Constructing Complex Semantic Mappings Between XML Data and Ontologies

Yuan An[1], Alex Borgida[2], and John Mylopoulos[1]

[1] University of Toronto, Canada
{yuana, jm}@cs.toronto.edu
[2] Rutgers University, USA
borgida@cs.rutgers.edu

Abstract. Much data is published on the Web in XML format satisfying schemas, and to make the Semantic Web a reality, such data needs to be interpreted with respect to ontologies. Interpretation is achieved through a *semantic mapping* between the XML schema and the ontology. We present work on the heuristic construction of *complex* such semantic mappings, when given an initial set of simple correspondences from XML schema attributes to datatype properties in the ontology. To accomplish this, we first offer a mapping formalism to capture the semantics of XML schemas. Second, we present our heuristic mapping construction algorithm. Finally, we show through an empirical study that considerable effort can be saved when constructing complex mappings by using our prototype tool.

1 Introduction

An important component of the Semantic Web vision is the annotation, using formal ontologies, of material available on the Web. Semi-structured data, published in XML and satisfying patterns expressed in DTD or XML Schema form an important subclass of such material. In this case, the annotation can be expressed in a formal way, through a semantic mapping connecting parts of the schema with expressions over the ontology. For example, [1,11] essentially connect paths in XML to chains of properties in an ontology. Such mappings have already found interesting applications in areas such as data integration as well as peer-to-peer data management systems [7].

Mappings from database schemas to ontologies could be as simple as value correspondences between single elements or as complex as logic formulas. In most applications, such as information integration, complex logic formulas are needed. Until now, it has been assumed that *humans* specify these complex mapping formulas — a highly complex, time-consuming and error-prone task. In this paper, we propose a tool that assists users in the construction of complex mapping formulas between XML schemas and OWL ontologies, expressed in a subset of First Order Logic.

Inspired by the success of the *Clio* tool [14,15], our tool takes three inputs: an ontology, an XML schema (actually, its unfolding into tree structures that we will call *element trees*), and simple correspondences between XML attributes and ontology datatype properties, of the kind possibly generated by already existing tools (e.g., [4,12,13]). The output is a ranked list of complex formulas representing semantic mappings of the kind described earlier.

In short, the main contributions of this work are as follows: (i) we propose a mapping formalism to capture the semantics of XML schemas based on tree-pattern formulas [3]; (ii) we propose a heuristic algorithm for finding semantic mappings, which are akin to a tree connection embedded in the ontology; (iii) we enhance the algorithm by taking into account information about (a) XML Schema features such as occurrence constraints, `key` and `keyref` definitions, (b) cardinality constraints in the ontology, and (c) XML document design guidelines under the hypothesis that an explicit or implicit ontology existed during the process of XML document design; (iv) we adopt the accuracy metric of schema matching [13] and evaluate the tool with a number of experiments.

The rest of the paper is organized as follows. Section 2 discusses related work, while Section 3 presents formal notations used later on. Section 4 describes some principles, as well as the mapping construction algorithm. Section 5 reports on empirical studies and Section 6 discusses how to refine the results by reasoning about ontologies. Finally, Section 7 summarizes the results of this work and suggests future directions.

2 Related Work

Much research has focused on converting and storing XML data into relational databases [16]. It is natural to ask whether we could utilize the mapping algorithm we have developed in [2] – for discovering mappings from relational schemas to ontologies – by first converting XML DTDs/schemas into relational tables. Unfortunately, this approach does not work. Among others, the algorithms that generate a relational schema from an XML DTD use backlinks and system generated *id*s in order to record the nested structure, and these confuse the algorithms in [2], which rely heavily on key and foreign key information.

The *schema mapping* tool *Clio* [14,15] discovers formal queries describing how target schemas can be populated with data from source schemas, given sets of simple value correspondences. The present work can be viewed as extending *Clio* to the case when the target schema is a ontology treated as a relational schema consisting of unary and binary tables. However, as argued in [2], the chase algorithm of *Clio* would not produce the desired mappings due to several reasons: (i) the chase only follows nested referential constraints along one direction, while the intended meaning of an XML element tree may follow a binary relationship along either direction (see also Section 4.1); (ii) *Clio* does not explore occurrence constraints in the XML schema. These constraints carry important semantic information in searching for “reasonable” connections in the ontology.

The Xyleme [5] project is a comprehensive XML data integration system which includes an automatic mapping generation component. A mapping rule in terms of a pair of paths in two XML data sources is generated based on term matching and structural, context-based constraints. Specifically, terms of paths are first matched syntactically and semantically. Then the structural information is exploited. Our work differs from it significantly in that we propose to discover the mappings between tree structures in XML data and that in ontologies. The discovery is guided by a forward engineering process.

The problem of *reverse engineering* is to extract a conceptual schema (UML diagram, for example) from an XML DTD/schema [8]. The major difference between *reverse engineering* and our work is that we are given an existing ontology, and want to interpret the XML data in terms of it, whereas reverse engineering aims to construct a new one.

Finally, *Schema Matching* [4,12,13] identifies semantic relations between schema elements based on their names, data types, constraints, and structures. The primary goal is to find the one-one simple correspondences which are part of the input for our mapping discovery algorithm.

