Information Preserving XML Schema Embedding

Philip Bohannon **Wenfei Fan** **Michael Flaster** **P. P. S. Narayan**
Bell Laboratories
{bohannon,wenfei,mflaster,ppsn}@research.bell-labs.com

Abstract

A fundamental concern of information 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 document(s) 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 defined 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 provide efficient heuristic algorithms to find candidate embeddings, along with experimental results to evaluate and compare the algorithms. These yield the first systematic and effective approach to finding information preserving XML mappings.

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. Of course, one can define XML mappings in a query language such as XQuery or XSLT, but such queries may be large and complex, and in practice it is desirable that XML mappings (1) guarantee type-safety and (2) preserve information.

It is clearly desirable that the document produced by an XML mapping conforms to a target schema, guaranteeing *type safety*. But this may be difficult to check for mappings defined in XQuery or XSLT [5]. Further, since in many applications one does not want to lose the original information of the source data, a mapping should also preserve information. Criteria for *information preservation* include: (1) *invertibility* [16]: can one recover the source document from the target? and (2) *query preservation*: for a particular XML query language, can all queries on source documents in that language be answered on target documents? We now illustrate these concepts with an example.

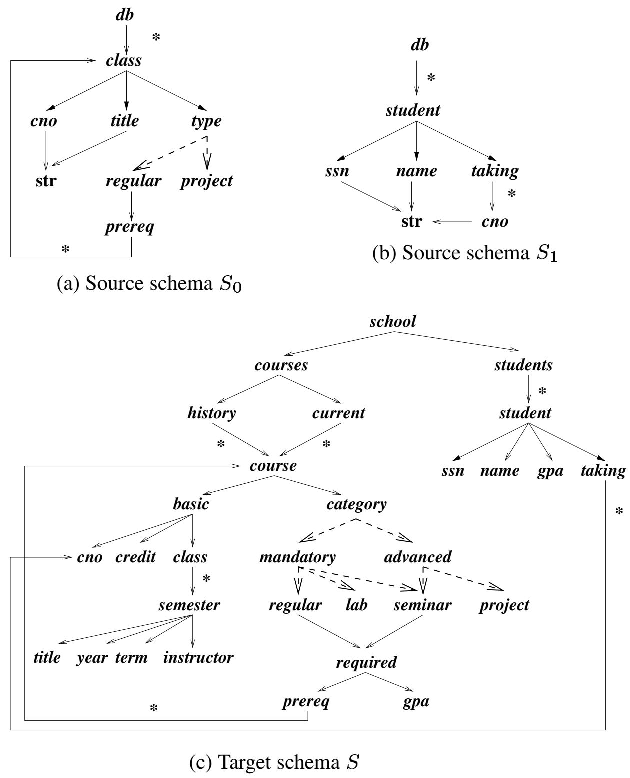
Example 1.1: Consider two source DTDs S_0, S_1 and a target DTD S represented as graphs in Fig. 1 (we omit the str child under *cno, credit, title, year, term, instructor, gpa* in Fig. 1(c)). A document of S_0 contains information of *class*es currently being taught at a school, and a document of S_1 consists of *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 represent different constructs of a DTD, namely, *solid edges* for a concatenation type (a unique occurrence of each children), *dashed edges* for disjunction (one and only one children), and *star edges* (edge labeled '$*$') for Kleene star (zero or more children). □

In this example, 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 posed on *class* and *student* documents using the *school* document. Two natural questions are: (a) can one determine whether an XML mapping is information preserving? (b) is there an efficient method to find information-preserving XML mappings?

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 typical in data integration, where the target DTD needs to accommodate data from *multiple sources* and thus cannot be similar to any of the sources; see, e.g., the *class*, *student* DTDs and the *school* DTD in Fig. 1.

Background. While information preservation has been studied for traditional database transformations [3, 16, 28, 29], 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 [13, 23, 25, 26, 27, 30]; 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 bisimulation (see,, e.g., [1]) to match structures, which is incapable of mapping DTDs with different structures such as those shown in Fig. 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.

Contribution. To this end we study information preserving XML mappings, and make the following contributions.



(a) Source schema S_0

(b) Source schema S_1

(c) Target schema S

Figure 1: Example: source and target schemas

First, as criteria for information preservation we revisit the notions of invertibility and query preservation [3, 16, 28, 29] for XML mappings (Section 2). While the two notions coincide for relational mappings w.r.t. relational calculus [16], 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 simple fragment of XQuery (or XSLT) is information preserving (Section 3).

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 Fig 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 [24]), and ensure type safety. As with schema-mapping techniques for other data models, by automatically producing this mapping the user is saved from writing and type-checking a complex mapping query. Moreover, we show that the *inverse* and *query rewriting functions* for the mapping are efficient (Section 4).

Third, 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 algorithms for finding embeddings are necessarily heuristic. A building block of our algorithms is an efficient algorithm to find a *local embedding* for individual productions in the source schema. Based on this, we develop three heuristic algorithms to compute embeddings. The first two algorithms repeatedly attempt to assemble local embeddings into a schema embedding (using a random or quality-specific order of the local embeddings, respectively), and when conflicts arise, attempt to generate new, non-conflicting local embeddings. The third algorithm generates a candidate pool of local embeddings, and then uses a heuristic solution to Maximum-Independent-Set to assemble a valid schema embedding (Section 5).

Finally, 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 (Section 6), and support the idea that automatically computed schema embeddings are a promising tool for computing information preserving XML mappings. We discuss related work in Section 7. Proofs of theorems are in [7].

To the best of our knowledge, this work is the first to study information preservation in the XML context, and it yields a systematic and effective approach to defining and finding information preserving XML mappings.

2 DTDs, XPath, Information Preservation

In this section we review DTDs and (regular) XPath, and revisit information preservation [16, 29] for XML.

2.1 XPath and Regular XPath

We consider a class of *regular* XPath queries proposed and studied in [24], denoted by $\mathcal{X}_R$ and defined as follows:

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

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; 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.

An XPath 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, and its result is the set of nodes (ids) of T reachable via p from v, denoted by $v[\![p]\!]$.

2.2 DTDs

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

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

where str denotes PCDATA, ϵ is the empty word, B is a type in *Ele* (referred to as a *child* of A), and '+', ',' and '$*$' denote *disjunction* (with $n > 1$), *concatenation* and the *Kleene star*, respectively. We refer to $A \to P(A)$ as the *production* of A. Note that this form of DTDs does not lose generality since any DTDs 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 [6].

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 $A \to P(A)$ of A. There are three different types of edges indicating different 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; 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 $B_1 + \ldots + B_n$ 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). If $P(A)$ is B^*, then there is a *solid* edge with *a '$*$' label* from the A node to the B node. Note that a DTD is *recursive* if its graph is *cyclic*. 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.

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

An XML *instance* of a DTD S is a node-labeled tree that conforms to S. We use $\mathcal{I}(S)$ to denote the set of all instances of S. A DTD S is *consistent* if it has no useless element types, i.e., each type of S has an instance. In the sequel we consider consistent DTDs only.

2.3 Invertibility and Query Preservation

For XML DTDs S_1, S_2, a 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 the notation $f(T)$ 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 $F : \mathcal{L} \to \mathcal{L}$ such that for any XML query $Q \in \mathcal{L}$ and any $T \in \mathcal{I}(S_1)$, $Q(T) = F(Q)(\sigma_d(T))$, i.e., $Q = F(Q) \circ \sigma_d$.

In a nutshell, invertibility is the ability that the original source XML document can be recovered from the target document; query preservation w.r.t. $\mathcal{L}$ indicates whether or not *all* queries of $\mathcal{L}$ on any source T of S_1 can be effectively answered over $\sigma_d(T)$, i.e., 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 [16] for relational mappings and later studied in [3, 28, 29]. 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. $\mathcal{L}$.

3 Information Preservation

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

Invertibility and Query Preservation. It was shown [16] 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 the last section.

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.* □

We identify sufficient conditions for the two to coincide:

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

- *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.*
- *Suppose $\mathcal{L}$ is composable, i.e., for any Q_1, Q_2 in $\mathcal{L}$, $Q_2 \circ Q_1$ is in $\mathcal{L}$; if σ_d is invertible and σ_d^{-1} is expressible in $\mathcal{L}$, then σ_d is query preserving w.r.t. $\mathcal{L}$.* □

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.

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 a mapping σ_d that is invertible but is not query preserving w.r.t. $\mathcal{X}_R$.* □

Complexity. It is common to find XML mappings defined in XQuery or XSLT. A natural yet important question is to decide whether or not an XML mapping 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, or relational algebra), e.g., XQuery, XSLT, even when $\mathcal{L}$ consists of projection queries only. Thus it is beyond reach to answer these questions for XQuery or XSLT mappings.

Theorem 3.4: *It is undecidable to determine, given an* XML *mapping σ_d defined in any language subsuming FO,*

1. *whether or not σ_d is invertible;*

2. *whether or not σ_d is query preserving w.r.t. projection queries.* □

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. This motivates us to look for a class of XML mappings that are *guaranteed* to be information preserving.

We approach this problem by specifying XML mappings at the schema level embeddings, and providing an automated derivation of instance-level mappings from these embeddings. Our notion of *schema embeddings* is novel, and extends the conventional notion of graph similarity by allowing edges in a source DTD schema to be mapped to a path in a target DTD with a "larger information capacity". For example, a STAR edge can only be mapped to a path with at least one STAR edge.

In this section we define XML schema embeddings, present an algorithm for deriving an instance-level mapping from a schema embedding, and verify that the resulting mappings ensure 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 $\mathsf{path}(A, B)$ from $\lambda(A)$ to $\lambda(B)$ in S_2, such that the S_2 paths mapped from sibling edges in S_1 are sufficiently distinct to allow information to be preserved. To define λ and path we first introduce a few notations.

$\mathcal{X}_R$ 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 is a path in S and it carries 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 Fig. 1(c), for example, *basic/class/semester/title* is an AND path as well as a STAR path, and *mandatory/regular* is an OR path.

Name 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 algorithm, e.g., [13, 22].

