Q$. In fact it is easy to develop an algorithm for directly evaluating the ANFA $\mathcal{M}_Q$ on an XML tree following the semantics of $\mathcal{X}_R$ query evaluation on XML trees [Marx 2004]. Indeed, evaluation algorithms and optimization techniques have been developed for a similar automaton representation of regular XPath queries [Fan et al. 2007], which have shown to outperform several commercial systems for evaluating XPath queries (to the best of our knowledge, a commercial system for regular XPath is not yet in place). We do not elaborate the evaluation algorithms as they are beyond the scope of this paper.

Schema-Directed Query Translation. We now present the definition of Tr. Recall that Tr is a mapping from $\mathcal{X}_R$ queries to ANFA's that, given any XML query $Q \in \mathcal{X}_R$, yields an ANFA $\mathsf{Tr}(Q)$ representing an $\mathcal{X}_R$ query Q' such that for any $T \in \mathcal{I}(S_1)$, $Q(T) = Q'(\sigma_d(T))$. The translation function Tr combines the definition of ANFA's $\mathcal{M}_Q$ for a query Q with the mapping δ on $\mathcal{X}_R$ paths given in Section 4.2.

This translation is *schema-directed* in that a translation of each subexpression q of Q is made *relative* to each element type A appearing in schema S_1. To see why a schema-directed translation is required, consider the source and target schemas of Figure 7. In this figure, a very simple schema embedding is considered in which, for all tags X, Y, $\lambda(X) = X$ and $\mathsf{path}(Y, X) = X$ (if (Y, X) is an edge in the source schema), when X, Y range over r, A, B and C. One might be tempted to think that query translation would be simple in this case, and that Q itself would work on the target schema. Indeed, simply substituting $\mathsf{path}(Y, X)$ for each edge (Y, X) in the source schema would yield the same query Q on target documents specified by the schema shown in Figure 7(b). However, consider a query $Q = r/(A \cup B \cup C)^*$. Clearly, such a query does not return any C children of B elements when run on S_1, but will return such nodes when run on $\sigma_d(T)$, since the C-child of B is a required node and will be added by the instance level mapping of Section 4.2. It is thus evident that the simple translation strategy by substituting $\mathsf{path}(Y, X)$ for (Y, X) does not work. Our schema-directed construction avoids such problems by matching target nodes only when they will be generated by nodes in the source schema by $\sigma_d()$.

More specifically, given an $\mathcal{X}_R$ query Q over S_1, $\mathsf{Tr}(Q)$ is computed by using the following functions. (i) For each element type A in E_1 and each subquery Q_1 of Q, the *local translation* $\mathsf{Tr}_l(Q_1, A) = (M, \nu)$ is the ANFA presenting an $\mathcal{X}_R$ query Q_2 over S_2 such that for any instance T of S_1 and any element a of type A in T, the result of evaluating Q_1 at a in T is the same as the result of evaluating Q_2 at a' on $\sigma_d(T)$, where a' is mapped from a by σ_d. (ii) For each final state f in M, the *label* $\mathsf{lab}(f, M, A)$ associates an element type of S_1 with f, which indicates the type of the elements *reached* via Q_1 when evaluated at an A element in an instance T of S_1. As will be seen shortly, the construction of the ANFA (M, ν) ensures that each final state f in M is associated with a single type of S_1.

To handle the case that $\mathsf{Tr}_l(Q_1, A)$ will never return any nodes, we introduce a special automaton Fail, which can be thought of as consisting of a single start state with no transitions and no final states. Note that any automaton with no final states is equivalent to Fail, and thus special cases can be introduced in other rules below to handle the case that an automaton for a subexpression is equivalent to Fail. However, to keep the presentation simple, we assume that a standard useless state removal algorithm is run on each completed automaton, which removes states that cannot reach a final state.

We compute $\mathsf{Tr}_l(Q_1, A)$ and $\mathsf{lab}()$ based on the structure of query Q_1 as follows.

(a) If Q_1 is ϵ, $\mathsf{Tr}_l(Q_1, A)$ is the ANFA as defined in case (1a), and $\mathsf{lab}(s, M, A) = A$.

(b) If Q_1 is a label B and $\mathsf{path}(A, B)$ is defined (i.e., $B \in P(A)$), then $\mathsf{Tr}_l(Q_1, A)$ is the ANFA coding the $\mathcal{X}_R$ query $\mathsf{path}(A, B)$, and $\mathsf{lab}(f, M, A) = B$, where f is the (single) final state of M. If Q_1 is a label B and $\mathsf{path}(A, B)$ is not defined, then M_{Q_1} is Fail.

(c) If Q_1 is $p_1 \cup p_2$, then $\mathsf{Tr}_l(Q_1, A)$ is the union of the ANFA's $\mathsf{Tr}_l(p_1, A)$ and $\mathsf{Tr}_l(p_2, A)$, and the function $\mathsf{lab}()$ is the union of labels for the final states (if any) of $\mathsf{Tr}_l(p_1, A)$ and $\mathsf{Tr}_l(p_2, A)$. We assume w.l.o.g. that $\mathsf{Tr}_l(p_1, A)$ and $\mathsf{Tr}_l(p_2, A)$ have distinct final states.

(d) If Q_1 is p_1/p_2, assume that $\mathsf{Tr}_l(p_1, A) = (M_1, \nu_1)$ and $L(M_1, A)$ is the set of labels $\{\mathsf{lab}(f, M_1, A) \mid f$ is a final state of $M_1\}$. For each $B \in L(M_1, A)$, let $\mathsf{Tr}_l(p_2, B) = (M_B, \nu_B)$ be the ANFA representing the local translation of p_2 at context type B. Then the ANFA $\mathsf{Tr}_l(Q_1, A)$ is defined to be (M, ν), where M is the concatenation of M_1 and all M_B's by connecting via ϵ-transition the final state f of M_1 and the start state of M_B if $\mathsf{lab}(f, M_1, A) = B$, ν is the union of ν_1 and all the ν_B's. The set F of final states of M is the union of the final states for the M_B's, and similarly the label function for the final states of M is defined to be the union of those label functions for M_B's.

Similarly for the case when Q_1 is $p/text()$, except that here $\mathsf{Tr}_l(p_2, B)$ represents the $\mathcal{X}_R$ query $\mathsf{path}(B, \mathsf{str})$, and all the final states of $\mathsf{Tr}_l(p_2, B)$'s are merged into a single final state f_s, for which $\mathsf{lab}(f_s, M, A) = \mathsf{str}$.