3 Formal Preliminaries

An OWL ontology consists of classes (unary predicates over individuals), object properties (binary predicates relating individuals), and datatype properties (binary predicates relating individuals with values). Classes are organized in terms of a subClassOf/ISA hierarchy. Object properties and their inverses are subject to cardinality restrictions; the ones used here are lower bound of 1 (marking *total* relationships), and upper bound of 1 (called *functional* relationships). We shall represent a given ontology using a directed graph, which has class nodes labeled with class names C, and edges labeled with object properties p. (Sometimes, when we speak class C, we may mean its corresponding node in the ontology graph.) Furthermore, for each datatype property f of class C, we create a separate attribute node $N_{f,C}$ labeled f and an edge labeled f too from C to $N_{f,C}$ in the graph. We propose to have edge p from C to B, written in the text as $\boxed{\texttt{C}}$ `-- p --` $\boxed{\texttt{B}}$, to represent that p has domain class C and range class B. (If the relationship is functional, we write $\boxed{\texttt{C}}$ `-- p -->-` $\boxed{\texttt{B}}$.) We may also connect C to B by edge labeled p if we find a restriction stating that each instance of C is related to *some (all)* instances of B by p. For the sake of space limitation, graphical examples of ontologies (see [2]) are omitted.

For our purpose, we require that each XML document be described by an XML schema consisting of a set of element and attribute type definitions. Specifically, we assume the following countably infinite disjoint sets: **Ele** of element names, **Att** of attribute names, and **Dom** of simple type names including the built-in XML schema datatypes. Attribute names are preceded by a "@" to distinguish them from element names. Given finite sets $E \subset$**Ele** and $A \subset$**Att**, an XML schema over (E, A) specifies the type of each element ℓ in E, the attributes that ℓ has, and the datatype of each attribute in A. Specifically, an element type τ is defined by the grammar $\tau ::= \epsilon|\mathsf{Sequence}[\ell_1 : \tau_1, ...\ell_n : \tau_n]|\mathsf{Choice}[\ell_1 : \tau_1, .., \ell_n : \tau_n]$, where $\ell_1, .., \ell_n \in E$, ϵ is for the empty type, and Sequence and Choice are complex types. Each element associates an occurrence constraint with two values: *minOccurs* indicating the minimum occurrence and *maxOccurs* indicating the maximum occurrence. (We mark with * multiply occurring elements.) The set of attributes of an element $\ell \in E$ is defined by the function $\rho : E \to 2^A$; and the function $\kappa : A \to$**Dom** specifies the datatypes of attributes in A. For brevity, in this paper we do not consider *simple type elements* (corresponding to DTD's **PCDATA**), assuming instead that they have been represented using attributes. We also assume the Unique Name Assumption (UNA) for attributes, i.e., for any two elements $\ell_i, \ell_j \in E$, $\rho(\ell_i) \cap \rho(\ell_j) = \emptyset$.

For example, an XML schema describing articles and authors has the following specification:
E =$\{article, author, contactauthor, name\}$,
A =$\{@title, @id, @authorid, @fn, @ln\}$,
$\tau(article)$ = $\mathsf{Sequence}[(author)* : \tau(author), contactauthor{:}\epsilon]$,
$\tau(author)$ = $\mathsf{Sequence}[name{:}\epsilon]$,
$\rho(article) = (@title)$, $\rho(author) = (@id)$, $\rho(contactauthor) = (@authorid)$, $\rho(name) = (@fn, @ln)$, $\kappa(@title)$ = String, $\kappa(@authorid)$ = Integer, $\kappa(@id)$= Integer, $\kappa(@fn)$= String, $\kappa(@ln)$= String, and the element *article* is the root. Note that for the *article* element, *title* and *contactauthor* only occur once, while *author* may occur many times. For the *author* element, *name* occurs once.

The XML Schema Language is an expressive language that can also express `key` and `keyref` constraints.

An XML schema can be viewed as a directed node-labeled graph called *schema graph* consisting of the following edges: parent-child edges $e = \ell \to \ell_i$ for elements $\ell, \ell_i \in E$ such that if $\tau(\ell)$= $\mathsf{Sequence}[...\ell_i : \tau_i...]$ or $\mathsf{Choice}[...\ell_i : \tau_i...]$; and attribute edges $e = \ell \to \alpha$ for element $\ell \in E$ and attribute $\alpha \in A$ such that $\alpha \in \rho(\ell)$. For a parent-child edge $e = \ell \to \ell_i$, if the *maxOccurs* constraint of ℓ_i is 1, we show the edge to be functional, drawn as $\ell \Rightarrow \ell_i$. Since attributes are single-valued, we always draw an attribute edge as $\ell \Rightarrow \alpha$. The schema graph corresponding to the XML schema above is shown in Figure 1.

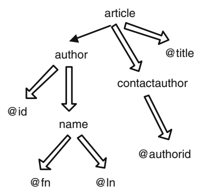

Fig. 1. The Schema Graph

Elements and attributes as nodes in a schema graph are located by path expressions. To avoid regular expressions, we will use a simple path expression $Q = \epsilon | \ell.Q$. In order to do this in a general fashion, we introduce the notion of *element tree*.