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 $A \in E_1$, $\mathsf{att}(A, \lambda(A)) > 0$.

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.

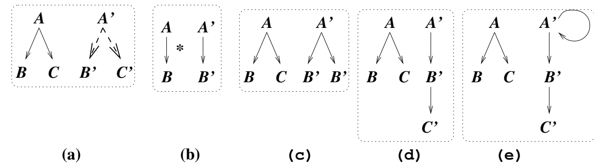


Figure 2: Path mappings for DTDs

For a particular element type A in E_1, we say that σ is *valid* for A if the following conditions hold, 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 $j \neq i$;
- if $P_1(A) = B_1 + \ldots + B_l$, 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 $j \neq i$;
- 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 validity requires a *path type* condition and a *prefix-free* condition, which, as will be seen shortly, are important for deriving the instance-level mapping from σ.

Example 4.1: Consider pairs of source (on the left) and target (on the right) DTDs depicted in Fig. 2, for which a type mapping λ is defined as $\lambda(X) = X'$ for X in $\{A, B, C\}$, except in Fig. 2(c) where both $\lambda(C) = B'$ and $\lambda(B) = B'$. Observe the following. For Fig. 2(a), there is no valid path embedding from the source DTD to the target; intuitively, B and C must coexist in a source document while only one of B' and C' exists in the target. Similarly for Fig. 2(b), where the source cannot be mapped to the target since there are possibly multiple B elements in the source, which cannot be accommodated by the target. For Fig. 2(c), a valid embedding is $\mathsf{path}(A, B) = B'[position() = 1]$ and $\mathsf{path}(A, C) = B'[position() = 2]$. For Fig. 2(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 Fig. 2(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 *valid* 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.

Example 4.2: Assume a similarity matrix att such that $\mathsf{att}(A, A') = 1$ for all A in the DTD S_0 of Fig. 1(a) and A' in S of Fig. 1(c). The source DTD S_0 can be embedded in the target S via $\sigma_1 = (\lambda_1, \mathsf{path}_1)$ 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/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	
$\mathsf{path}_1(A, \mathsf{str})$	= *text()*	/* A for cno, title */

Note that $\mathsf{path}_1(A, B)$ is a path in S denoting how to reach $\lambda_1(B)$ from $\lambda_1(A)$, *i.e.*, the path is *relative to* $\lambda_1(A)$. 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. Also observe that 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.

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. □

Embedding Quality. There are many possible metrics. In this paper we consider only a simple one: the quality of a schema embedding $\sigma = (\lambda, \mathsf{path})$ w.r.t. att is the sum of $\mathsf{att}(A, \lambda(A))$ 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 $\mathsf{qual}(\sigma, \mathsf{att})$.

4.2 Instance Level Mapping

For a valid schema embedding $\sigma = (\lambda, \mathsf{path})$ from S_1 to S_2, we give its semantics by defining an instance-level mapping $\sigma_d : \mathcal{I}(S_1) \rightarrow \mathcal{I}(S_2)$, 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. In a nutshell, σ_d constructs T_2 top down starting from the root r_2 of T_2, mapped from the root r_1 of T_1 (recall $\lambda(r_1) = r_2$). Inductively, for each $\lambda(A)$ element u in T_2 that is mapped from an A element v in T, σ_d generates a distinct $\lambda(B)$ node u' in T_2 for each distinct B child v' of v in T_1, such that u' is reached from u via $\mathsf{path}(A, B)$ in T_2, i.e., u' is uniquely identified by the $\mathcal{X}_R$ path from u. More specifically, the construction is based on the production $A \rightarrow P_1(A)$ in S_1 as follows.

(1) $P_A(A)$ is $B_1, \ldots, B_n$. For each child v_i of v, σ_d creates a node u_i bearing the same id as v_i. These nodes are added to T_2 as follows. For each $i \in [1, n]$, u_i is added to T_2 by creating $\mathsf{path}(A, B_i)$ emanating from u to u_i, such that the path shares any prefix already in T_2 which were created for, e.g., $\mathsf{path}(A, B_j)$ for $j < i$.

(2) $P_1(A)$ is $B_1 + \ldots + B_n$. Here v in T_1 must have a unique child v_i. For v_i, σ_d creates a node u_i bearing the same id as v_i, and adds u_i to T_2 via $\mathsf{path}(A, B_i)$ as above.

(3) $P_1(A)$ is B^*. By the definition of valid path function, $\mathsf{path}(A, B)$ is of the form $\mathsf{path}(A, A_1)/B_1/\mathsf{path}(B_1, B)$, where A_1 is the first type defined in terms of Kleene star in P_2, i.e., $P_2(A_1) = B_1^*$. Let $v_1, \ldots, v_k$ be all the children of v. Then σ_d creates $u_1, \ldots, u_k$ bearing the same id's as $v_1, \ldots, v_k$, and adds these nodes to T_2 as follows. It first generates a single $\mathsf{path}(A, A_1)$ from u to an A' node u' if it does not already exist in T_2, and for each $i \in [1, k]$, it creates a distinct i-th B_1 child if it is not already in T_2. From the i-th B_i node it generates $\mathsf{path}(B_1, B)$ leading to u_i, in the same way as in (1) above. Note that the order of the children of v is preserved by σ_d.

(4) $P_1(A)$ is str. 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.

We repeat the process until all nodes in T_1 are mapped to nodes in T_2. We finally complete $\sigma_d(T)$ by adding *necessary* default elements such that $\sigma_d(T)$ conforms to S_2. Recall from Section 2 that we can assume w.l.o.g. consistent DTDs. Thus for each element type A in S_2, we can pick a fixed instance I_A of A and use it as A's *default* element.

Example 4.3: Consider the XML mapping σ_d of the embedding defined in Example 4.2. Given an instance T_1 of S_0 of Fig. 1(a), σ_d generates a tree T_2 of S of Fig. 1(c) as follows: σ_d first creates the root *school* of T_2, bearing the node id of the root *db* of T_1. Then, σ_d creates a single *courses* child x of *school*, a single *current* child y of x, and for each *class* child c of *db*, σ_d creates a distinct *course* child z of y bearing the id of c, such that the *course* children of y are in the same order as the *class* children of *db*. It then maps the *cno, title, type* children of c to *cno, title, category* descendants of z in T_2, based on path_1. In particular, to map *title* in S_0, it creates a single *class* child x_c of the *basic* element, a *single semester* child x_s under x_c (although *class* is defined with a Kleene star), and then a *title* child under x_s. For the *category* element w mapped from the *type* child t of c, σ_d creates a distinct path *advanced/project* under w if t has a *project* child, or a *mandatory/regular* path otherwise, but not both. The process proceeds until all nodes in T_1 are mapped to T_2. Finally, default elements of *history*, *credit*, *year*, *term*, *instructor* and *gpa* are added to T_2 such that T_2 conforms to S. At the last stage, no children of disjunctive types *category*, *mandatory* or *advanced* are added, and no children are created under *history*. That is, default elements are added only *when necessary*. □

Note that $\sigma_d(T_1)$ does not have to be a new document of S constructed starting from scratch. Under certain conditions a mild variation of σ_d allows us to expand an existing instance T_2 of S to accommodate nodes of T_1, by modifying default elements in T_2 and introducing new elements.

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 that 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, project)* 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 below shows that the *prefix-free* condition in the definition of valid path functions ensures that conflicting paths do not exist, and thus prevents violation caused by conflicting paths.

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. Indeed, σ determines an injective *path-mapping* function δ such that, for each $\mathcal{X}_R$ path $\rho = A_1[q_1]/\ldots/A_k[q_k]$ in S_1 from r_1, $\delta(\rho)$ is $\mathsf{path}(r_1, A_1)[q_1]/\ldots/\mathsf{path}(A_{k-1}, A_k)[q_k]$, 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. Since each node in T_1 is uniquely determined by an $\mathcal{X}_R$ path from the root, it follows that σ_d is injective.

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

4.3 Properties of Schema Embeddings

We have shown that the XML mapping σ_d of a valid schema embedding σ is guaranteed to type check. We next show that σ_d and σ also have all the other desired properties.

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

Theorem 4.2: *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 F that given any $\mathcal{X}_R$ query Q over S_1, computes an $\mathcal{X}_R$ query $F(Q)$ equivalent w.r.t. σ_d over S_2 in $O(|Q|\ |\sigma|\ |S_1|)$ time.* □

Example 4.4: The $\mathcal{X}_R$ query Q below, over S_0 of Fig. 1(a), is to find all the classes that are (direct or indirect) prerequisites of CS331. It is translated to an $\mathcal{X}_R$ query Q' over S of Fig. 1(c), which is equivalent w.r.t. the mapping σ_d given in Example 4.3, 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.

Q: *class[cno/text()='CS331']/(type/regular/prereq/class)*.*

Q': *courses/current/course[cno/text()='CS331']/(category/mandatory/regular/required/prereq/course)*.* □

In contrast, the notion of graph similarity ensures neither invertibility nor query preservation w.r.t. $\mathcal{X}_R$. As a simple example, the source and target schemas in Fig. 2(a) are bisimilar (if the conventional definition of graph similarity is not extended to accommodate the cardinality constraints of different DTD constructs). However, there exists no instance-level mapping from the source to the target, not to mention inverse mappings and query translation.

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 4.5: The embedding $\sigma_2 = (\lambda_2, \mathsf{path}_2)$ below maps S_1 of Fig. 1(b) to the target DTD S of Fig. 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, one can define schema embeddings $\sigma_i : S_i \to S$ to simultaneously map S_i to S. Their XML mappings $\sigma_d^1, \ldots, \sigma_d^n$ are invertible and query preserving w.r.t. $\mathcal{X}_R$ as long as δ_i, δ_j are *pairwise disjoint*, where δ_i is the path mapping function derived from σ_i to map $\mathcal{X}_R$ paths from root in S_i to $\mathcal{X}_R$ paths from root in S. The instance-level XML mapping σ_d is a composition of individual $\sigma_d^1, \ldots, \sigma_d^n$, such that σ_d^i increments the document constructed by σ_d^j's for $j < i$ by modifying default elements or introducing new elements, as described earlier.