(e) If Q_1 is $p[q]$, assume that $\mathsf{Tr}_l(p, A) = (M_p, \nu_p)$ and $L(M_p, A)$ is the set of labels $\{\mathsf{lab}(f, M_p, A) \mid f \text{ is a final state of } M_p\}$. For each $B \in L(M_p, A)$, let $\mathsf{Tr}_l([q], B) = (M_B, \nu_B)$ be the ANFA representing the local translation of $[q]$ at context type B. Then $\mathsf{Tr}_l(Q_1, A)$ is defined to be (M, ν), where M is an extension of M_p such that for any final state f of M_p, $\theta(f) = X_B$ if $\mathsf{lab}(f, M_p, A) = B$, and $\nu(X_B) = \mathsf{Tr}_l([q], B)$. The label function of $\mathsf{Tr}_l(Q_1, A)$ is the same as that of $\mathsf{Tr}_l(p, A)$.

We now present the translation of qualifiers q.

(f) If q is p, then $\mathsf{Tr}_l(q, A)$ is defined as in case (2a), except that $\theta(s) = X_p$ and $\nu(X_p) = \mathsf{Tr}_l(p, A)$.

(g) If q is $p/text() = c$, then $\mathsf{Tr}_l(q, A)$ is defined as in case (2a), except that $\theta(s) = [X/text() = c]$ and $\nu(X) = \mathsf{Tr}_l(p, A)$.

(h) If q is $position() = k$, then $\mathsf{Tr}_l(q, A)$ is defined as in case (2a), except that $\theta(s) = [position() = k]$.

(i) If q is $\neg q_1$, then $\mathsf{Tr}_l(q, A)$ is defined as in case (2a), except that $\theta(s) = [\neg X]$, and $\nu()$ is an extension of ν_{q_1} by letting $\nu(X) = \mathsf{Tr}_l(q_1, A)$, where ν_{q_1} is the annotation of $\mathsf{Tr}_l(q_1, A)$.

(j) If q is $q_1 \wedge q_2$, $\mathsf{Tr}_l(q, A)$ is defined as in case (2a), except that $\theta(s) = [X_1 \wedge X_2]$, and $\nu()$ is an extension of the union of ν_{q_1} and ν_{q_2} by letting $\nu(X_i) = \mathsf{Tr}_l(q_i, A)$, where ν_{q_i} is the annotation of $\mathsf{Tr}_l(q_i, A)$ for $i \in [1, 2]$.

Similarly for the case when q is $q_1 \vee q_2$.

Finally we consider translation of Kleene-star constructs.

(k) If Q_1 is p^*, the computation of $\mathsf{Tr}_l(Q_1, A)$ requires an iteration. For each element type B in S_1, we define a Boolean flag $\mathsf{visited}(B)$, which is initially set to false. The computation is conducted as follows. We first compute $\mathsf{Tr}_l(p, A)$ and set $\mathsf{visited}(A)$ to true. Let $\mathcal{M} = (M, \nu)$ be $\mathsf{Tr}_l(p, A)$. While there exists a final state f in M such that (a) $\mathsf{lab}(f, M, A) = B$, (b) $\mathsf{visited}(B)$ is false for some B in S_1 and (c) $\mathsf{Tr}_l(p, B) \neq \mathsf{Fail}$, then assuming that $\mathsf{Tr}_l(p, B) = (M_B, \nu_B)$, the following three steps are taken: (i) Concatenate M with M_B by connecting f to the initial state of M_B via ϵ-transition and extend the states, transition function, final states and the θ function of M by including the counterparts of M_B. The initial state of M remains unchanged. (ii) Set $\mathsf{visited}(B)$ to true. (iii) Extend ν by including ν_B. This process is repeated at most $|S_1|$ times since each iteration marks at least one element type in S_1 to true. Upon the completion of the process, we extend the final states of M by including its initial state (to capture the case of p^0), and define $\mathsf{Tr}_l(Q, A)$ to be $\mathcal{M}$ obtained as above.

Given these, we define $\mathsf{Tr}(Q)$ to be $\mathsf{Tr}_l(Q, r_1)$. The proof of the following theorem is straightforward by induction on the structure of Q.

THEOREM 4.2. *The functions* $\mathsf{Tr}_l(Q_1, A)$ *and* $\mathsf{lab}()$ *are well defined, and for any instance* T *of* S_1*,* $Q(T) = Q'(\sigma_d(T))$*, where* Q' *is the* $\mathcal{X}_R$ *query represented by* ANFA $\mathsf{Tr}(Q)$.

Example 4.8. Consider $Q =$ *class[cno/text()='CS331']/(type/regular/prereq/class)**. Over the DTD S_0 of Figure 1(a), this $\mathcal{X}_R$ query is to find all the classes that are (direct or indirect) prerequisites of CS331. It is translated to the ANFA shown in Figure 6, which represents the $\mathcal{X}_R$ query Q' given in Example 4.7. Over the DTD S of Figure 1(c), Q' is equivalent to Q w.r.t. the mapping σ_d given in Example 4.4, i.e. $Q(T) = Q'(\sigma_d(T))$ for any $T \in \mathcal{I}(S_0)$, when evaluated on T with the root as the context node.

In contrast, the notion of graph similarity ensures neither invertibility nor query preservation w.r.t. $\mathcal{X}_R$. As an example, the source and target schemas in Figure 3(a) are bisimilar by the conventional definition of graph similarity, which does not consider cardinality constraints of different DTD constructs. However, there exists no instance-level mapping from the source to the target with an inverse mapping or a query translation function.

4.5 Properties of Schema Embeddings

Theorem 4.1 has shown that the XML mapping σ_d of a valid schema embedding σ is guaranteed to be type safe. We next show that σ_d and σ have all the other desired properties.

Information Preservation. In contrast to Theorem 3.4, information preservation is guaranteed by schema embeddings. Recall the fragment $\mathcal{X}_R$ of regular XPath from Section 2.

THEOREM 4.3. *The* XML *mapping* σ_d *of a valid schema embedding* $\sigma : S_1 \to S_2$ *is invertible and is query preserving w.r.t.* $\mathcal{X}_R$*. More precisely, (a) there exists an inverse* σ_d^{-1} *of* σ_d *that, given any* $\sigma_d(T)$*, recovers* T *in* $O(|\sigma_d(T)|^2)$ *time; and (b) there is a query translation function* Tr *that given any* $\mathcal{X}_R$ *query* Q *over* S_1*, computes an* $\mathcal{X}_R$ *query* $\mathsf{Tr}(Q)$ *equivalent w.r.t.* σ_d *over* S_2*; furthermore, the query* $\mathsf{Tr}(Q)$ *has a bounded size* $O(|Q|\,|\sigma|\,|S_1|)$ *and can be computed in* $O(|Q|^2\,|\sigma|\,|S_1|^2)$ *time if it is represented in an automaton format.*

PROOF. To show that σ_d is invertible and query preserving w.r.t. $\mathcal{X}_R$, it suffices to define a query translation function $\mathsf{Tr} : \mathcal{X}_R \to \mathcal{X}_R$. For if it holds, then σ_d is query preserving w.r.t. $\mathcal{X}_R$ and in addition, by Theorem 3.3 it is also invertible. By Theorem 4.2, Tr is just such a query translation function.