An *element tree* represents an XML structure whose semantics we are seeking. A semantic mapping from the entire XML schema to an ontology consists of a set of mapping formulas each of which is from an element tree to a conjunctive formulas in the ontology. An *element tree* can be constructed for *each element* by doing a depth first search (DFS). During the DFS, shared attributes are renamed to maintain the UNA, and cycles are unfolded. For the schema graph shown in Figure 2 (a), the element trees for the

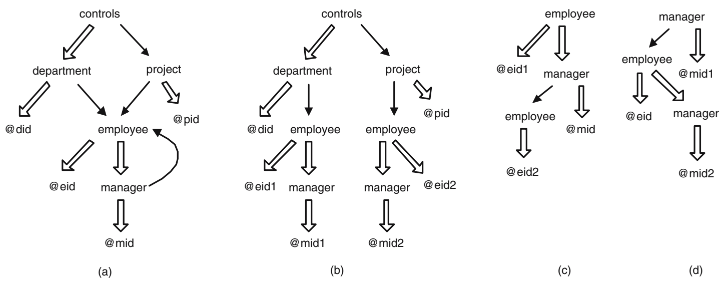

Fig. 2. Schema Graph and Element Trees

elements *controls*, *employee*, and *manager* are shown in Figure 2 (b), (c), (d). For simplicity, we specify each element tree as rooted in the element from which the tree is constructed.

Now we turn to the mapping language describing XML schemas in terms of ontologies. On the XML side, we start with *attribute formulas*, which are specified by the syntax $\alpha ::= \ell|\ell(@a_1 = x_1, .., @a_n = x_n)$, where $\ell \in E$, $@a_1, .., @a_n \in A$, and $x_1, \ldots, x_n$ are distinct variables. *Tree formulas* over (E, A) are defined by $\varphi ::= \alpha|\alpha[\varphi_1, .., \varphi_n]$, where α are attribute formulas over (E, A). For example,
$employee(@eid1 = x_1)[manager(@mid = x_2)[employee(@eid2 = x_3)]]$
is the tree formula representing the element tree in Figure 2 (c).

On the ontology side, we use conjunctive formulas, which treat concepts and properties as unary and binary predicates respectively.

A *mapping formula* between an element tree and an ontology then has the form $\Phi(\overline{X}) \rightarrow \Psi(\overline{X}, \overline{Y})$, where $\Phi(\overline{X})$ is a tree formula in the XML schema and $\Psi(\overline{X}, \overline{Y})$ is a conjunctive formula in the ontology. For example, given an ontology containing the class *Employee*, with a datatype property *hasId*, and a functional property *hasManager* (whose inverse is *manages*, which is not functional), the following mapping formula ascribes a semantics of the element tree in Figure 2 (c):
$employee(@eid1 = x_1)[$
$\quad manager\ (@mid = x_2)[$
$\qquad employee\ (@eid2{=}x_3)\ ]] \rightarrow$
$Employee(Y_1), hasId(Y_1, x_1),\ Employee(Y_2), hasId(Y_2, x_2),$
$hasManager(Y_1, Y_2),\ Employee(Y_3), hasId(Y_3, x_3), manages(Y_2, Y_3).$
Since we maintain the UNA assumption, we can drop the variable names x_is, and just use attribute names in the formula. The variables Y_js are implicitly existentially quantified and refer to individuals in the ontology.

Given an element tree T and an ontology O, a *correspondence* $P.@c \leftrightsquigarrow C.f$ will relate the attribute "$@c$" of the element E reached by the simple path P to the datatype property f of the class C in the ontology. A simple path P is always relative to the root of the tree. For example, we can specify the following correspondences for the element tree in Figure 2 (c):
$employee.@eid1 \leftrightsquigarrow Employee.hasId,$
$employee.manager.@mid \leftrightsquigarrow Employee.hasId.$
$employee.manager.employee.@eid2 \leftrightsquigarrow Employee.hasId$

Since our algorithm deals with ontology graphs, formally a correspondence L will be a mathematical relation $L(P, @c, C, f, N_{f,C})$, where the first two arguments determine unique values for the last three.

4 Mapping Construction Algorithm

Before presenting the algorithm, we first explain some principles underlying it.

4.1 Principles

As in the relational case [2], we start from a methodology presented in the literature [6,9] for designing XML DTDs/schemas from an ontology/conceptual model (CM). As with relational schemas, there is a notion of XML normal form (XNF) for evaluating the absence of redundancies and update anomalies in XML schemas [6]. The methodology in [6] claims to develop XNF-compliant XML schemas from CMs. It turns out that these "good" XML schemas are trees embedded in the graph representations of the CMs. Using the term *"element tree"* instead of *"schema tree"* in [6], we briefly describe the algorithm of [6] (called *EM-algorithm).*

Example 1. *For a "binary and canonical hypergraph" H (viz. [6]), representing a CM,* EM-algorithm *derives an element tree T such that T is in XNF and every path of T reflects a sequence of some connected edges in H. For example, starting from the Department node of the ontology in Figure 3 the following element tree (omitting attributes) T is obtained: Department[(FacultyMember[(Hobby)*, (GradStudent[Program, (Hobby)*])*])*], where we use [ ] to indicate hierarchy and ( )* to indicate the multiple occurrences of a child element (or non-functional edges) in element trees.*

In essence, EM-algorithm *recursively constructs the element tree T as follows: it starts from a concept node N in CM, creates tree T rooted in a node R corresponding to N, and constructs the direct subtrees below R by following nodes and edges connected to N in CM. Finally, a largest hierarchical structure embedded within CM is identified and an edge of T reflects a semantic connection in the CM.* □

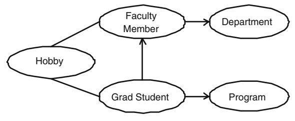

Fig. 3. Sample CM/ontology graph

A binary and canonical CM can naturally be viewed as an OWL ontology: concepts are classes, binary relationships are object properties, and attributes are datatype properties. So, given an XNF-compliant element tree T, we may assume that there is a *semantic tree S* embedded in an ontology graph such that S is isomorphic to T. If the correspondences between elements and classes were given, we should be able to identify S in terms of the ontology.