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 4.3: *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, $|\mathsf{path}(A, B)| \leq (k+1)\ |E_2|$, where $S_2 = (E_2, P_2, r_2)$, and k is the size of the production $P_2(A)$.* □

5 Computing Schema Embeddings

In this section we address the computation of XML schema embeddings as defined by the following problem, stated in terms of two XML DTD schemas $S_1 = (E_1, P_1, r_1)$ and $S_2 = (E_2, P_2, r_2)$, and a similarity matrix att:

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

In practice, a reasonable goal is to find an embedding $\sigma : S_1 \to S_2$ with as high a value for $\mathsf{qual}(\sigma, \mathsf{att})$ as possible. The ability to efficiently find good solutions to this problem will lead to an automated tool that, given two DTD schemas, compute candidate embeddings to recommend to users.

However desirable, this problem is NP-complete. Its intractability is rather robust: 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* DTD*s.* □

In light of the intractable results we develop two efficient yet accurate heuristic algorithms for computing schema embedding candidates in the rest of the section.

Notations. Recall that a schema embedding is a path mapping σ that is valid for each element type A in S_1. Since the validity conditions for A involve only A's immediate children, it is useful to talk about mappings local to A. A *local*

Algorithm findPathsDAG (G, s, L_{tar})

Input: Directed Acyclic Graph G, source node s,
a bag of target nodes $L_{\mathsf{tar}} = \{\!|t_1, \ldots, t_k|\!\}$.
Output: Paths $\rho_1, \ldots, \rho_k$ satisfying the prefix-free condition.

path ρ := <empty>;
$\mathcal{P} = \emptyset$;
marked (n) := false for all n ;
traverse $(G, s, \rho, L_{\mathsf{tar}}, \mathcal{P})$;
if L_{tar} is nonempty
  return $\emptyset$;
else return $\mathcal{P}$;

Figure 3: Algorithm findPathsDAG

mapping for A is simply a *partial* path mapping $(\lambda_0, \mathsf{path}_0)$ such that (a) λ_0 and path_0 are defined exactly on all the element types appearing in A's production $A \to P_1(A)$, including A itself; and (b) it is *valid*, *i.e.,* it satisfies the path type and prefix-free conditions given in the last section.

Consider two partial mappings, $\sigma_0 = (\lambda_0, \mathsf{path}_0)$ and $\sigma_1 = (\lambda_1, \mathsf{path}_1)$. We say that λ_0 and λ_1 *conflict on* A if both $\lambda_0(A)$ and $\lambda_1(A)$ are defined, but $\lambda_0(A) \neq \lambda_1(A)$, and similarly for path_0 and path_1. We say σ_0 and σ_1 are *consistent* if they do not conflict, either on λ or path. The *union of consistent partial mappings*, denoted by $\sigma_0 \oplus \sigma_1$, is a partial embedding defined to be $(\lambda_1 \oplus \lambda_2,\ \mathsf{path}_1 \oplus \mathsf{path}_2)$, where

$$\lambda_1(A) \oplus \lambda_2(A) = \begin{cases} \lambda_1(A) & \text{if } \lambda_2(A) \text{ is } \perp \text{ (undefined)} \\ \lambda_2(A) & \text{if } \lambda_1(A) \text{ is } \perp \\ \lambda_1(A) & \text{otherwise} \end{cases}$$

similarly for $\mathsf{path}_1(A, B) \oplus \mathsf{path}_2(A, B)$.

Outline. In the rest of the section we first present a technique for finding local embeddings, already a nontrivial yet interesting problem. Making use of this algorithm, we then provide three heuristics for finding embedding candidates. The first two are based on randomized programming and the last is by reduction from our problem to the Max-Weight-Independent-Set problem for which a well-developed heuristic tool [9] is available.

5.1 Finding Valid Local Mappings

We start by giving an algorithm to find a local embedding $\sigma_0 = (\lambda_0, \mathsf{path}_0)$ when the partial type mapping λ_0 is fixed, as this is a key building block of our schema-embedding algorithms. We then extend the algorithm to handle the general case when λ_0 is not given. To simplify the presentation we focus on nonrecursive DTDs, *i.e.,* DTDs with a *directed acyclic graph (DAG)* structure, but we show that our technique also works on recursive (cyclic) DTDs.

Finding Valid Paths. Let $A \in E_1$ be a source element type with production $A \to P_1(A)$, in which the element types appearing in $P_1(A)$ are $B_1, \ldots, B_k$. Assume that the type mapping λ_0 is already given as a partial function from E_1 to E_2 that is defined on $B_1, \ldots, B_k$ and A. The Valid-Paths problem is to find paths $\mathsf{path}_0(A, B_1), \ldots, \mathsf{path}_0(A, B_k)$ such that $(\lambda_0, \mathsf{path}_0)$ is a valid local mapping for A.

The validity conditions stated for embeddings in Section 4.1 require that (a) target paths for each edge are of the appropriate *type* (AND, OR, or STAR path), and (b) that the target path for an edge is *not a prefix* of a *sibling*'s target path. We abstract the second condition as a directed-graph problem: Given a directed graph $G = (V, E)$, a source vertex s and a *bag* of target vertices $L_{\mathsf{tar}} = \{\!|t_1 \ldots t_k|\!\}$, find paths $\rho_1, \ldots, \rho_k$ such that no path is equal to or is the prefix of another. That is, for all $i \neq j$, $\rho_j \neq \rho_i/\rho_{ij}$ for any ρ_{ij} including the empty path. In contrast to most sub-problems of Schema-Embedding, this can be solved in *polynomial time*. We introduce our solution by giving an algorithm that works only on a DAG and discuss extending it to handle cycles below.

Algorithm traverse $(G, n, \rho, L_{\mathsf{tar}}, \mathcal{P})$

Input: Directed Acyclic Graph G, node n,
a bag of target nodes $L_{\mathsf{tar}} = \{\!|t_1, \ldots, t_k|\!\}$,
ρ, the current path to the root
and $\mathcal{P}$ the output set of prefix-free paths
Global variables: marked: maps nodes to {true,false }
Output: a list of paths

1. if (marked (n)) return false;
2. if ($n \in L_{\mathsf{tar}}$)
3.   remove n from L_{tar};
4.   add ρ to $\mathcal{P}$
5.   return true;
6. else ret = false;
7.   for each edge $e = (n, m)$ outgoing from n
8.   append e to ρ;
9.   ret := ret or traverse $(G, m, \rho, L_{\mathsf{tar}}, \mathcal{P})$;
10.   remove e from ρ;
11.   if (not ret) marked (n):=true;
12. return ret;

Figure 4: Algorithm traverse

We present our algorithm, findPathsDAG, in Fig. 3, for finding prefix-free paths in a DAG. The algorithm depends on the recursive procedure traverse, shown in Fig. 4. The intuition of this algorithm is to modify a simple (but exponential) algorithm to recursively enumerate all paths in a DAG in such a way that prefix-free paths are found, but excessive running time is avoided. In a nutshell, traverse conducts a depth-first-search on the input graph G, enumerating paths from the source node s to target nodes in L_{tar}, and identifies prefix-free ones. It uses a (global) boolean array marked (n) to keep track of whether the subgraph rooted at a node n has been searched and yielded no matches for nodes in L_{tar}, and if so, it does not re-enter the subgraph. A (local) variable *ret* is used to indicate whether the search of a subgraph finds any matches to nodes in L_{tar}.

To see that traverse is correct, consider removing line 5 in which the algorithm returns early, and line 11 in which nodes are marked to avoid revisiting them. It is clear that the resulting algorithm considers every possible path leading to nodes in L_{tar}, and assigns one path to each $n \in L_{\mathsf{tar}}$, but it does not avoid assigning one node the prefix of another path. However, the prefix-free condition is assured by the return at line 5 *without affecting correctness*, since a suffix of the path assigned to n could only be generated by continuing the recursion from this node. Thus it remains to argue that the algorithm is still correct if line 11 is in place.

The intuition of line 11 is simple: if no new target nodes were found in the subtree of a node when it was explored by the recursive calls of lines 7-10, then the current node will not be on any path to any n' remaining in L_{tar}.

Example 5.1: Consider the schema embedding problem shown in Figure 1. Say that att (*regular*, *seminar*), and att (*project*, *advanced*) in S_0 are 0.75. This means that the bag of possible target matchings for source tags $\{$*regular*, *project*$\}$ in S_0 can be $\{|$*seminar*, *advanced*$|\}$ from S. We then invoke traverse with S, *category*, ρ (which is empty), and L_{tar} as $\{|$*seminar*, *advanced*$|\}$. The first call to traverse would result in all edges from *category* to be recursed. Say, our algorithm first picks the edge to *advanced*. Line 2 of traverse would check *advanced* to be in L_{tar} and add the path to *advanced* into $\mathcal{P}$. It would then return back from the recursion and try the other edges from *category* in lines 7 though 10. This would result in a prefix-free path *mandatory/seminar* which would then be added to $\mathcal{P}$. ☐

To analyze the performance of findPathsDAG, consider traverse as a sequence of forward and backward traversals of edges in the graph. A forward traversal occurs at line 9 and a backward traversal at lines 1, 5 and 12. Clearly, the number of forward traversals and backward traversals in a run are the same. Further, observe that one returns from an un-marked node at line 5 only on the path *back* from some node newly removed from L_{tar}. Thus, there can be at most $|L_{\mathsf{tar}}|\ |V|$ such backward steps, and at most $|E|$ other backward steps (which mark the child of the edge traversed). Since G is a DAG, the algorithm is in $O(|L_{\mathsf{tar}}|\ |V|)$ time.

To use findPathsDAG in our algorithms for schema embedding, we must further ensure that the paths returned match the types needed for $n \in L_{\mathsf{tar}}$. That is easy to accomplish, as the type of a path can be maintained incrementally as it is lengthened and shortended (by storing counts of nodes of each type), and be checked at line 2.