For the complexity of the query translation function Tr, recall the definition of Tr given in Section 4.4. Note that the set of final states in an ANFA is bounded by $|S_1|$ and that the size of the ANFA $\mathsf{Tr}_l(Q_1, A)$ is bounded by $O(|Q|\;|\mathsf{path}|\;|S_1|)$, which is less than $O(|Q|\;|\sigma|\;|S_1|)$. The computation of $\mathsf{Tr}_l(Q_1, A)$ can be conducted by dynamic programming, and it takes at most $O(|Q|^2\;|\sigma|\;|S_1|^2)$ time to compute $\mathsf{Tr}(Q)$.

The inverse function σ_d^{-1} is defined along the same lines as the function in the proof of Theorem 3.3. Given any $\sigma_d(T)$ in $\mathcal{I}(S_2)$, it takes at most $O(|\sigma_d(T)|^2)$ time to compute the source instance T. □

Multiple Sources. In contrast to graph similarity, it is possible to embed multiple source DTD schemas to a single target DTD, as illustrated by the example below. This property is particularly useful in data integration.

Example 4.9. The embedding $\sigma_2 = (\theta_2, \mathsf{path}_2)$ below maps the source DTD S_1 of Figure 1(b) to the target DTD S of Figure 1(c).

λ_2(db) = school
$\lambda_2(A)$ = A /* A: student, ssn, name, taking, cno */

path_2(db, student) = students/student
path_2(student, B) = B /* B: ssn, name, taking */
path_2(taking, cno) = cno
path_2(C, str) = *text()* /* C: ssn, name, cno */

Taken together with σ_1 of Example 4.2, this allows us to integrate a *course* document of S_0 and a *student* document of S_1 into a single *school* instance of the target DTD S.

In general, given multiple source DTDS $S_1, \ldots, S_n$ and a single target DTD S, it is possible to define schema embeddings $\sigma_i : S_i \to S$ to simultaneously map S_i to S. In a nutshell, this can be done as follows. Assume that $S_i = (E_i, r_i, P_i)$, and assume a certain order on $S_1, \ldots, S_n$. To simplify the discussion, first assume that $E_i \cap E_j = \emptyset$ for all $i \neq j$. We can then define a single source DTD $S' = (E', r', P')$ such that $E' = E_1 \cup \ldots \cup E_n$, $r' \to P_1(r_1), \ldots, P_n(r_n)$ (following the order on the source DTDS), and for each $A \in E'$, $P'(A) = P_i(A)$ if $P_i(A)$ is defined. Intuitively, S' merges multiple sources into a single source. Thus if there exists an embedding σ from S' to S, σ can be decomposed into $\sigma_i : S_i \to S$ such that the instance-level mapping σ_d^i is invertible and query preserving w.r.t. $\mathcal{X}_R$. In the general setting, if E_i and E_j are not disjoint, one can define S' as a specialized DTD [Papakonstantinou and Vianu 2000] to preserve the element type definitions of P_i and P_j on common element types. It is natural to extend the definition of schema embedding to specialized DTDS.

One can treat the target schema S as a global schema, and the XML mappings $\sigma_i : S_i \to S$ as the definition of a global view of multiple sources $S_1, \ldots S_n$, following the global-as-view approach. The invertibility of σ_i assures that the view is *exact* [Lenzerini 2002]. The need for the query preservation is evident in this context since one wants to be able to query the source data via the global view. As observed in Fagin [2006], invertibility is also useful in defining new views via mapping compositions, and in data migration where the user may decide to "roll back" to the original data source; furthermore, it is helpful in data provenance, when one needs to recover the original source to trace the origin of certain data [Buneman et al. 2001].

Small Model Property. The result below gives us an upper bound on the length $|\mathsf{path}(A, B)|$, and allows us to reduce the search space when defining or finding an embedding.

THEOREM 4.10. *If there exists a valid schema embedding $\sigma : S_1 \to S_2$, then there exists one such that for any edge (A, B) in S_1, $l = |\mathsf{path}(A, B)| \leq (k+1)\,|E_2|$,* *where* $S_1 = (E_1, P_1, r_1)$*,* $S_2 = (E_2, P_2, r_2)$*, and* k *is the size of the production* $P_1(A)$*. More specifically,*

—$|\mathsf{path}(A, B)| \leq k\ |E_2|$ *if* A *is a concatenation type;*

—$|\mathsf{path}(A, B)| \leq (k + 1)\ |E_2|$ *if* A *is a disjunction type;*

—$|\mathsf{path}(A, B)| \leq 2\ |E_2|$ *if* A *is a Kleene closure;*

—$|\mathsf{path}(A, B)| \leq |E_2|$ *if* B *is* str.

PROOF. Suppose that there is a valid embedding $\sigma : S_1 \to S_2$, where $S_1 = (E_1, P_1, r_1)$ and $S_2 = (E_2, P_2, r_2)$, and $\sigma = (\lambda, \mathsf{path})$. Consider an arbitrary edge (A, B) in S_1.

(1) A is a concatenation type. Then $\mathsf{path}(A, B)$ is an AND $\mathcal{X}_R$ path that can be simplified to one that contains at most k cycles, where k cycles may be necessary to ensure that $\mathsf{path}(A, B)$ is not a prefix of any $\mathsf{path}(A, B')$ for distinct subelement types B, B' of A. Any other cycles can be removed, and all of the k cycles can be made simple cycles (i.e., a cycle that does not contain repeated labels), while the modified σ remains well defined. Thus $|\mathsf{path}(A, B)|$ is bounded by $k\ |E_2|$.

(2) A is a disjunction type. Then $\mathsf{path}(A, B)$ is a disjunction $\mathcal{X}_R$ path that can be simplified to one that contains at most $k+1$ simple cycles: k cycles to ensure that $\mathsf{path}(A, B)$ is not a prefix of any $\mathsf{path}(A, B')$, where B' is another subelement type of A, and an additional cycle to include a dashed edge. After the simplification the modified σ remains well defined. Thus $|\mathsf{path}(A, B)| \leq (k + 1)\ |E_2|$.

(3) A is defined to be a Kleene closure $A \to B^*$. Then $\mathsf{path}(A, B)$ is a STAR $\mathcal{X}_R$ path, which can be simplified such that $\mathsf{path}(A, B)$ contains at most one simple cycle (to include a star edge). Thus $\mathsf{path}(A, B) \leq 2\ |E_2|$.