Example 2. *Suppose elements in the element tree T of Example 1 correspond to the classes (nodes) in Figure 3 by their names. Then we can recover the semantics* *of T recursively starting from the bottom, e.g., for the subtree GradStudent[Program, (Hobby)*], because the edge GradStudent $\Rightarrow$ Program is functional and GradStudent $\rightarrow$ Hobby is non-functional, and GradStudent is the root, we look for functional edges from GradStudent to Program and $1:N$ or $M:N$ edges from GradStudent to Hobby in the ontology graph. Likewise, we can recover the edges from $FacultyMember$ to GradStudent and $Hobby$. Finally, the $1:N$ edge between Department and $FacultyMember$ is recovered.* □

In an element tree T, attributes are the leaves of T and correspond to the datatype properties of classes in an ontology. There has been much research on schema matching tools [4,12,13] which focus on generating these kinds of correspondences automatically. Given the correspondences from XML attributes to datatype properties of an ontology, we expect to identify the root and the remaining nodes of the semantics tree S and connect them meaningfully.

Example 3. *Suppose the following correspondences:*
$\mathcal{X}$*:GradStudent.@ln*$\leftrightsquigarrow$$\mathcal{O}$*:GradStudent.lastname,*
$\mathcal{X}$*:GradStudent.@fn* $\leftrightsquigarrow$ $\mathcal{O}$*:GradStudent.firstname,*
$\mathcal{X}$*:GradStudent.Program.@pname*$\leftrightsquigarrow$$\mathcal{O}$*:Program.name,*
are for the element tree GradStudent(@ln, @fn)[Program(@pname)], where we use prefixes $\mathcal{X}$ and $\mathcal{O}$ to distinguish terms in the element tree and the ontology. Then we could identify the class $\mathcal{O}$:GradStudent as the root of the semantic tree and recover it as the edge $\mathcal{O}$:GradStudent `-->-` *$\mathcal{O}$:Program.* □

The *first principle* of our mapping construction algorithm is to identify the root of a semantic tree and to construct the tree by connecting the root to the rest of nodes in the ontology graph using edges having compatible cardinality constraints with edges in the element tree.

However, identifying the root of the semantic tree is the major obstacle. The following example illustrates the problem for an XML schema which is not XNF compliant. Such a schema can be easily encountered in reality.

Example 4. *for the element tree*
GradStudent[Name(@ln, @fn), Program(@pname)]
with the correspondences
$\mathcal{X}$*:GradStudent.Name.@ln*$\leftrightsquigarrow$$\mathcal{O}$*:GradStudent.lastname,*
$\mathcal{X}$*:GradStudent.Name.@fn* $\leftrightsquigarrow$$\mathcal{O}$*:GradStudent.firstname,*
$\mathcal{X}$*:GradStudent.Program.@pname*$\leftrightsquigarrow$$\mathcal{O}$*:Program.name,*
the element $\mathcal{X}$:Name corresponds to $\mathcal{O}$:GradStudent by its attributes and the element $\mathcal{X}$:Program corresponds to $\mathcal{O}$:Program. Further, both $\mathcal{X}$:Name and $\mathcal{X}$:Program occur once and are at the same level. Then the question is which one is the root of the semantic tree*? $\mathcal{O}$:GradStudent or $\mathcal{O}$:Program? Since the order of nodes on the same level of the element tree does not matter, both are potential roots. Therefore, the mapping algorithm should recover the functional edges from $\mathcal{O}$:GradStudent to $\mathcal{O}$:Program as well as from $\mathcal{O}$:Program to $\mathcal{O}$:GradStudent, if any.* □

This leads to the *second principle* of our algorithm: for each class C in the ontology graph such that C corresponds to a child element E of the root element

R in the element tree T and $R \Rightarrow E$ is functional, C is a potential root of the semantic tree S. Treating an attribute as a subtree, the mapping construction algorithm will recursively recover the semantic tree S in a bottom-up fashion.

Unfortunately, not every functional edge from a parent element to a child element represents a functional relationship. Specifically, some element tags are actually the collection tags containing a set of instances of the child elements. For example, for the element tree: *GradStudent*[*Name*(@*ln*, @*fn*), *Hobbies*[(*Hobby*(@*title*))*]] with the correspondences
$\mathcal{X}$:*GradStudent.Name.@ln* $\leftrightsquigarrow$ $\mathcal{O}$:*GradStudent.lastname*,
$\mathcal{X}$:*GradStudent.Name.@fn* $\leftrightsquigarrow$ $\mathcal{O}$:*GradStudent.firstname*,
$\mathcal{X}$:*GradStudent.Hobbies.Hobby.@title* $\leftrightsquigarrow$ $\mathcal{O}$:*Hobby.title*,
the element tag $\mathcal{X}$:*Hobbies* represents a collection of hobbies of a graduate student. Although the edge $\mathcal{X}$:*GradStudent* $\Rightarrow$ $\mathcal{X}$:*Hobbies* is functional, $\mathcal{X}$:*Hobbies* $\rightarrow$ $\mathcal{X}$:*Hobby* is non-functional. Therefore, when $\mathcal{O}$:*Hobby* is identified as the root of the semantic tree for the subtree *Hobbies*[(*Hobby*(@*title*))*], $\mathcal{O}$:*Hobby* should not be considered as a potential root of the semantic tree for the entire element tree. Eliminating classes corresponding to collection tags from the set of the potential roots is our *third principle*.