Schema Embeddings with a Given λ. This algorithm can be used to directly find a schema embedding $\sigma = (\lambda, \mathsf{path})$ from S_1 to S_2 when the type mapping λ is a given total function from E_1 to E_2. As remarked earlier, the validity conditions for any A in E_1 involve only A's children; thus to find path we only need to find valid paths for each A in E_1 and take the union of these valid local embeddings. This yields an $O(|S_1|\ |S_2|)$ algorithm to find embeddings in this special setting, which is not so uncommon since one may know in advance which target type a source type should map to, based on, *e.g.,* machine-learning techniques [13].

Handling Multiple Targets. However, to find valid local mappings when λ is not given, we must consider that there are *multiple* possible target nodes for each source node. The general Local-Embedding problem is to find a local embedding $(\lambda_0, \mathsf{path}_0)$ when λ_0 may not be fixed. This problem is no longer tractable as shown below.

Theorem 5.2: *The Local-Embedding problem is NP-hard for nonrecursive* DTD*s.* ☐

One heuristic approach to finding local embeddings is to extend findPathsDAG as follows. We compute the set of all pairings of source nodes A and possible matches for A from att and pass it as L_{tar}. We also modify line 3 of traverse to (a) pick an arbitrary pair with the current node as the target from L_{tar} at line 2 and (b) remove all pairs associated with source node A from L_{tar} at line 3. While this may work, it is essentially a greedy algorithm and may not find a solution if one exists. To compensate for this, we actually use a randomized variant findPathsRand (not shown) which (a) picks a random source node associated with n at line 2 of traverse, and (b) tries outgoing edges from n at line 7 in random order. The ability of findPathsRand to find embeddings varies with the size of L_{tar}, and will be investigated in Section 6.

Handling Cycles. Of course, schemas are frequently cyclic (recursive), and the algorithms as presented so far only handle DAGs. In fact, handling cycles generally is somewhat more complicated, but not hard – it is easy to see that an arbitrary number of paths can be generated by repeated loops around some cycle on the path to a target, and careful use of these paths can guarantee the prefix-free property (Figure 2(e) gives such an example, in which the cycle is unfolded once to get a prefix-free path, in contrast to Fig. 2(d)). While we present this full algorithm in [7], the complication is not warranted here since long cyclic paths are almost certainly semantically uninteresting. In practice, we have extended findPathsDAG once again to allow limited exploration of cycles limited by (a) no more than k trips through visited nodes and (b) no more than l total path length. A bound on k and l is given in Theorem 4.3 and usually k and l are set to small numbers.

5.2 Three Methods for Finding Schema Embeddings

We next introduce our three heuristic embedding-search algorithms, namely, QualityOrdered, RandomOrdered and RandomMaxInd.

Finding Solutions with Ordered Algorithms. Our first two heuristics are based on a common subroutine Ordered, shown in Fig. 5. A key data structure is a table, C, where $C(A)$ is a set of known local embeddings for a source node A. The initialization of this table is discussed later. Given C and an ordered set O of source schema element types, Ordered tries to assemble a consistent solution σ by considering each A in O order (line 2), and trying to find a local embedding σ_A in $C(A)$ which can be merged with the existing σ without a conflict (lines 3-8).

Our first algorithm based on Ordered, QualityOrdered, is shown in Figure 6. In this algorithm, $C(A)$ is initialized with a single randomly chosen local embedding for each source node A. The ordering O is sorted by the *quality* of the local embedding.

In our second algorithm RandomOrdered (not shown), C is the complete set of local embeddings discovered so far for each source node (lines 4 and 5 in Figure 6), while O is a random ordering of source nodes (line 6 in Figure 6).

A Reduction Approach. We now discuss our third heuristic, RandomMaxInd. To understand this heuristic, consider

Algorithm Ordered (S_1, S_2, O, C)

Input: Schemas S_1 and S_2, an ordered set of source tags O, and C, a set of local embeddings for each source tag.
Output: a schema embedding from S_1 to S_2 if one is found.

σ := empty solution $(\emptyset, \emptyset)$;
for A in O
  for σ_A in $C(A)$
    c := conflict between σ and σ_A
    if c is null
      $\sigma = \sigma \oplus \sigma_A$; break;
  if c is not null
    findPathsRand $(G, A, L_{\mathsf{tar}}(A) - c)$;
if c is not null return $\emptyset$;
return σ;

Figure 5: Algorithm Ordered

Algorithm QualityOrdered (S_1, S_2)

Input: Schemas S_1 and S_2.
Output: a schema embedding from S_1 to S_2 if one is found.

count := 0;
while (count < MAX_TRIES) do
  count++;
  for each source node A
    $C(A)$:= {a local embedding, σ_A for A as found by findPathsRand };
  O := All source nodes, ordered by qual(σ_A, att);
  σ := Ordered (S_1, S_2, O, C);
  if $\sigma \neq \emptyset$
    return σ;
return $\emptyset$;

Figure 6: Algorithm QualityOrdered

the following problem defined on the table C of local mappings defined above:

PROBLEM:	Assemble-Embedding
INPUT:	Two DTDs S_1 and S_2, a similarity matrix att, and a table C.
OUTPUT:	A schema embedding $\sigma : S_1 \to S_2$, valid w.r.t. att, formed as the union of a subset of embeddings in C if one exists.

Composing a schema embedding from local embeddings in C is nontrivial:

Theorem 5.3: *The Assemble-Embedding problem is NP-hard for nonrecursive* DTD*s.* □

To cope with this, the RandomMaxInd heuristic takes the approach of reducing the Assemble-Embeddings problem to the problem of finding high-weight independent sets in a graph. It uses an existing heuristic solution [9] to produce partial or complete solutions to this problem, which can be used to create partial or complete embeddings.

Before describing our reduction, we review the definition of Max-Weight-Independent-Set. That problem is defined on an undirected graph $G = (V, E)$ (not to be confused with a schema graph) with node weights $w[v], v \in V$. The goal is to find a subset V' of V such that for v_i and v_j in V', there is no edge from v_i to v_j. In other words, $(v_i, v_j) \notin E$ and the weight of V', defined as $\sum_{v \in V'} w[v]$, is maximized.

Given an instance of the Assemble-Embedding problem, it is straightforward to construct an instance of Max-Weight-Independent-Set. First, for each local mapping $\sigma_a \in C(A)$ for any $A \in E_1$, we construct a vertex v_{σ_a} in V. Second, for each pair σ_a, σ_b of such mappings, we construct an edge between v_{σ_a} and v_{σ_b} if σ_a and σ_b *conflict*. The weight of v_{σ_a} is given as qual(σ_a, att).

To complete the algorithm on the resulting graph, we use an existing heuristic tool for Max-Weight-Independent-Set, which returns a subset V' of V. Finally, we construct an embedding σ by adding local embedding σ_a to σ for each $v_{\sigma_a} \in V'$. The quality of σ is warranted by the heuristic tool used, and its correctness is verified below.

Theorem 5.4: *If* $|V'| = |E_1|$, *σ constructed as above is a schema embedding from S_1 to S_2.* □

If σ is not a full embedding, we use findPathsRand to generate new local mappings, if any are available, for tags A *not* mapped by σ, and repeating the process until either it finds a valid embedding or it reaches a threshold of tries.

6 Experimental Study

In this section, we present an experimental evaluation of our schema embedding algorithms. Our approach is to vary the difficulty of the matching task by introducing artificial noise into a target schema, and measuring the ability of our algorithms to find an embedding.

Our experiments are based on real-world DTDs taken from a publicly available repository [31], plus the DTD of the XMark benchmark [34]. Each DTD was normalized into our graph representation. The XMark schema is the largest, with 57 productions after normalization. The XMark schema is apparently the most involved schema as the others scale better (see Fig. 10), and accordingly, we evaluate our algorithms for all the schemas but use the XMark schema for more detailed experiments.

Generating Target Schemas. Target schemas are generated from source schemas with added complexity and noise. Obviously, a target schema that is a simple copy of the source will match trivially. As we introduce noise, we take care to preserve this matching, but make it harder to find in a number of ways. This approach allows us to attribute any failure to find a matching to the algorithm rather than the data. Particular target schemas are generated according to a probability noise in two steps: First, for each edge in the schema, with probability noise, the edge in the target is replaced with a path of between 1 and 5 nodes. When new nodes are added, with probability .5, the name of the node is formed as a small mutation of an existing name. Also, the type of the deleted edge (AND, OR, STAR) is used as the type of the first introduced edge to ensure that the original mapping is still possible.

In the second step, each node in the target (including newly-added nodes) are visited again, and with probability noise, a new subtree is added under it. The new subtree adds between 1 and 10 nodes. After each subtree addition,

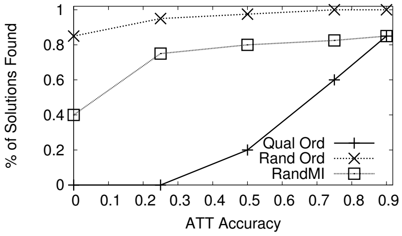


Figure 7: Varying accuracy

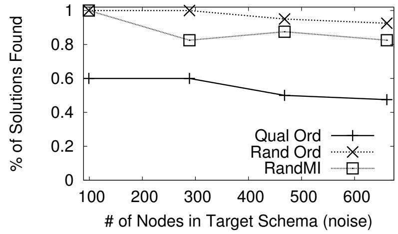


Figure 8: Noise vs. success rate

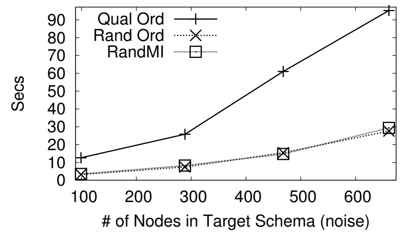


Figure 9: Noise vs. time

each leaf in the new subtree is visited, and with probability .5, an edge is added to an existing leaf outside the newly-added subtree. (This leaf may later have a subtree added under it.) The intuition for this last step is that confusion between different parts of the tree is more likely to arise if the same “attributes” (leaf nodes) appear in multiple places.