(4) A is defined to be $A \to$ str. As in (1), $\mathsf{path}(A, B)$ is no longer than $|E_2|$. $\square$

Transformation Language. The nice properties of schema embeddings suggests a language for specifying XML transformations. Given two DTDS S_1, S_2, one can specify a mapping from $\mathcal{I}(S_1)$ to $\mathcal{I}(S_2)$ by defining embedding $\sigma = (\lambda, \mathsf{path})$, that is, specifying a mapping λ from types of S_1 to types of S_2, and a mapping from edges over S_1 to $\mathcal{X}_R$ paths over S_2, both at the schema level in a declarative manner. Such an embedding specification σ yields an XML mapping σ_d of σ that guarantees the following: (1) it is type safe, that is, for any $T \in \mathcal{I}(S_1)$, $\sigma_d(T)$ conforms to the target schema S_2; (2) it is invertible, that is, there exists a quadratic time function σ_d^{-1} such that $\sigma_d^{-1}(\sigma_d(T)) = T$ for any $T \in \mathcal{I}(S_1)$, and (3) it is query preserving w.r.t. $\mathcal{X}_R$, that is, there is a query translation function Tr such that for any $Q \in \mathcal{X}_R$ and any $T \in \mathcal{I}(S_1)$, $Q(T) = \mathsf{Tr}(Q)(\sigma_d(T))$. This language is able to capture XML DTD mappings commonly found in data migration and integration, and provides a practical approach to defining XML schema mappings. One might want to define a mapping in a richer such as XQuery or XSLT. However, this is hampered by the negative results of Section 3, which tell us that the richer the language is, the more difficult it is to identify information-preserving mappings. In addition, many XML mappings found in practice can be expressed by annotating schemas with (regular) XPath expressions, along the same lines as schema embedding.

5. COMPUTING SCHEMA EMBEDDINGS

In this section, we state the computation problem for schema embeddings and briefly summarize several techniques for computing XML schema embeddings. Details of these techniques, as well as an experimental evaluation, are presented in Bohannon et al. [2005].

The problem of computing XML schema embeddings is formally stated as follows:

PROBLEM:	Schema-Embedding
INPUT:	Two DTDs schemas S_1 and S_2 and a similarity matrix att.
OUTPUT:	A schema embedding $\sigma : S_1 \to S_2$ valid w.r.t. att if one exists.

Here $S_1 = (E_1, P_1, r_1)$ and $S_2 = (E_2, P_2, r_2)$.

In practice, it is useful to find an embedding $\sigma : S_1 \to S_2$ with as high a value for qual(σ, att) as possible (recall from Section 4.1 for the definition of qual(σ, att)). The ability to efficiently find good solutions to this problem will lead to an automated tool that, given two DTD schemas, can compute candidate embeddings to recommend to users, or to rank schema matches that are known to participate in an information-preserving embedding higher than those that are not.

It turns out that if att allows *ambiguity*, that is, if a single schema element in S_1 may map to more than one element of S_2, then it is intractable to solve Schema-Embedding. Worse still, it remains NP-hard for *nonrecursive* DTDs even when they are defined in terms of concatenation types only.

THEOREM 5.1. *The Schema-Embedding problem is NP-complete. It remains NP-hard for nonrecursive* DTDs *defined with concatenation types only.*

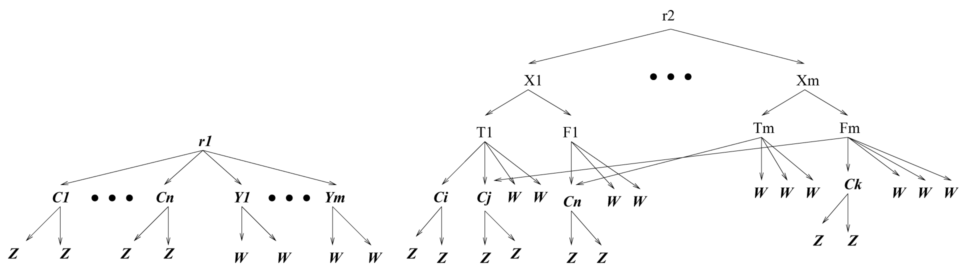
PROOF. An NP algorithm is as follows: guess a mapping, and then check whether it is an embedding. Theorem 4.4 gives maximum sizes that need to be guessed, and thus a mapping can be guessed in polynomial time. Checking whether or not a mapping is a valid embedding can also be done in PTIME.

For the NP-hardness, it suffices to show that the problem is NP-hard for nonrecursive DTDs, by reduction from 3SAT, which is NP-complete [Garey and Johnson 1979]. An instance of 3SAT is a well-formed Boolean formula $\phi = C_1 \wedge \cdots \wedge C_n$ of which we want to decide satisfiability.

Given an instance ϕ of 3SAT, we define two nonrecursive DTDs S_1, S_2 such that ϕ is satisfiable iff there is a valid schema embedding from S_1 to S_2. We define a similarity matrix att such that for all element types A in S_1 and B in S_2, att$(A, B) = 1$, that is, there is no restriction on the mapping. Assume that all the propositional variables in ϕ are $x_1, \ldots, x_m$. We define S_1, S_2 as follows.

$S_1 = (E_1,\ P_1,\ r_1)$, where
$E_1 = \{r_1, Z, W\} \cup \{C_i \mid i \in [1, n]\} \cup \{Y_s \mid s \in [1, m]\}$;
P_1 is defined as:

$r \to C_1, \ldots, C_n,\ Y_1, \ldots, Y_m,$
$C_i \to Z, \ldots, Z;$ /* $n + i$ occurrences of Z */
$Y_s \to W, \ldots, W$ /* $2n + s$ occurrences of W */
$A \to \epsilon$ /* for A ranging over W, Z */



(a) Source S_1 (b) Target S_2

Fig. 8. DTD schemas in the proof of Theorem 5.1.

$S_2 = (E_2,\ P_2,\ r_2)$, where
$E_2 = \{r_2, W, Z\} \cup \{C_i \mid i \in [1, n]\} \cup \{X_s, T_s, F_s \mid s \in [1, m]\}$;
P_2 is defined as:

$r \to X_1, \ldots, X_m$,
$X_i \to T_i, F_i$, /* for $i \in [1, m]$*/
$T_i \to C_{i_1}, \ldots, C_{i_k}, W, \ldots, W$ /* all C_{i_j} in which x_i appears, and $2n + i$ occurrences of W*/
$F_i \to C'_{i_1}, \ldots, C'_{i_k}, W, \ldots, W$ /* all C'_{i_j} in which $\overline{x_i}$ appears, and $2n + i$ occurrences of W */
$C_i \to Z, \ldots, Z$; /* $n + i$ occurrences of Z*/
$A \to \epsilon$ /* for A ranging over W, Z.