In most cases, we try to discover the semantic mapping between an XML schema and an ontology such that they were developed independently. In such cases, we may not be able to find an isomorphic semantic tree S embedded in the ontology graph, or we may find an isomorphic tree that is not the intended one, for a given element tree. For example, for the element tree *City*(@*cityName*)[*Country* (@*countryName*)] and a ontology with a path $\boxed{\texttt{City}}$ `-- locatedIn -->-` $\boxed{\texttt{State}}$ `-- locatedIn -->-` $\boxed{\texttt{Country}}$ (recall `-->-` indicates a functional property), the intended semantics is the path rather than a single edge. The *fourth principle* for discovering mappings is to find shortest paths in the ontology graph instead of single edges, where the semantics of the paths is consistent with the semantics of the edges in the element tree in terms of cardinality constraints.

Even though we could eliminate some collection tags from the set of potential roots to reduce the number of possible semantic trees, there are still too many possibilities if the ontology graph is large. In order to further restrict the set of potential roots, we can make use of `key` and `keyref` definitions in XML schemas.

Example 5. *For the element tree*
Article[*Title*(@*title*), *Publisher*(@*name*),
ContactAuthor(@*contact*), (*Author*(@*id*))*]
if the attribute @title is defined as the `key` *for Article, then we should only choose the class corresponding to @title as the root of the semantic tree, eliminating the classes corresponding to @name and @contact (picked by the second principle). Further, if @contact is defined as a* `keyref` *referencing some key, we also can eliminate the class corresponding to @contact.* □

So our *fifth principle* is to use `key` and `keyref` definitions to restrict the set of potential roots.

Reified Relationships. To represent n-ary relationships in OWL ontologies, one needs to use classes, called *reified relationship (classes)*. For example, an ontology may have class $\mathcal{O}$:*Presentation* connected with functional *roles* to classes $\mathcal{O}$:*Author*, $\mathcal{O}$:*Paper*, and $\mathcal{O}$:*Session*, indicating participants. It is desirable to recover reified relationships and their role connections from an XML schema. Suppose the element tree *Presentation[Presenter(@author), Paper(@title), Session(@eventId)]* represents the above ternary relationship. Then, in the ontology, the root of the semantic tree is the *reified relationship class* $\mathcal{O}$:*Presentation*, rather than any one of the three classes which are role fillers. The *sixth principle* then is to look for *reified relationships* for element trees with only functional edges from a parent to its children that correspond to separate classes[1].
ISA. In [6], ISA relationships are eliminated by collapsing superclasses into their subclasses, or vice versa. If a superclass is collapsed into subclasses, correspondences can be used to distinguish the nodes in the ontology. If subclasses are collapsed into their superclass, then we treat the ISA edges as special functional edges with cardinality constraints $0:1$ and $1:1$. The *last principle* is then to follow ISA edges whenever we need to construct a functional path[2].

4.2 Algorithm

First, to get a better sense of what we are aiming for, we present the $\mathsf{encodeTree}(S, L)$ procedure, which translates an ontology subtree S into a conjunctive formula, taking into account the correspondences L.

```
Function encodeTree(S, L)
Input subtree S of ontology graph, correspondences L from attributes of element
tree to datatype properties of class nodes in S.
Output variable name generated for root of S, and conjunctive formula for the
tree.
Steps:
 1. Suppose $N$ is the root of $S$, let $\Psi = \{\}$.
 2. If $N$ is an attribute node with label $f$, find $@d$ such that $L(\_, @d, \_, f, N) = true$, return $(@d, true)$.
 3. If $N$ is a class node with label $C$, then introduce new variable $Y$; add conjoint
    $C(Y)$ to $\Psi$; for each edge $p_i$ from $N$ to $N_i$:
    (a) let $S_i$ be the subtree rooted at $N_i$;
    (b) let $(v_i, \phi_i(Z_i))$=encodeTree$(S_i, L)$;
    (c) add conjunct $p_i(Y, v_i) \wedge \phi_i(Z_i)$ to $\Psi$;
 4. return $(Y, \Psi)$.
```

[1] If a parent functionally connects to only two children, then it may represent an M:N binary relationship. So recover it as well.

[2] Thus, ISA is taken care of in the forthcoming algorithm by proper treatment of functional path.