Generating the att. The similarity array, att, is initialized by computing pairwise string-edit distances between source and target tags (string edit distance with unit cost is also known as Damerau-Levenshtein distance). Furthermore, if a minimum threshold, sel, of similarity is not met by a pair, the similarity of that pair is set to 0, and as a result the tags cannot be matched. Note that the “similar names” introduced above range in similarity from .5 for short strings to over .8 for longer strings, and will be counted as potential matches in many experiments. There are also similar names in the schemas themselves, caused by the conversion of the schema to our graph format.

Clearly, sel, referred to as the *selectivity* of att, is an important parameter, as it directly determines the size of the candidate pool of target tags matching each source tag. Larger selectivities make the problem easier, and for our experimental data if sel is 1.0 (exact matches only), then finding a schema embedding reduces to finding valid prefix-free paths for each local embedding in the source schema.

A second important parameter is the *accuracy* of att. This matters greatly for heuristic algorithms, since the valid embedding in our generated data always has the highest average quality. Accuracy is implemented with a parameter c, which varies between 0 and 1. Each entry m in att is replaced by $cm+(1-c)\mathsf{rnd}$, where rnd is a random number from 0 to 1. A low accuracy tends to mislead heuristics that rely heavily on att. Combining a low accuracy with a very low selectivity makes the problem very difficult to solve.

Experimental Setting. Experiments are conducted by copying the source schema, adding some amount of noise based on the parameter noise, and adjusting the att according to sel and c. Then the three algorithms given in Section 5 (RandomOrdered, QualityOrdered and RandomMaxInd) are used to try to find embeddings. For the ordered algorihms, the set C is initialized by finding 3 random mappings for each A , and discarding the two with the lowest qual ratings. When not otherwise stated, experiments are run with $\mathsf{sel} = 0.6$, $c = 0.75$ (accuracy) and $\mathsf{noise} = 0.25$. Since all algorithms (and the noise introduction) have a random component, they are repeated with 40 different random seeds, and an average is used.

The software is written in Java, except for the external heuristic for maximum independent sets [8], which is an optimized C program. Experiments are run on a variety of machines with Pentium III processors running at either 933MHZ or 1.0GHZ, with 256MB of RAM.

Accuracy Results. Figure 7 shows how the three algorithms perform while varying accuracy, with $\mathsf{noise} = 0.25$. The y axis shows the percentages of runs for which a successful embedding is found. For this noise amount, the target schema is approximately three times as large as the source schema. This graph shows that QualityOrdered is extremely sensitive to the quality of the att values. It uses att extensively in its search pattern, and thus cannot find solutions unless att is accurate.

Figure 7 also shows that RandomOrdered finds correct solutions more frequently than RandomMaxInd. While RandomOrdered takes into account att when it is seeking its solution set, it tries to find alternative solutions based on the conflicts it detects, independent of the att values. RandomMaxInd seeks alternative solutions for nodes based solely on their weights, as defined by att. It does not use conflicts to guide its search.

Varying Target Schema Size. We also consider target schemas with different numbers of erroneous nodes and edges introduced. These results are shown in Fig. 8. Because this graph shows results when accuracy is 0.75, QualityOrdered does not do well, as expected. RandomOrdered and RandomMaxInd both find the correct solution the majority of the time, decreasing somewhat as noise increases. The running times are shown in Fig. 9.

Different Source Schemas. We also run tests with different source schemas. We vary noise over five different source schemas, using RandomOrdered and accuracy= 0.75. Figure 10 shows the running times for the various source schemas. For all runs across the different schemas, a solution was found more than 90% of the time (not shown).

Varying Selectivity. We also run experiments with different values of selectivity. Both RandomOrdered and RandomMaxInd find solutions less frequently as selectivity decreases (not shown). QualityOrdered is relatively indifferent to the selectivity level, finding approximately the same number of solutions at $\mathsf{sel} = 0.3$ as at $\mathsf{sel} = 0.7$. The running time increases dramatically, however, once sel falls below 0.4. The results are shown in Figure 11.

Discussion. Our experimental results show that, when a feasible matching exists, it is likely to be almost completely found for schema sizes of up to a few hundred

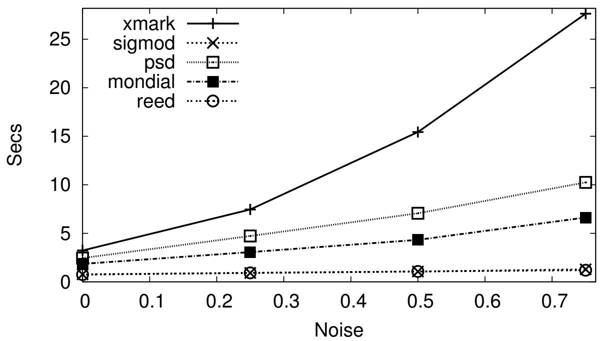


Figure 10: Time required for different source schemas

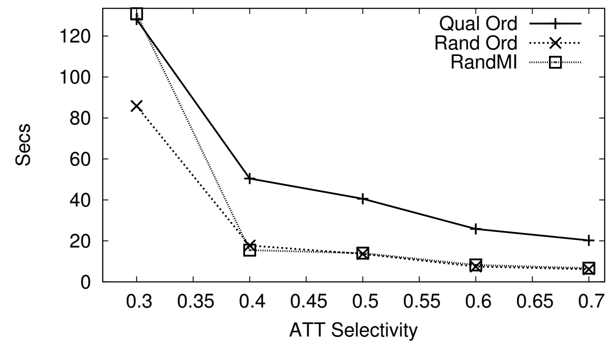


Figure 11: Varying Selectivity

nodes. While this does not demonstrate that similar results can be obtained with differing target schemas and the use of real-world tools to produce att, it is certainly promising. Further, we found that the randomized algorithm RandomOrdered performs better than RandomMaxInd, and that QualityOrdered only does well with a highly accurate att. Based on these results, we plan to integrate RandomOrdered and RandomMaxInd, since the external independent set heuristic is very fast in practice. Finally, we note that QualityOrdered may be important in practice, where the att values may in fact be reliable.

7 Related Work

A wide variety of techniques have been developed to solve different forms of schema matching for relational, ER and object-oriented models (e.g., [18, 22, 32, 12, 20]; see [33] for a recent survey). While these are not focused on XML DTD schema matching, 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 matching are [13, 23, 25, 26, 27, 30]. LSD [13] proposes powerful machine-learning techniques that make use of instance-level information to determine XML DTD tag mapping. Systems of [23, 25, 26] target a wide class of schemas and can be tailored to a variety of data models. The similarity flooding algorithm of [25] provides a novel schema matching tool based on graph-similarity. Cupid [23] is a generic schema matching tool that encompasses a variety of techniques such as linguistic analyses, referential constraints and context dependencies. Rondo [26] 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 [30] considers instance-level mappings based on schema matching, and uses a semi-automatic mechanism to match highly similar schemas. Clio [27] also focuses on deriving instance translation from schema mappings. Both TransScm and Clio adopt a similarity heuristic, and decouple the process of getting an instance mapping from schema matching. To our knowledge, no previous work has addressed information preservation for XML DTD schemas. Our notion of schema embedding extends graph similarity by allowing a source DTD schema to be mapped to a structurally different target DTD. It is capable of mapping multiple source DTDs to a single target schema, which is typical in data integration. Furthermore, an instance mapping can be *automatically* derived from a schema embedding and it *guarantees* both invertibility and query preserving w.r.t. regular XPath queries.

Information preservation has been studied for nested relational and complex data models (e.g., [3, 16, 28, 29]). [16] proposed several notions of dominance and studied their relationships, which were revisited in [28]. The focus of [3, 29] has mainly been on the information capacity of type constructs and structural transformation rules. 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 [16]. We revise these notions and study their basic properties for XML DTD schemas and XML queries, and 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.

Query preservation is related to query rewriting using views, which has been extensively studied for conjunctive and datalog queries for relational databases and regular path queries on semistructured data (e.g., [2, 10, 11, 21]; see [15, 19] for surveys). View-based query rewriting 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 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 mappings (views) that automatically preserves all the queries in an XML query language, rather than to determine the losslessness of views. It is worth remarking that Theorem 3.2 establishes a connection between invertibility and query rewriting. For example, if the query language $\mathcal{L}$ includes the identity query id, then a view σ_d is invertible and σ_d^{-1} is in $\mathcal{L}$ iff id has a rewriting in $\mathcal{L}$ using σ_d.

Also related to schema matching is the problem of type checking one schema against another. For XML schemas, the problem of determining the existence of a subsumption map has been studied (*e.g.,* [17]), but data restructuring and query translation are not addressed in that context.

8 Conclusions

We have studied information-preservation criteria for XML mappings by revising the notions of invertibility and query preservation [3, 16, 28, 29], and by establishing separation, equivalence and complexity results. In light of the difficulties of determining information preservation, we have introduced a novel notion of schema embedding for XML DTD schemas, from which an instance-level mapping is automatically derived and is guaranteed to be information preserving, type checking, 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 embedding candidates, which are efficient and accurate as shown by our experimental results. These provide a practical approach to specifying and computing information-preserving XML mappings.

This is a first step toward developing a useful tool for lossless XML data migration and integration. For future work, we plan to extend the notion of schema embedding to (a) accommodate more general XML schemas with constraints and inheritance, (b) allow one source type to map to different target types in *different contexts*, (c) allow certain queries in XQuery in the `path` function, and (d) preserve XQuery fragments as query languages.

Acknowledgments The authors would like to thank Michael Benedikt, Minos Garofalakis, Rajeev Rastogi, and Ali Shokoufandeh for helpful discussions on graph and schema matching. Renee Miller provided helpful comments on information-preservation semantics, and we thank Steve Fortune for his helpful comments on the prefix-free path problem.

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufman, 2000.

[2] S. Abiteboul and O. M. Duschka. Complexity of answering queries using materialized views. In *PODS*, 1998.

[3] S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 1988.

[4] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[5] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *PODS*, 2001.

[6] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. In *PODS*, 2005.

[7] P. Bohannon, W. Fan, M. Flaster, and P. P. S. Narayan. Information preserving XML schema embedding. *http:// homepages.inf.ed.ac.uk/wenfei/papers/embedding.pdf*, 2005.