The DTDs S_1 and S_2 are depicted in Figure 8(a) and 8(b), respectively. Note that both S_1, S_2 are nonrecursive and are defined in terms of concatenation types only. Intuitively, S_2 encodes ϕ, and S_1 is to assert the existence of a truth assignment to $x_1, \ldots, x_m$ that satisfies all the clauses in ϕ. In both S_1 and S_2, C_i is to code clause C_i, which has a "signature" consisting of $n + i$ occurrences of Z that is to ensure that C_i in S_1 is mapped to C_i in S_2. In S_2, X_j codes the variable x_j in ϕ, which may have either a true value or false, indicated by T_i and F_i, respectively. In DTD S_1, $Y_1, \ldots, Y_m$ also encode variables, and Y_s can only map to T_s or F_s (or some ancestor thereof) in S_2 due to the number of W children below Y_s and T_s, F_s (technically, Y_s could map to some $T_{s'}, s' > s$, but it is easy to see that for every Y_s to map successfully, it is necessary that Y_s maps to T_s or F_s).

The mappings of the Y_s elements are to code the "negation" of a truth assignment μ to variables in ϕ: Y_s is mapped to F_s if $\mu(x_s)$ is true for some $j \in [1, m]$, and Y_s is mapped to T_s if $\mu(x_s)$ is false. To understand how this coding works, consider that clauses are disjunctive, and thus clause C_i can be satisfied by the correct assignment to any x_j appearing in C_i (i.e. to true if x_j appears in C_i, and to false if $\overline{x_j}$ appears in C_i). The paths into C_i thus correspond to the *failure* by a particular variable's assignment to satisfy C_i. If every variable fails to satisfy C_i, then some Y_j will be mapped to *every* ancestor of C_i in S_2, leaving no paths by which the C_i in S_1 can be mapped to C_i in S_2 without violating the prefix-free property. We now formalize this intuition.

We next show that S_1, S_2 are indeed a reduction from 3SAT, that is, there is a valid embedding from S_1 to S_2 iff ϕ is satisfiable. First, suppose that ϕ is satisfiable. Then there exists a truth assignment μ to $x_1, \ldots, x_m$ that satisfies ϕ. We define an embedding $\sigma = (\lambda, \mathsf{path})$ such that $\lambda(C_i) = C_i$, $\lambda(Z) = Z$, $\lambda(W) = W$, $\lambda(Y_i) = F_i$ and $\mathsf{path}(r_1, Y_i) = X_i/F_i$ if $\mu(x_i)$ is true, $\lambda(Y_i) = T_i$ and $\mathsf{path}(r_1, Y_i) = X_i/T_i$ if $\mu(x_i)$ is false. Furthermore, $\mathsf{path}(r_1, C_i)$ is a path ρ_i from r_2 to C_i in S_2 such that there exists $j \in [1, m]$ and X_j/T_j is on ρ_i if clause C_i is satisfied by $\mu(x_j)$ = true, and X_j/F_j is on ρ_i if clause C_i is satisfied by $\mu(x_j)$ = false; since ϕ is satisfied by μ, there must exist such a variable x_j for every C_i. It is easy to verify that σ is indeed an embedding from S_1 to S_2.

Conversely, suppose that there exists a valid embedding $\sigma = (\lambda, \mathsf{path})$ from S_1 to S_2. Observe that σ must have the following properties. (1) for each x_j there exists Y_j such that $\lambda(Y_j) = V_j$, where V_j is either T_j or F_j; and (2) for $i \in [1, n]$, $\lambda(C_i)$ is either C_i or V_j, where V_j is either T_j or F_j, such that $\lambda(Y_j) \neq \lambda(Y_k)$ and $\lambda(Y_j)$ is not an ancestor or self of $\lambda(C_i)$ for $k \neq j, i \neq j$. This is because, by the definitions of S_1, S_2, (1) $\lambda(C_i)$ must have $n + i$ descendants of type Z, like C_i in S_1; and (2) $\lambda(Y_j)$ must have $2n + j$ descendants of type W, and it may not be an ancestor of $\lambda(Y_s)$ or $\lambda(C_i)$, and vice versa. We define a truth assignment μ such that $\mu(x_s)$ is true if $\lambda(Y_s) = F_s$ and $\mu(x_s)$ is false if $\lambda(Y_s) = T_s$. As a result, for each clause C_i, $\lambda(C_i)$ is either Z_j or a child of Z_j, where Z_j is a truth value T_j or F_j; furthermore, if Z_j is true then $\lambda(Y_j) = F_j$, and if Z_j is false then $\lambda(Y_j) = T_j$. That is, C_i is satisfied by the truth assignment μ. Thus it is easy to verify that μ satisfies ϕ. □

In light of the intractability result we develop efficient and accurate heuristic algorithms for computing schema embedding candidates. Below we give an outline of these algorithms. The reader is encouraged to consult Bohannon et al. [2005] for the details of the algorithms and related experimental study.

5.1 Local Mappings

A *local mapping* is a schema mapping in which the domain is restricted to the schema elements appearing in a single production of the source schema S_1. We say that two local mappings *conflict* if they assign some source element A to different target elements in S_2. Based on this observation, we decompose the problem of computing schema embeddings to a) finding alternative local mappings (the Local-Embedding problem) and b) assembling non-conflicting local mappings into a valid schema embedding (the Assemble-Embedding problem). Unfortunately, each of these problems is independently NP-complete:

THEOREM 5.2. *The Local-Embedding problem is NP-complete for nonrecursive* DTDs.

PROOF. The NP-hardness can be verified by reduction from 3SAT. We omit the full proof for brevity, but note that it differs from the proof of Theorem 5.1 in that source elements are restricted to map to *exactly two* target elements, rather than allowing them to map to any target element as in Theorem 5.1. □

THEOREM 5.3. *The Assemble-Embedding problem is NP-complete for nonrecursive* DTDs.

PROOF. This proof is omitted for brevity. The lower bound is again verified by reduction from 3SAT. This proof is similar in spirit to the proof of 5.1, but shows that the more restrictive problem where local embeddings are fixed remains NP-hard. □

5.2 Computing Embeddings

In contrast to the previous results, it turns out that if λ is fixed (equivalently, att is *unambiguous*), then a schema embedding, if it exists, can be computed in low polynomial time. In other words, when the semantic correspondences between tags in the source schema and tags in the target are unique, it is easy to identify local embeddings, that is, embeddings in which the a single production in the source schema is considered; furthermore, from the local embeddings a global embedding can easily assembled. The central idea is that an unambiguous *local* embedding can be found by solving the *prefix-free path* problem, which we define as follows: Given a source node s and n target nodes $t_1 \ldots t_n$, find n paths $p_1 \ldots p_n$ such that a) each path originates at s, b) path p_i terminates at t_i, and c) no path p_i is a *prefix* of another path p_j. This problem can be solved in polynomial time on a DAG by a variant of depth-first search that, upon finding a path from s to some target t_i, returns from that search *without marking* t_i as done. The algorithm can be extended to handle cycles by first solving the problem on a DAG of the connected components in S_2. See Bohannon et al. [2005] for details.