The following procedure constructTree(T, L) generates the subtree of the ontology graph for the element tree after appropriately replicating nodes[3] in the ontology graph.

```
Function constructTree$(T, L)$
Input an element tree $T$, an ontology graph, and correspondence $L$ from attributes in $T$ to datatype properties of class nodes in the ontology graph.
Output set of (subtree $S$, root $R$, $collectionTag$) triples, where $collectionTag$ is a boolean value indicating whether the root corresponds to a collection tag.
Steps:
 1. Suppose $N$ is the root of tree $T$.
 2. If $N$ is an attribute, then find $L(\_, N, \_, \_, R) = true$; return $(\{R\}, R, false)$.
    /*the base case for leaves.*/
 3. If $N$ is an element having $n$ edges $\{e_1, .., e_n\}$ pointing to $n$ nodes $\{N_1, .., N_n\}$,
    let $T_i$ be the subtree rooted at $N_i$,
    then compute $(S_i, R_i, collectionTag_i)=$ constructTree$(T_i, L)$ for $i = 1, .., n$;
    (a) If $n = 1$ and $e_1$ is non-functional, return $(S_1, R_1, true)$;/*N probably is a collection tag representing a set of instances each of which is an instance of the $N_1$ element.*/
    (b) Else if $n = 1$ and $e_1$ is functional return $(S_1, R_1, collectionTag_1)$.
    (c) Else if $R_1 = R_2 = ... = R_n$, then return (combine$(S_1, .., S_n)$, $R_1$, $false$)[4].
    (d) Else let $F = \{R_{j_1}, .., R_{j_m} |$ s.t. $e_{j_k}$ is functional and $collectionTag_{j_k} = false$ for $k = 1, .., m$, $j_k \in \{1, ..., n\}\}$ and $NF = \{R_{i_1}, .., R_{i_h} |$ s.t. $e_{i_k}$ is non-functional, or $e_{i_k}$ is functional and $collectionTag_{i_k} = true$ for $k = 1, .., h$, $i_k \in \{1, ..., n\}\}$, let $ans = \{\}$, /*separate nodes according to their connection types to $N$.*/
        i. Try to limit the number of nodes in $F$ by considering the following cases: 1) keep the nodes corresponding to key elements located on the highest level; 2) keep those nodes which are not corresponded by keyref elements.
       ii. If $NF = \emptyset$, find a reified relationship concept $R$ with $m$ roles $r_{j_1}, .., r_{j_m}$ pointing to nodes in $F$, let $S=$ combine$(\{r_{j_k}\}, \{S_{j_k}\})$ for $k = 1, .., m$; let $ans = ans \cup (S, R, false)$. If $R$ does not exist and $m = 2$, find a non-functional shortest path $p$ connecting the two nodes $R_{j_1}, R_{j_2}$ in $F$; let $S=$ combine$(p, S_{j_1}, S_{j_2})$; let $ans = ans \cup (S, R_{j_1}, false)$.
           /*N probably represents an n-ary relationship or many-many binary relationship (footnote 3 of the sixth principle.)*/
      iii. Else for each $R_{j_k} \in F$ $k = 1, .., m$, find a shortest functional path $p_{j_k}$ from $R_{j_k}$ to each $R_{j_t} \in F - R_{j_k}$ for $t = 1, .., k-1, k+1, .., m$; and find a shortest non-functional path $q_{i_r}$ from $R_{j_k}$ to each $R_{i_r} \in NF$ for $r = 1, .., h$; if $p_{j_k}$ and $q_{i_r}$ exist, let $S=$ combine$(\{p_{j_k}\}, \{q_{i_r}\}, \{S_1, .., S_n\})$; let $ans = ans \cup (S, R_{j_k}, false)$. /*pick an root and connect it to other nodes according to their connection types.*/
```

[3] Replications are needed when multiple attributes correspond to the same datatype property. See [2] for details.

[4] Function combine merges edges of trees into a larger tree.

iv. If $ans \neq \emptyset$, return ans; else find a minimum Steiner tree[5] S connecting $R_1, .., R_n$, return (S,R_1, $false$). */*the default action is to find a shortest Steiner tree.*/*

It is likely that the algorithm will return too many results. Therefore, at the final stage we set a threshold N_{thresh} for limiting the number of final results presented. In the following experimental section, this threshold was set to 10.

5 Mapping Construction Experiences

We have implemented the mapping algorithm and conducted a set of experiments to evaluate its effectiveness and usefulness.

Measures for mapping quality and accuracy. We first attempt to use the notions of *precision* and *recall* for the evaluation. Let R be the number of correct mapping formulas of an XML schema, let I be the number of correctly identified mapping formulas by the algorithm, and let P be the total number of mapping formulas returned. The two quantities are computed as: $precision = I/P$ and $recall = I/R$. Please note that for a single input element tree T, which has a single correct mapping formula, the algorithm either produces the formula or not. So the *recall* for T is either 0 or 1, but the *precision* may vary according to the number of output formulas. For measuring the overall quality of the mapping results, we computed the average precision and recall for all tested element trees of an XML schema.

However, precision and recall alone cannot tell us how useful the algorithm is to users. The purpose of our tool is to *assist* users in the process of constructing complex mappings, so that productivity is enhanced. Consider the case when only one semantic mapping is returned. Even if the tool did not find the exactly right one, it could still be useful if the formula is accurate enough so that some labor is saved. To try to measure this, we adopt the accuracy metric for schema matching [13]. Consider the mapping formula $\Phi(\overline{X}) \rightarrow \Psi(\overline{X}, \overline{Y})$ with the formula $\Phi(\overline{X})$ encoding an element tree. The formula $\Psi(\overline{X}, \overline{Y})$ encodes a semantic tree $S = (V, E)$ by using a set of unary predicates for nodes in V, a set of binary predicates for edges in E, and a set of variables, $\overline{Y}$, assigned to each node (there are predicates and variables for datatype properties as well). For a given element tree T, writing the complex mapping formula consists of identifying the semantic tree and encoding it into a conjunctive formula (which could be treated as a set of atomic predicates). Let $\Psi_1 = \{a_1(\overline{Z}_1), a_2(\overline{Z}_2), .., a_m(\overline{Z}_m)\}$ encode a tree S_1, let $\Psi_2 = \{b_1(\overline{Y}_1), b_2(\overline{Y}_2), .., b_n(\overline{Y}_n)\}$ encode a tree S_2. Let $D = \Psi_2 \backslash \Psi_1 = \{b_i(\overline{Y}_i)|$ s.t. for a given partial one-one function $f : \overline{Y} \rightarrow \overline{Z}$ representing the mapping from nodes of S_2 to nodes of S_1, $b_i(f(\overline{Y}_i)) \in \Psi_1\}$. One can easily identify the mapping $f : \overline{Y} \rightarrow \overline{Z}$ by comparing the two trees S_2 and S_1 (recall an ontology graph contains class nodes as well as attribute nodes representing datatype properties) so we consider that it comes for free. Let $c = |D|$. Suppose Ψ_1 be the correct