[8] S. Busygin. QUALEX: QUick ALmost EXact maximum weight clique/independent set solver. *http://www. busygin.dp.ua/npc.html.*

[9] S. Busygin, S. Butenko, and P. M. Pardalos. A heuristic for the maximum independent set problem based on optimization of a quadratic over a sphere. *J. Comb. Optim.*, 6(3):287–297, 2002.

[10] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Lossless regular views. In *PODS*, 2002.

[11] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Rewriting of regular expressions and regular path queries. *JCSS*, 64(3):443–465, 2002.

[12] S. Castano, V. D. Antonellis, and S. D. C. di Vimercati. Global viewing of heterogeneous data sources. *TKDE*, 13(2):277–297, 2001.

[13] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD*, 2001.

[14] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[15] A. Y. Halevy. Theory of answering queries using views. *SIGMOD Record*, 29(4), 2001.

[16] R. Hull. Relative information capacity of simple relational database schemata. *SIAM J. Comput*, 15(3):239–265, 1986.

[17] G. M. Kuper and J. Simeon. Subsumption for XML types. In *ICDT*, 2001.

[18] L. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL – a language for interoperability in relational multi-database systems. In *VLDB*, 1996.

[19] M. Lenzerini. Data integration: A theoretical perspective. In *PODS*, 2002.

[20] B. S. Lerner. A model for compound type changes encountered in schema evolution. *TODS*, 25(1):83–127, 2003.

[21] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS*, 1995.

[22] W.-S. Li and C. Clifton. SemInt: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. *Data Knowl Eng*, 33(1):49–84, 2000.

[23] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *VLDB*, 2001.

[24] M. Marx. XPath with conditional axis relations. In *EDBT*, 2004.

[25] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm. In *ICDE*, 2002.

[26] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD*, 2003.

[27] R. J. Miller, M. A. Hernández, L. M. Haas, L.-L. Yan, C. T. H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1):78–83, 2001.

[28] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *VLDB*, 1993.

[29] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. Schema equivalence in heterogeneous systems: bridging theory and practice. *IS*, 19(1):3–31, 1994.

[30] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, 1998.

[31] U. of Washington. XML repository. *http://www.cs.washington.edu/research/xmldatasets.*

[32] L. Palopoli, D. Sacca, and D. Ursino. Semi-automatic semantic discovery of properties from database schemas. In *IDEAS*, 1998.

[33] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 2001.

[34] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, 2002.

Appendix

Proof of Theorem 3.1

(1) We first show that invertibility does not entail query preservation w.r.t. $\mathcal{X}$. Consider a source DTD S_1 and a target DTD S_2:

$S_1 = (\{r, A, B, C\}, P_1, r)$, where P_1 is:
$r \to A, \quad A \to B, C, \quad B \to A + \epsilon, \quad C \to \epsilon,$
$S_2 = (\{r, A\}, P_2, r)$, where P_2 is:
$r \to A, \quad A \to A + \epsilon.$

The mapping $\sigma_d : \mathcal{I}(S_1) \to \mathcal{I}(S_2)$ is defined 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 mapped to the child and the grandchild of v', respectively, and the A child of the B node is mapped to the great grandchild of v'. Formally, this can be expressed by the function path from the edges of T to (relative) paths of $\sigma_d(T)$ as follows:

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

Obviously σ_d is invertible: one can restore the original T from $\sigma_d(T)$ inductively top-down 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+1}. It is easy to prove that A^{3k+1} is not expressible in $\mathcal{X}$ by contradiction. Thus σ_d is not query preserving w.r.t. $\mathcal{X}$.

(2) We next show that query preservation w.r.t. $\mathcal{X}$ does not entail invertibility. Consider a source DTD S_1:

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

and the target DTD S_2 is identical to S_1.

The mapping $\sigma_d : \mathcal{I}(S_1) \to \mathcal{I}(S_2)$ is defined 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 children of r_1 are mapped to 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).

One can pose the following forms of $\mathcal{X}$ queries over $T \in \mathcal{I}(S_1)$: $\epsilon, A, A[q]$, 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. $\square$

Proof of Theorem 3.2

(1) Suppose that σ_d is query preserving w.r.t. $\mathcal{L}$. Then there exists a computable function $F : \mathcal{L} \to \mathcal{L}$ such that for any $Q \in \mathcal{L}$ and any $T \in \mathcal{I}(S_1)$, $Q(T) = F(Q)(\sigma_d(T))$. Since id is in $\mathcal{L}$, we have $T = id(T) = F(id)(\sigma_d(T))$ for any $T \in \mathcal{I}(S_1)$. That is, $\sigma_d^{-1} = F(id) \circ \sigma_d$, 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 $F : \mathcal{L} \to \mathcal{L}$ to be $F(Q) = Q \circ \sigma_d^{-1}$ for any $Q \in \mathcal{L}$. Obviously F is computable. Furthermore, for any $Q \in \mathcal{L}$ and $T \in \mathcal{I}(S_1)$, $Q(T) = Q(\sigma_d^{-1}(\sigma_d(T)) = F(Q)(\sigma_d(T))$. Thus F is an effective query translation function for $\mathcal{L}$. $\square$

Proof of Theorem 3.3

Suppose that $\sigma_d : S_1 \to S_2$ 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. It first creates the root r_1 of T, which is identical to the root r_2 of $\sigma_d(T)$. It then recursively expands T top-down, until T cannot be expanded further. More specifically, 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 $F : \mathcal{X}_R \to \mathcal{X}_R$, as follows. Note that there is a unique $\mathcal{X}_R$ path ρ from r_1 to v.

(1) $\alpha = A_1, \ldots, A_n$. For each A_i, define an $\mathcal{X}_R$ query $Q_i = F(\rho/A_i[position() = k]$ and evaluate $Q_i(\sigma_d(T))$ at the context node v in $\sigma_d(T)$, where k is the k-th A_i element in α if it has multiple A_i elements. Since $\sigma_d(T)$ is mapped from an XML tree T, Q' always returns a single node v_i. Treat $v_1, \ldots, v_n$ 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) $\alpha = A_1 + \ldots + A_n$. For each A_i, let $Q_i = F(\rho/A_i)$ and evaluate $Q_i(\sigma_d(T))$ at the context node v in $\sigma_d(T)$. Since $\sigma_d(T)$ is mapped from an XML tree T, there exists one and only one Q_i such that $Q_i(\sigma_d(T))$ returns a single note v_i (and the others return empty). Treat v_i 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 context node v in $\sigma_d(T)$, where $Q_k = F(\rho/B[position() = k]$, until it reaches a k_0 such that $Q_{k_0}(\sigma_d(T)) = \emptyset$. Since $\sigma_d(T)$ is mapped from an XML tree T, there exists one and only one v_k returned by $Q_k(\sigma_d(T))$ for each $k < k_0$, and for any $k \geq k_0$, $Q_k(\sigma_d(T) = \emptyset$. Treat v_k 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 via $\rho/text()$.

(5) $\alpha = \epsilon$. Nothing needs be done here.

This process terminates since $\sigma_d(T)$ is generated from a finite T. 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. 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 a query A/B over S_1 to an equivalent one over S_2. □

Proof of Theorem 3.4

(1) We prove the undecidability by reduction from the equivalence problem for relational-algebra (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 $Q_1 \equiv Q_2$, i.e., whether or not for any relational database I_1 of $\mathcal{R}_1$, $Q_1(I) = Q_2(I)$. The equivalence problem is undecidable (cf. [4]).

It suffices to show that the invertibility problem is undecidable for relational mappings defined in RA. Since relational data can be coded in XML and RA queries can be expressed in FO over XML trees, the undecidability carries over to XML mappings defined in FO.

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:

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

Observe that $Q_1 \equiv Q_2$ iff $\Delta(Q_1, Q_2) = \emptyset$, i.e., the query 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).

(2) The proof is similar to (1), by reduction from the equivalence problem for RA queries.

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 F such that $F(Q) = Q$. 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 F such that $Q' = F(Q)$. Recall I_1, I_2 given above. Obviously, $Q(I_1) \neq Q(I_2)$, while $Q'(V(I_1)) = Q'(V(I_2))$. Thus either $Q(I_1) \neq Q'(V(I_1))$ or $Q(I_2) \neq Q'(V(I_2))$; that is, F 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. □

Proof of Theorem 4.1

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 ρ_1 is 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 $\mathsf{path}(A, B_1)$ is 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 $\sigma_d(T)$. More specifically, this obviously holds if the parent of v is of a concatenation or disjunction type or str; and moreover, if the type 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 well defined, i.e., 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 $A \to B_1 + \ldots + B_k$, 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 root over S_1 such that $\rho_1 = \rho/A[q]/\rho_1'$ and $\rho_2 = \rho/A[q]/\rho_2'$ such that v_1, v_2 have the lowest common ancestor v that is identified by ρ and mapped to either u by σ_d or an ancestor u' of u. 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, which contra-

dicts the assumption. If v is mapped to u', since v_1, v_2 both exist and v is the lowest common ancestor of v_1 and v_2, v must have a concatenation type $A' \to B'_1, \ldots, B'_n$ such that v_1, v_2 are descendants of the B'_i, B'_j children of v; indeed, A' cannot be a Kleene star type since otherwise by the definition of σ_d, v_1, v_2 cannot be mapped to nodes that have a common ancestor u; and A' cannot be a disjunction type since v is the lowest common ancestor of v_1, v_2. Since both $\mathsf{path}(A', B'_i)$ and $\mathsf{path}(A', B_j)$ end up to be suffixes of $\rho/A[q]$, it contradicts the definition of schema embedding since either $\mathsf{path}(A', B'_i)$ is a prefix of $\mathsf{path}(A', B_j)$ or the other way around.