This algorithm to find local embeddings given a fixed λ immediately yields an algorithm to find global embeddings given a fixed λ. Furthermore, the ability to find local embeddings can be generalized slightly to serve as part of a family of heuristic approaches to solving the more general Schema-Embedding problem. The idea is to randomly order the possible target matches for a source element in order to generate a candidate local mapping, and then heuristically attempt to assemble a global mapping. If the attempt fails, new random orderings can be used in an attempt to find additional local mappings.

Assembling Schema Embeddings. Given the ability to find valid local embeddings, a complete schema embedding can be computed if a *consistent* assembly of these local embeddings can be found, that is one in which all assembled local embeddings agree on the mapping of S_1 schema elements. We consider three heuristic approaches to Assemble-Embeddings. The first two attempt to incrementally assemble a full embedding by finding local embeddings for elements in S_1 in *some order*, either Random, or Quality-Ordered. In Random, as the name implies, the elements of S_1 are visited in random order. In Quality-Ordered, a quality metric based on att is used to assign a quality to local embeddings, and the elements of S_1 are re-ordered according to decreasing quality. The idea is to start with "better" mappings in an effort to find a good solution. The final approach reduces the Assemble-Embeddings problem to that of finding high-weight independent sets in a graph, and uses an existing heuristic solution to the latter problem [Busygin et al. 2002] to produce partial or complete embeddings.

Experimental Results. Experimental results of running the above heuristics on a variety of real-world schemas are reported in Bohannon et al. [2005].

The results show that the Random approach finds a high percentage of correct solutions over a wide range of att accuracies, and that running times are in the range of seconds or minutes. From this it seems reasonable that practical tools for computing schema embeddings can be included in the data integration process, and further, that it is practical to search for information-preserving embeddings without asking humans to hand-check the results of the schema-matching step first—that is, when there is ambiguity in the target element to which some of the source elements should be mapped.

6. RELATED WORK

Other than Bohannon et al. [2005] and Barbosa et al. [2005], we are aware of no previous work that has considered information preservation for XML DTD schema mappings. Schema embedding was introduced in a preliminary version of this article [Bohannon et al. 2005]. The recent work [Barbosa et al. 2005] studies invertible XML-to-relation mappings that guarantee the source XML document to remain valid in the presence of updates to the mapped relations. More specifically, it considers lossless and valid XML shredding into relations such that the original XML documents can be recovered from their relational storage (invertibility). Similar to Theorem 3.4, an undecidability result is given in Barbosa et al. [2005] for deciding whether XML shredding into relations are invertible. It characterizes DTDs in terms of datalog constraints and proposes a systematic approach to designing invertible XML shredding. The approach is based on a recursive rewriting system by means of (atomic) equivalence-preserving transformations, along the same lines as Abiteboul and Hull [1988] (see further on). This work differs from ours in the following. (a) It focuses on XML-to-relation mappings instead of XML-to-XML mappings. (b) It considers invertibility, but not XML query preservation. (c) Its rewriting system is quite different from the notion of schema embedding. Our notion of schema embedding extends graph similarity and allows multiple source DTD schemas to be mapped to a single structurally different target DTD. Furthermore, from a schema embedding an instance mapping can be *automatically* derived and it *guarantees* both invertibility and query preserving w.r.t. regular XPath queries. The ability of finding information-preserving XML-to-XML mappings is important for data integration, migration [Lenzerini 2002] and P2P systems [Fuxman et al. 2005; Kementsietsidis et al. 2003; Halevy et al. 2004], among other things.

Information preservation has been studied for nested relational and complex data models [Abiteboul and Hull 1988; Hull 1986; Miller et al. 1993, 1994]. Hull [1986] proposed several notions of dominance and studied their relationships in the relational model. In particular, it established the equivalence between query dominance (invertibility) and calculus dominance (the existence of an injective mapping defined in relational calculus). This is consistent with Theorem 3.2, which says that for any query language $\mathcal{L}$, if $\mathcal{L}$ is composable and can express the identity mapping, then the invertibility and query preservation w.r.t. $\mathcal{L}$ coincide; indeed, relational calculus is composable and can express the identity mapping. The notions of relative information capacity were revisited in [Miller et al. 1993, 1994], which showed, among other things, the invertibility in a complex data model is undecidable, similar to Theorem 3.4. The focus of [Abiteboul and Hull 1988; Miller et al. 1994] has mainly been on the information capacity of type constructs in complex data models that unlike DTDs, do not have recursive constructs. Information preserving schema transformations have also been studied there, based on local structural transformation rules that preserve or augment information capacity. Our study of information preservation is inspired by the prior work: our notions of invertibility and query preservation are mild extensions of calculus dominance and query dominance [Hull 1986]. We revise these notions and study their basic properties for XML DTD schemas and XML queries. Our focus is to develop the notion of DTD schema embedding that preserves information by ensuring *both* effective invertible mapping *and* efficient XML query translation, *without* employing a recursive rewrite system that repeatedly applies local type construct transformation rules.

A wide variety of techniques have been developed to solve different forms of schema matching or mapping for relational, ER and object-oriented models [Athitsos et al. 2005; Castano et al. 2001; Lakshmanan et al. 1996; Li and Clifton 2000; Palopoli et al. 1998]; see Rahm and Bernstein [2001] for a recent survey). While these are not focused on XML DTD schema mapping, some techniques, such as linguistic analyses and machine learning, are useful for finding name/label similarity, which our algorithms take as input.

Closer to XML schema mapping are [Doan et al. 2001; Madhavan et al. 2001; Melnik et al. 2002; Melnik et al. 2003; Miller et al. 2001; Milo and Zohar 1998]. LSD [Doan et al. 2001] proposes machine-learning techniques that make use of instance-level information to determine XML DTD tag matching, which can be used to compute similarity matrix att. Systems of [Madhavan et al. 2001; Melnik et al. 2002; Melnik et al. 2003] target a wide class of schemas and can be tailored to a variety of data models. The similarity flooding algorithm of [Melnik et al. 2002] provides a novel schema matching tool based on graph-similarity. Cupid [Madhavan et al. 2001] is a generic system that encompasses a variety of techniques such as linguistic analyses and context dependencies. Rondo [Melnik et al. 2003] proposes a powerful set of model mapping operators. For structure-level schema matching, these systems adopt graph similarity to map a single source schema to a target. TransScm [Milo and Zohar 1998] considers instance-level mappings based on schema matching, and uses a semi-automatic mechanism to match highly similar schemas. Clio [Miller et al. 2001] also focuses on deriving instance translation from schema matching. It specifies schema matches by inclusion dependencies from source to target, from which a schema mapping can be derived by means of chase techniques for reasoning about the dependencies. None of these considers information preservation.