[5] A Steiner tree on $R_1, .., R_n$ is a spanning tree that may contain nodes other than $R_1, .., R_n$.

formula and Ψ_2 be the formula returned by the tool for an element tree. To reach the correct formula Ψ_1 from the formula Ψ_2, one needs to delete $n-c$ predicates from Ψ_2 and add $m-c$ predicates to Ψ_2. On the other hand, if the user creates the formula from scratch, m additions are needed. Let us assume that additions and deletions need the same amount of effort. However, browsing the ontology for correcting formula Ψ_2 to formula Ψ_1 is different from creating the formula Ψ_1 from scratch. So let α be a cost factor for browsing the ontology for correcting a formula, and let β be a factor for creating a formula. We define the accuracy or labor savings of the tool as $labor\,savings = 1 - \frac{\alpha[(n-c)+(m-c)]}{\beta m}$. Intuitively, $\alpha < \beta$, but for a worst-case bound let us assume $\alpha = \beta$ in this study. Notice that in a perfect situation, $m = n = c$ and $labor\,savings = 1$.

Schemas and ontologies. To evaluate the tool, we collected 9 XML schemas varying in size and nested structure. The 9 schemas come from 4 application domains, and 4 publicly available domain ontologies were obtained from the Web and literature. Table 1 shows the characteristics of the schemas and the ontologies; the column heads are self-explanatory. The *company* schema and ontology are obtained from [9] in order to test the principles of the mapping construction. The *conference* schema is obtained from [10]. *UT DB* is the schema used for describing the information of the database group in University of Toronto. *SigmodRecord* is the schema for SIGMOD record. The rest of the schemas are obtained from the *Clio* test suite (http://www.cs.toronto.edu/db/Clio). The KA ontology, CIA factbook, and the Bibliographic-Data are all available on the Web. We have published the schemas and ontologies on our website along with some sample mapping results at the following URL:
http://www.cs.toronto.edu/ ˜yuana/research /maponto/testData.html.

Experimental results. Our experiments are conducted on a Dell desktop with a 1.8GHZ Intel Pentium 4 CPU and 1G memory. The first observation is the efficiency. In terms of the execution times, we observed that the algorithm generated results on average in 1.4 seconds which is not significantly large, for our test data.

Table 1. Characteristics of Test XML Schemas and Ontologies

XML Schema	Max Depth (DFS) in Schema Graph	# Nodes in Schema Graph	# Attributes in Schema Graph	Ontology	# Nodes	# Links
Company	6	30	17	Company	18	27
Conference	5	21	12	KA	105	4396
UT DB	6	40	20	KA	105	4396
Mondial	6	214	93	CIA factbook	52	77
DBLP 1	3	132	63	Bibliographic	75	749
DBLP 2	5	29	11	Bibliographic	75	749
SigmodRecord	3	16	7	Bibliographic	75	749
Amalgam 1	3	117	101	Bibliographic	75	749
Amalgam 2	3	81	53	Bibliographic	75	749

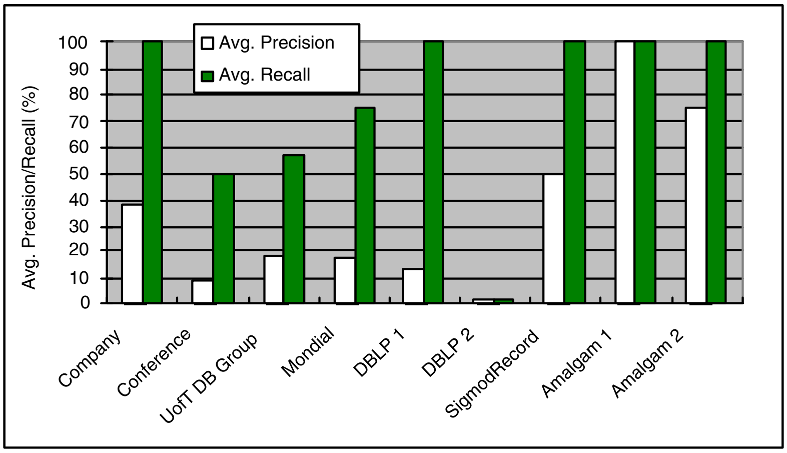

Fig. 4. Average Recall and Precision for 9 Mapping Cases

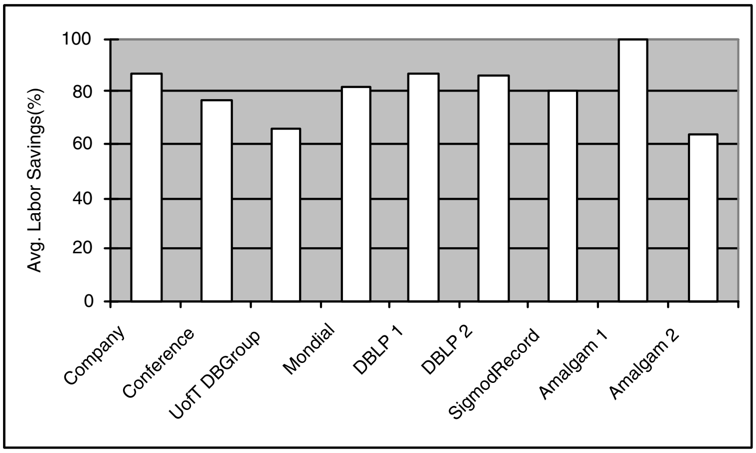

Fig. 5. Average Labor Savings for 9 Mapping Cases

Figure 4 shows the average precision and recall measures of the 9 mapping pairs. For each pair of schema and ontology, the average precision and recall are computed as follows. For the element trees extracted from the schema graph, a set of correct mapping formulas is manually created. We then apply the algorithm on the element trees and ontologies to generate a set of formulas. Next we examine each of the generated formulas to count how many are correct and compute the average precision and recall. The overall average precision is 35% and overall average recall is 75%. Notice that we have limited the number of formulas returned by the tool to 10.