Similarly, it can be verified that violations cannot be caused by AND and STAR paths either. □

Proof of Theorem 4.2

(1) We first show that σ_d is invertible and query preserving w.r.t. $\mathcal{X}_R$. It suffices to define a query translation function $F : \mathcal{X}_R \to \mathcal{X}_R$. 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. The translation function F extends the mapping δ on $\mathcal{X}_R$ paths given earlier, by substituting $\mathsf{path}(A, B)$ for each edge (A, B) in a given $\mathcal{X}_R$ query. More specifically, given an $\mathcal{X}_R$ query Q over S_1, $F(Q)$ is computed by using functions f and reach defined below. For each element type A in E_1 and each sub-query Q_1 of Q,

- *the local translation* $f(Q_1, A)$ of Q_1 at A is an $\mathcal{X}_R$ query over S_2 such that for any instance T of S_1 and any A element a in T, the result of evaluating Q_1 at a in T is the same as the result of evaluating $f(Q_1, A)$ at a' on $\sigma_d(T)$, where a' is mapped from a by σ_d;
- $\mathsf{reach}(Q_1, A)$ is the set of element types in S_1 that are *reached* via Q_1 when evaluated at an A element in an instance T of S_1.

More specifically, $f(Q_1, A)$ and $\mathsf{reach}(Q_1, A)$ are defined based on the structure of query Q_1 as follows.

(a) If Q_1 is ϵ, then

$$\mathsf{reach}(Q_1, A) = \{A\},$$
$$f(Q_1) = \epsilon.$$

(b) If Q_1 is B, then

$$\mathsf{reach}(Q_1, A) = \{B\},$$
$$f(Q_1, A) = \mathsf{path}(A, B).$$

(c) If Q_1 is $p/text()$, then

$$\mathsf{reach}(Q_1, A) = \emptyset,$$
$$f(Q_1, A) = f(p, A)/(\bigcup_{B \in \mathsf{reach}(p,A)} \mathsf{path}(B, \mathsf{str})).$$

(d) If Q_1 is p_1/p_2, then

$$\mathsf{reach}(Q_1, A) = \bigcup_{B \in \mathsf{reach}(p_1,A)} \mathsf{reach}(p_2, B),$$
$$f(Q_1, A) = f(p_1, A)/(\bigcup_{B \in \mathsf{reach}(p_1,A)} f(p_2, B)).$$

(e) If Q_1 is $p_1 \cup p_2$, then

$$\mathsf{reach}(Q_1, A) = \mathsf{reach}(p_1, A) \cup \mathsf{reach}(p_2, A),$$
$$f(Q_1, A) = f(p_1, A) \cup f(p_2, A).$$

(f) If Q_1 is p^*, then

$$\mathsf{reach}(Q_1, A) = \{A\} \cup \mathsf{reach}(p, A),$$
$$f(Q_1, A) = f(p_1, A)^*.$$

(g) If Q_1 is $p[q]$, then

$$\mathsf{reach}(Q_1, A) = \mathsf{reach}(p, A),$$
$$f(Q_1, A) = f(p_1, A)[\bigcup_{B \in \mathsf{reach}(p,A)} f(q, B)].$$

(h) If Q_1 is $[p]$, then

$$\mathsf{reach}(Q_1, A) = \mathsf{reach}(p, A),$$
$$f(Q_1, A) = f(p, A).$$

(i) If Q_1 is $[p/text() = c]$, then

$$\mathsf{reach}(Q_1, A) = \emptyset,$$
$$f(Q_1, A) = f(p, A)/(\bigcup_{B \in \mathsf{reach}(p,A)} \mathsf{path}(A, \mathsf{str})).$$

(j) If Q_1 is $[position() = k]$, then

$$\mathsf{reach}(Q_1, A) = \emptyset,$$
$$f(Q_1, A) = [position() = k].$$

(k) If Q_1 is $\neg q$, then

$$\mathsf{reach}(Q_1, A) = \emptyset,$$
$$f(Q_1, A) = [\neg f(q, A)].$$

(l) If Q_1 is $q_1 \wedge q_2$, then

$$\mathsf{reach}(Q_1, A) = \emptyset,$$
$$f(Q_1, A) = [f(q_1, A) \wedge f(q_2, A)].$$

(m) If Q_1 is $q_1 \vee q_2$, then

$$\mathsf{reach}(Q_1, A) = \emptyset,$$
$$f(Q_1, A) = [f(q_1, A) \vee f(q_2, A)].$$

Given these, we define $F(Q)$ to be $f(Q, r_1)$. One can easily verify that F is indeed a query translation function such that for any instance T of S_1, $Q(T) = F(Q)(\sigma_d(T))$, again by induction on the structure of Q.

(2) For the complexity of the query translation function F, note that $|\mathsf{reach}(Q, A)|$ is bounded by $|S_1|$ and thus $f(Q, A)$ is bounded by $O(|Q|\ |\mathsf{path}|\ |S_1|)$. The computation of $f(Q, A)$ and $\mathsf{reach}(Q, A)$ can be conducted by dynamic programming, and it takes at most $O(|Q|\ |\mathsf{path}|\ |S_1|)$ time to compute $F(Q)$.

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. □

Proof of Theorem 4.3

Suppose that there exists 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 $A \to \mathsf{str}$. As in (1), $\mathsf{path}(A, B)$ is no longer than $|E_2|$. □

Proof of Theorem 5.1

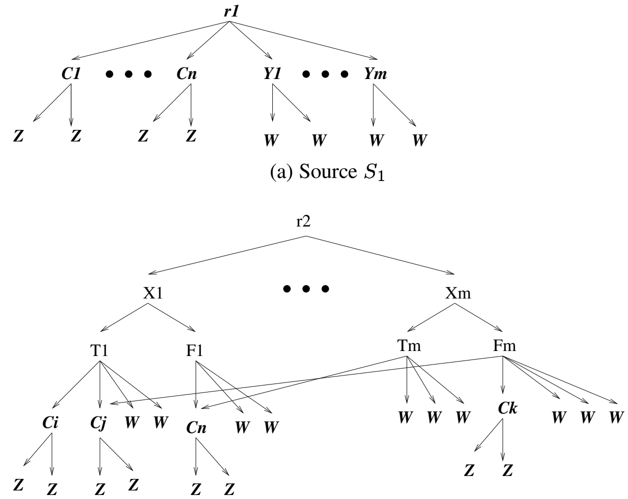
We show that the schema embedding problem is NP-complete. A NP algorithm is as follows: guess a mapping, and then check whether it is an embedding; the latter can be done in PTIME. The NP-hardness is verified by reduction from 3SAT, which is NP-complete (cf. [14]). It suffices to show that the problem is NP-hard for nonrecursive DTDs, by reduction from 3SAT. 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, $\mathsf{att}(A, B) = 1$, i.e., 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*/

$S_2 = (E_2,\ P_2,\ r_2)$, 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:



(a) Source S_1

(b) Target S_2

Figure 12: DTD schemas in the proof of Theorem 5.1

$r \to X_1, \ldots, X_m$,
$X_i \to T_i, F_i$, /* 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 are depicted in Fig. 12(a) and 12(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, $Y_1, \ldots, Y_m$ 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*. This is asserted by the number of W children below Y_s and T_s, F_s.

We next show that S_1, S_2 are indeed a reduction from 3SAT, i.e., 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 $\lambda(C_i) = C_i$, $\lambda(Z) = Z$, $\lambda(W) = W$, $\lambda(Y_i) = F_i$ if $\mu(x_i)$ is *true*, $\lambda(Y_i) = T_i$ 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) $\lambda(C_i)$ is either C_i or V_i, where V_i is either T_i or F_i; and (2) $\lambda(Y_j)$ is mapped to V_j, where V_j is either T_j or F_j, such that $\lambda(Y_j) \neq \lambda(Y_k)$ and $\lambda(Y_j) \neq \lambda(C_i)$ for $k \neq j$, $i \neq j$; and furthermore, for each x_j there exists Y_j such that $\lambda(Y_j) = V_j$. 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 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 $\lambda(Y_s) = T_s$. It is easy to verify that μ satisfies ϕ. $\square$

Proof of Theorem 5.2

We show that the Local-Embedding problem is NP-hard for nonrecursive DTDs, by reduction from 3SAT. Given an instance $\phi = C_1 \wedge \cdots \wedge C_n$ of 3SAT, we define two nonrecursive DTDs S_1, S_2, where S_1 consists of a single production, and a similarity matrix att such that ϕ is satisfiable iff there is a valid schema embedding from S_1 to S_2 w.r.t. att. 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
$E_1 = \{r_1\} \cup \{C_i \mid i \in [1, n]\} \cup \{Y_s \mid s \in [1, m]\}$;
P_1 is defined as:
$\quad r \to C_1, \ldots, C_n,\ Y_1, \ldots, Y_m$

$S_2 = (E_2,\ P_2,\ r_2)$, where
$E_2 = \{r_2\} \cup \{C_i \mid i \in [1, n]\} \cup \{X_s, T_s, F_s \mid s \in [1, m]\}$;
P_2 is defined as:
$\quad r \to X_1, \ldots, X_m,$
$\quad X_i \to T_i, F_i,$ /* for $i \in [1, m]$*/
$\quad T_i \to C_{i_1}, \ldots, C_{i_k}$ /* all C_{i_j} in which x_i appears*/
$\quad F_i \to C'_{i_1}, \ldots, C'_{i_k}$ /* all C'_{i_j} in which $\overline{x_i}$ appears */
$\quad A \to \epsilon$ /* for A ranging over C_i.

Similar to the proof of Theorem 5.1, we use $Y_1, \ldots, Y_m$ in S_1 to code the "negation" of a truth assignment μ to variables in ϕ. In both S_1 and S_2, C_i is to code clause C_i. Note that both S_1, S_2 are nonrecursive and are defined in terms of concatenation types only. Furthermore, S_1 consists of a single production.

The similarity matrix att is defined such that

$\mathsf{att}(C_i, C_i) = 1$, and $\mathsf{att}(C_i, X) = 0$ for any $X \neq C_i$,
$\mathsf{att}(Y_j, Z_j) = 1$ if $Z_j = T_i$ or $Z_j = F_i$, and
$\mathsf{att}(Y_j, Z) = 0$ if $Z \neq T_j$ and $Z \neq F_j$

That is, C_i in S_1 can only map to C_i in S_2, and Y_j in S_1 can only map to either T_j or F_j.