Query preservation is related to query rewriting and query answering using views, which have been extensively studied for conjunctive and datalog queries in the relational model and for regular path queries on semistructured data ([Abiteboul and Duschka 1998; Calvanese et al. 2002; Levy et al. 1995]; see Halevy [2000] and Lenzerini [2002] for surveys). View-based query rewriting (resp. answering) mainly studies whether a given query on the source can be answered using materialized data from a set of views (lossless), by translating the query to an equivalent query (resp. in a particular language) on the views. In contrast, query preservation deals with the issue whether *all* queries in an (infinite) query language on an XML source can be rewritten to equivalent queries over XML target (view). Moreover, the focus of this work is to generate XML "views" that automatically preserves all the queries in an XML query language, rather than to determine the losslessness of views. Note that Theorem 3.2 establishes a connection between invertibility and query rewriting; *e.g.,* if the query language $\mathcal{L}$ includes the identity query id, then a view σ_d is invertible and σ_d^{-1} is in $\mathcal{L}$ if and only if id has a rewriting in $\mathcal{L}$ using σ_d.

There has also been recent work on data exchange based on schema mapping specified in terms of tuple generating dependencies (TGDs; see Kolaitis [2005] for a survey). This line of research considers mostly schema mapping between relational schemas, rather than XML schemas (except Arenas and Libkin [2005]). The focus is to decide, given a schema mapping specified by TGDs from source to target and TGDs on the target, as well as an instance of the source schema, whether or not there exists a solution, that is, an instance of the target schema that satisfies the TGDs? Furthermore, it investigates, in the presence of multiple solutions, the existence of a universal solution (the most general one) and the complexity to compute the universal solution. It aims to provide a guideline for materializing target instances in the data exchange context. The connection between universal solutions and certain query answers has also been explored. While certain query answers concern all possible instantiations of the target schema, our work focuses on the ability to restore the original source data and to answer all queries in a language on source data by queries in the same language on the target data. While Arenas and Libkin [2005] considers XML-to-XML mappings, its focus is on the consistency and certain query answers in connection with a *given schema mapping* defined via *TGD*s. This line of research differs from our work in that we aim at providing a *systematic method* for *developing* schema mappings that guarantee type safety, invertibility and query preservation. A schema embedding, even a local one, is defined in terms of edge-to-path mappings that are not expressible as TGDs. Invertible mappings and their practical need are studied in Fagin [2006]. It differs from this work in that it considers relation-to-relation mapping defined via TGDs; in contrast to XML mappings of schema embedding, a mapping defined via TGDs may map a source instance to multiple target instances; the focus of Fagin [2006] thus aims to find an appropriate notion of inverse for such mappings.

7. CONCLUSIONS

We have revised information-preservation criteria for XML mappings, and have established related separation, equivalence and complexity results. To cope with the difficulties of determining information preservation, we have introduced a novel notion of schema embedding for XML DTD schemas, from which an instance-level XML mapping is automatically derived and is guaranteed to be information preserving, type safe, and able to accommodate multiple source schemas. While we show that finding a schema embedding is NP-complete, we have provided heuristic algorithms to compute embeddings, which are efficient and accurate as verified by our experimental results [Bohannon et al. 2005].

This is a first step toward developing a practical tool for lossless XML data migration and integration. There is naturally much to be done. One open problem concerns how to extend the notion of schema embedding to accommodate XML Schema [Fallside (W3C) 2000], as commonly found in practice. XML Schema is more involved than DTDS: it supports not only a richer type system (e.g., with inheritance) but also integrity constraints. Indeed, it is already undecidable to determine whether or not a schema in XML Schema is consistent, i.e., whether there exists an instance conforming to the schema [Fan and Libkin 2002]. As a consequence, all the negative results of Section 3 remain intact for XML Schema, and it is nontrivial to develop a simple yet clean notion of schema embedding for XML Schema to capture both types and constraints. On the other hand, it is natural and not very difficult to extend schema embedding to specialized DTDS proposed in Papakonstantinou and Vianu [2000], an extension of DTDS.

Another extension is by allowing certain queries in XQuery in the path() function. The need for this is evident in practice. In data integration, for example, one often wants to group certain source data elements together and map the grouped data to a target element. Referring to schema S of Figure 1(c), for example, one may want to group *course*s under *taking* of *student*s in order to map instances of S to a *student* document. This calls for a nontrivial extension of the path() function to support upward traversal and also *group-by*. We are currently investigating this extension.

A third open problem concerns query preservation *w.r.t.* practical XQuery fragments. While the results of Section 3 carry over to any XQuery fragments that subsume regular XPath expressions, the notion of schema embedding has to be extended in order to ensure query preservation *w.r.t.* these richer query languages. In this context it is not clear whether the lower bound of Theorem 5.1 is tight; in other words, it remains unknown whether or not the problem of finding (extended) schema embedding is still in NP.

Finally, it is important to develop a notion of *partial* information preservation. In some applications one may find the notion of information preservation studied in this work too strong: one often wants to select part of the source data and require this part of data to be transformed to a target document without loss of information, instead of insisting on lossless mapping of the entire source data. We are currently revising the notions of invertibility and query preservation in response to this need, and are developing specification languages for users to identify the part of source data for which the information should be preserved.

ACKNOWLEDGMENTS

We thank Michael Flaster and P. P. S. Narayan for designing, implementing and experimenting with algorithms for finding schema embedding, which can be found in the conference version [Bohannon et al. 2005]. We also thank Renée Miller for discussions on information-preservation semantics. We thank referees for helpful suggestions on simplifying and improving the article.

REFERENCES

ABITEBOUL, S., BUNEMAN, P., AND SUCIU, D. 2000. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufman.

ABITEBOUL, S. AND DUSCHKA, O. M. 1998. Complexity of answering queries using materialized views. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*.

ABITEBOUL, S. AND HULL, R. 1988. Restructuring hierarchical database objects. *Theoretical Computer Science 62*, 1-2, 3–38.

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison-Wesley.

ALON, N., MILO, T., NEVEN, F., SUCIU, D., AND VIANU, V. 1995. XML with data values: Typechecking revisited. *J. Comput. Syst, Sci. 66*, 4, 688–727.

ARENAS, M. AND LIBKIN, L. 2005. XML data exchange: Consistency and query answering. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*.