Finally, we evaluate the usefulness of the tool. Figure 5 shows the average values of labor savings for the 9 mapping cases. For each mapping case, the average labor savings is computed as follows. Examine each incorrect formula returned by the algorithm and compute its labor saving value relative to the manually created one. Take the average value of the labor savings of all incorrect formulas. Note that even when the correct formula was identified by the algorithm, we still computed the labor savings for all incorrect ones to see how useful the tool is in case only one formula was returned. The overall average labor savings is over 80%, which is quite promising. Especially in view of the pessimistic assumption that $\alpha = \beta$ in the *labor savings* formula, we take this as evidence that the tool can greatly assist users in constructing complex mappings between XML schemas and ontologies with a proper schema matching tool as a front-end component.

6 Refining Mappings by Ontology Reasoning

Rich ontologies provide a new opportunity for eliminating "unreasonable" mappings. For example, if the ontology specifies that once a Person owns a CellPhone, they do not rent another one, then a candidate semantic formula $Person(X)$, $rents(X, Y)$, $Cell(Y)$, $owns(X, Z)$, $Cell(Z)$ can be eliminated [6], since no objects X can satisfy it. When ontologies, including constraints such as the one about renting/owning, are expressed in OWL, one can actually use OWL reasoning to detect inconsistent semantics by converting semantic trees into OWL concepts, and then testing them for incoherence with respect to the ontology. For example, the above formula can be translated, using an algorithm resembling encodeTree(S,L), into the OWL concept whose abstract syntax is *intersectionOf(Person, restriction(rents someValuesFrom(Cell)), restriction(owns someValues-From(Cell)))*. The ontologies we have found so far are unfortunately not sufficiently rich to demonstrate the usefulness of this idea.

7 Conclusions

In this paper, we have motivated and defined the problem of constructing complex semantic mappings from XML data to ontologies, given a set of simple correspondences from XML attributes to OWL datatype properties. The problem is well-motivated by the needs to annotate XML data in terms of ontologies, to translate XML data into ontologies, and to integrate heterogeneous XML data on the semantic web. We have proposed a tool for semi-automatically constructing complex mappings for users, and we evaluated the tool on a variety of real XML schemas and ontologies. Our experimental results suggest that quite significant savings in human work could be achieved by the use of our tool.

Integrating our tool with schema matching tools which automatically generate schema and ontology element correspondences is an open problem to address in the future. We also plan to develop filters for mappings by making use of instance data to assist users in choosing the correct mapping among a list of possible candidates.

[6] Probably some other relationship than $rents(X, Y)$ needs to be used.

Acknowledgments. We are grateful to anonymous reviewers for offering valuable comments, corrections, and suggestions for improvement.

References

1. B. Amann, C. Beeri, I. Fundulaki and M. Scholl. Ontology-Based Integration of XML Web Resources. In ISWC'02.
2. Y. An, A. Borgida, and J. Mylopoulos. Inferring Complex Semantic Mappings between Relational Tables and Ontologies from Simple Correspondences. In ODBASE'05.
3. M. Arenas and L. Libkin. XML Data Exchange: Consistency and Query Answering. In PODS'05, Baltimore, USA.
4. R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering Complex Semantic Matches between Database Schemas. In SIGMOD'04.
5. C. Delobel, C. Reynaud, and M. Rousset, J. Sirot, and D. Vodislav. Semantic Integration in Xyleme: A Uniform Tree-Based Approach. Data and Knowledge Engineering 44(2003), 267-298, 2002.
6. D. W. Embley and W. Y. Mok. Developing XML Documents with Guaranteed "Good" Properties. In ER'01.
7. A. Halevy, Z. Ives, P. Mork, and I. Tatarinov. Piazza: Data Management Infrastructure for Semantic Web Applications. In WWW'03.
8. M. Jensen, T. Moller and T. Pedersen. Converting XML DTDs to UML Diagrams for Conceptual Data Integration. Data and Knowledge Engineering 44(2003), 323-346, 2002.
9. C. Kleiner and U. W. Lipeck. Automatic Generation of XML DTDs from Conceptual Database Schemas. GI Jahrestagung (1), 2001.
10. D. Lee and W. W. Chu. Constraint-Preserving Transformation from XML Document Type Definition to Relational Schema. In ER'00.
11. L. V.S. Lakshmanan and F. Sadri. Interoperability on XML Data. In ISWC'03.
12. J. Madhavan, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In VLDB'01.
13. S. Melnik, H. Garcia-Molina and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In ICDE'02.
14. R. Miller, L. Haas, and M. Hernandez. Schema Mapping as Query Discovery. In VLDB'01.
15. L. Popa, Y. Velegrakis, R. Miller, M. Hernandez, and R. Fagin. Translating Web Data. In VLDB'02.
16. J. Shanmugasundaram et al. Relational Database for Querying XML Documents: Limitations and Opportunities. In VLDB'99.