Along the same lines as in the proof of Theorem 5.1, one can verify that Y_j in S_1 can only be mapped to the negation of the truth value of X_j, and as a result, ϕ is satisfiable iff there is a valid schema embedding from S_1 to S_2 w.r.t. att. $\square$

Algorithm findpaths (G, s, L_{tar})
Input: Directed graph G, source node s,
a bag of target nodes $L_{\mathsf{tar}} = \{\!|t_1, \ldots, t_k|\!\}$.
Output: Paths $\rho_1, \ldots, \rho_k$ satisfying the prefix-free condition,
or "no" if such paths do not exist.

```
1. Let G_r be the subgraph of G reachable from s
2.     if any t_i is not in G_r return "no";
3. Let G_c be the component graph of G_r; /* see text */
4. Let C be the subset of the components c_i in G_c
       with a nonempty target shadow; /* see text */
5. Let L'_tar be the subset of L_tar not in the shadow of any c_i;
6. Add each c_i with a nonempty target shadow to L'_tar;
7. Let G' be G_c with the shadow of each c_i removed;
8. Let P =findPathsDAG (G', s, L''_tar);
9. For each P_i ∈ P that ends at a new target for some c_i
10.     remove P_i from P;
11.     add findpathCycle (G, G_c, P_i, c_i, L_tar) to P;
12. return P;
```

Figure 13: Algorithm findpaths

Prefix-free Paths in Cyclic Graphs

In Section 5.1, an algorithm was given for finding paths in an acyclic graph. In this section, we describe how this algorithm can be generalized to handle cycles.

The overall algorithm has three parts: 1) break up the original graphs into a DAG of connected components, 2) solve this problem for the DAG case, 3) add back in the connected components and 4) use the cycles found in these components to create prefix-free paths to nodes in the component or reachable therefrom. We now describe these steps in more detail.

In Figure 13 the overall algorithm for finding a set of prefix-free paths from a single root s to a bag of target nodes in a possibly cyclic graph is given. In step 3 the algorithm computes a "component graph" of a graph G which we define as the graph produced from G by replacing each *cyclic connected component* in G with a new node, c_i, and creating edges to and from c_i for each edge which entered or left component i in the original graph. Note that the resulting graph is a directed graph containing component nodes c_i as well as any node in G that did not participate in a cycle.

We now introduce some notations: a node $n \in V$ is said to be *in the shadow* of c_i if n can be reached in G by a node in c_i. The *target shadow* of c_i is the subset of L_{tar} in the shadow of c_i. Note that a target node may be in the shadow of more than one component node.

To continue with the discussion of findpaths, the algorithm at line 4-5 removes cyclic components and all nodes reachable from them from the component graph. For each cyclic node that contains a target node or from which a target node is reachable, a node for that component is added back to the graph as a new target node. The idea is that the computed path to this node will be used as the prefix for all paths in that component or reachable therefrom. A bag L''_{tar} is computed from L_{tar} by removing any target

Algorithm findpathCycle $(G, G_c, \rho, c_i, L_{\mathsf{tar}})$

Input: Directed graph G, component graph G_c,
connected component c_i, path to c_i in G_c
target nodes $T = t_1, \ldots t_k$ in shadow of c_i

Output: Paths $\rho_1, \ldots, \rho_k$ for $t_1 \ldots t_k$.

while L_{tar} not empty
  let Y be a cycle in L_{tar} containing at least
    one node from $L_{\mathsf{tar}} \cap c_i$ and the last node in ρ;
  count := 0;
  for each node n in L_{tar} in reverse topological order
    s.t. n is reachable from Y
    output $\rho_n = \rho$ + count trips around Y
      + path from last node in Y to n;
    remove n from L_{tar};
    count++;

Figure 14: Algorithm findpathCycle

nodes no longer in the graph and adding any of the just-mentioned component nodes. Finally, for each component node c_i with a non-empty target cycle, findpathCycle is called to construct paths to target nodes reachable from c_i.

Procedure findpathCycle is shown in Figure 14. The intuition for this algorithm is as follows: consider a path consisting of the nodes in a cycle, say $y_1, \ldots, y_n, y_1$, followed by three nodes, a, b, c. How can we construct prefix free paths to a, b and c? Clearly we can create an arbitrary number of paths to each node by going around $y_1, \ldots, y_n, y_1$ some number of times. If we form paths to b and c, say P_b and P_c, it is easy to see that the prefix-free property will only hold if there are *more* instances of the y cycle in the path to P_b than that to P_c. $\square$

Proof of Theorem 5.3

We show that the Assemble-Embedding problem is NP-hard for nonrecursive DTDs, by reduction from 3SAT. Given an instance $\phi = C_1 \wedge \cdots \wedge C_n$ of 3SAT, we define two nonrecursive DTDs S_1, S_2 and a table local such that ϕ is satisfiable iff there exists a valid schema embedding from S_1 to S_2 that is formed by composing a subset of the assignments found in local. The similarity matrix att is defined such that for all element types A in S_1 and B in S_2, $\mathsf{att}(A, B) = 1$, i.e., 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
$E_1 = \{r_1\} \cup\ \{C_i, V_i \mid i \in [1, n]\}\ \cup \{X_j \mid j \in [1, m]\}$;
P_1 is defined as:
  $r \to C_1, \ldots, C_n$
  $C_i \to V_i$
  $V_i \to X_i^1, X_i^2, X_i^3$ /* X_i^1, X_i^2, X_i^3 are the variables in C_i */

$S_2 = (E_2,\ P_2,\ r_2)$, where
$E_2 = \{r_2\} \cup \{C_i, V_{(i,j)} \mid i \in [1, n], j \in [1, 7]\}$
  $\cup \{T_j, F_j \mid j \in [1, m]\}$;
P_2 is defined as:
  $r \to C_1, \ldots, C_n,$
  $C_i \to V_{(i,1)}, \ldots, V_{(i,7)},$ /* for $i \in [1, n]$*/
  $V_{(i,j)} \to Y_{s1}, Y_{s2}, Y_{s3}$ /* see below */
  $A \to \epsilon$ /* for A ranging over C_i.

The production for $V_{(i,j)}$ is to code a possible truth assignment μ_i to variables in C_i such that μ_i satisfies C_i. More specifically, assume that X_{s1}, X_{s2}, X_{s3} are the variables in C_i. Then Y_{s1}, Y_{s2}, Y_{s3} are the truth values of a possible truth assignment μ_i to these variables, i.e., Y_{sl} is either T_{sl} or F_{sl}, $l \in [1, 3]$, such that there exists Y_{sj} for $j \in [1, 3]$ that is T_{sj} if X_{sj} is positive in C_i, and it is F_{sj} if X_{sj} is negative in C_i. That is, not all Y_{s1}, Y_{s2}, Y_{s3} are the "negations" of the truth values that satisfy C_i. For example, if $C_i = x_1 \vee \bar{x_2} \vee x_3$, then $V_{(i,j)}$ is to specify all the truth values of x_1, x_2, x_3 except F_1, T_2, F_3. Intuitively, S_2 codes ϕ and S_1 is to select a truth assignment for the variables.

The table local is given as follows.

- $r \to C_1, \ldots, C_n$: $\mathsf{local}(r)$ consists of a single local embedding such that $r \mapsto r$, $C_i \mapsto C_i$, with corresponding edge mapping.
- $C_i \to V_i$: $\mathsf{local}(C_i)$ consists of seven local embeddings, namely, $C_i \mapsto C_i$, $V_i \mapsto V_{(i,j)}$ for $j \in [1, 7]$, with corresponding edge mapping.
- $V_i \to X_i^1, X_i^2, X_i^3$: $\mathsf{local}(V_i)$ consists of seven local embeddings, namely, $V_i \mapsto V_{(i,j)}$ and $X_i^j \mapsto Y_{sj}$, for $j \in [1, 7]$, with corresponding edge mapping.

That is, each of the local embeddings in $\mathsf{local}(V_i)$ is a truth assignment μ_i to those variable involved in the clause C_i such that μ_i satisfies C_i. In other words, any local embedding in $\mathsf{local}(V_i)$ satisfies C_i. Note that a schema embedding formed composing a subset of local has to take one and only one local embedding from $\mathsf{local}(X)$ for each X in S_1 such that these local embeddings do not conflict with each other. That is, each variable in the embeddings has a unique truth value; in other words, there cannot be local embeddings $\sigma_i = (\lambda_i, \mathsf{path}_i)$ from $\mathsf{local}(V_i)$ and $\sigma_j = (\lambda_j, \mathsf{path}_j)$ from $\mathsf{local}(V_j)$ such that $\sigma_i(X_s) = T_s$ and $\sigma_j(X_s) = F_s$ for any X_s.

One can verify that ϕ is satisfiable iff there exists a valid schema embedding $\sigma : S_1 \to S_2$ formed by composing a subset of local, one from the set of local embeddings for each production in S_1. $\square$

Proof of Theorem 5.4

Suppose that the algorithm for Max-Weight-Independent-Set returns a subset V' of V such that $|V'| = |E_1|$. We show that σ constructed from V' and local is a valid embedding. To do so, it suffices to show the following: (1) for any element type A in E_1, there exists a unique v_{σ_A} in V' such that σ_A is a local embedding for A; and (2) for any $v_{\sigma_A}, v_{\sigma_B}$ in V', σ_A and σ_B do not conflict with each other. For if these hold, then the union of all the local embeddings corresponding to nodes in V' is a valid embedding from S_1 to S_2.

To see (1), observe that in the construction of G, for any pair local embeddings σ_A, σ'_A for the same element type A, there is a conflict edge between v_{σ_A} and $v_{\sigma'_A}$. Since

V' is an independent set, it cannot possibly contain more than one local embedding for A. Then from $|V'| = |E_1|$ it follows that for each type A in E_1, there exists exactly one local embedding v_{σ_A} for A such that v_{σ_A} is in V'.

To see (2), note that V' is an independent set, and thus for any nodes $v_{\sigma_A}, v_{\sigma_B}$ in V', there exists no conflict edge between the two in G. That is, σ_A and σ_B are consistent with each other. $\square$