ATHITSOS, V., HADJIELEFTHERIOU, M., KOLLIOS, G., AND SCLAROFF, S. 2005. Query-sensitive embeddings. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD)*.

BARBOSA, D., FREIRE, J., AND MENDELZON, A. 2005. Designing information-preserving mapping schemes for XML. In *Proceedings of International Conference on Very Large Databases (VLDB)*.

BENEDIKT, M., FAN, W., AND GEERTS, F. 2005. XPath satisfiability in the presence of DTDs. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*.

BOHANNON, P., FAN, W., FLASTER, M., AND NARAYAN, P. P. S. 2005. Information preserving XML schema embedding. In *Proceedings of International Conference on Very Large Databases (VLDB)*.

BUNEMAN, P., KHANNA, S., AND TAN, W. C. 2001. Why and where: A characterization of data provenance. In *Proceedings of International Conference on Database Theory (ICDT)*.

BUSYGIN, S., BUTENKO, S., AND PARDALOS, P. M. 2002. A heuristic for the maximum independent set problem based on optimization of a quadratic over a sphere. *J. Comb. Optim. 6*, 3, 287–297.

CALVANESE, D., GIACOMO, G. D., LENZERINI, M., AND VARDI, M. Y. 2002. Lossless regular views. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*.

CASTANO, S., ANTONELLIS, V. D., AND DI VIMERCATI, S. D. C. 2001. Global viewing of heterogeneous data sources. *IEEE Trans. Data Knowl. Engin. 13*, 2, 277–297.

CLARK, J. 1999. XSL Transformations (XSLT). W3C Recommendation. `http://www.w3.org/TR/xslt`.

CLARK, J. AND DEROSE, S. 1999. XML Path Language (XPath). W3C Working Draft.

DOAN, A., DOMINGOS, P., AND HALEVY, A. Y. 2001. Reconciling schemas of disparate data sources: A machine-learning approach. In *Proceedings of ACM SIGMOD Conference on Management of Data*.

EHRENFEUCHT, A. AND ZEIGER, H. P. 1976. Complexity measures for regular expressions. *J. Comput. Syst. Sci. 12*, 2, 134–146.

FAGIN, R. 2006. Inverting schema mappings. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*.

FALLSIDE, D. C., Ed. 2000. *XML Schema Part 0: Primer*. World Wide Web Consortium (W3C). `http://www.w3.org/TR/xmlschema-0/` .

FAN, W., GEERTS, F., JIA, X., AND KEMENTSIETSIDIS, A. 2007. Rewriting regular xpath queries on XML views. In *IEEE International Conference on Data Engineering (ICDE)*.

FAN, W. AND LIBKIN, L. 2002. On XML integrity constraints in the presence of DTDs. *J. ACM 49*, 3, 368–406.

FUXMAN, A., KOLAITIS, P., MILLER, R., AND TAN, W. 2005. Peer data exchange. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*.

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

HALEVY, A. Y. 2000. Theory of answering queries using views. *SIGMOD Record 29*, 4, 40–47.

HALEVY, A. Y., IVES, Z. G., MADHAVAN, J., MORK, P., SUCIU, D., AND TATARINOV, I. 2004. The Piazza peer data management system. *IEEE Trans. Data Knowl. Engin. 16*, 7, 787–798.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Languages and Computation*. Addision Wesley.

HULL, R. 1986. Relative information capacity of simple relational database schemata. *SIAM J. Comput. 15*, 3, 239–265.

KEMENTSIETSIDIS, A., ARENAS, M., AND MILLER, R. 2003. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD)*.

KOLAITIS, P. G. 2005. Schema mappings, data exchange, and metadata management. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*.

LAKSHMANAN, L., SADRI, F., AND SUBRAMANIAN, I. N. 1996. SchemaSQL—a language for interoperability in relational multi-database systems. In *Proceedings of International Conference on Very Large Databases (VLDB)*.

LENZERINI, M. 2002. Data integration: A theoretical perspective. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*.

LEVY, A. Y., MENDELZON, A. O., SAGIV, Y., AND SRIVASTAVA, D. 1995. Answering queries using views. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*.

LI, W.-S. AND CLIFTON, C. 2000. SemInt: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data Knowl. Engin. 33*, 1, 49–84.

MADHAVAN, J., BERNSTEIN, P. A., AND RAHM, E. 2001. Generic schema matching with Cupid. In *Proceedings of International Conference on Very Large Databases (VLDB)*.

MARX, M. 2004. XPath with conditional axis relations. In *Proceedings of the International Conference on Extending Database Technology*.

MELNIK, S., GARCIA-MOLINA, H., AND RAHM, E. 2002. Similarity flooding: A versatile graph matching algorithm. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*.

MELNIK, S., RAHM, E., AND BERNSTEIN, P. A. 2003. Rondo: A programming platform for generic model management. In *Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD)*.

MILLER, R. J., HERNÁNDEZ, M. A., HAAS, L. M., YAN, L.-L., HO, C. T. H., FAGIN, R., AND POPA, L. 2001. The Clio project: Managing heterogeneity. *SIGMOD Record 30*, 1, 78–83.

MILLER, R. J., IOANNIDIS, Y. E., AND RAMAKRISHNAN, R. 1993. The use of information capacity in schema integration and translation. In *Proceedings of International Conference on Very Large Databases (VLDB)*.

MILLER, R. J., IOANNIDIS, Y. E., AND RAMAKRISHNAN, R. 1994. Schema equivalence in heterogeneous systems: Bridging theory and practice. *Inform. Syst. 19*, 1, 3–31.

MILO, T. AND ZOHAR, S. 1998. Using schema matching to simplify heterogeneous data translation. In *Proceedings of International Conference on Very Large Databases (VLDB)*.

PALOPOLI, L., SACCA, D., AND URSINO, D. 1998. Semi-automatic semantic discovery of properties from database schemas. In *Proceedings International Database Engineering & Applications Symposium (IDEAS)*.

PAPAKONSTANTINOU, Y. AND VIANU, V. 2000. Type inference for views of semistructured data. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*.

RAHM, E. AND BERNSTEIN, P. A. 2001. A survey of approaches to automatic schema matching. *VLDB J. 10*, 4, 334–350.

SIMÉON, J. AND FERNANDEZ, M. Galax. `http://db.bell-labs.com/galax`.

TARJAN, R. E. 1981. Fast algorithms for solving path problems. *J. ACM 28*, 3, 594–614.

WADLER, P. 2000. A formal semantics for patterns in XSL. Tech. rep., Bell Labs.

XERCES AND XALAN. http://xml.apache.org.

YU, S. 1996. Regular languages. In G. Rosenberg and A. Salomaa, Eds. *Handbook of Formal Languages*, Vol. 1. Springer.