Induction of Integrated View for XML Data with Heterogeneous DTDs

Euna Jeong
Computer Science and Information Engineering
National Taiwan University
#1 Roosevelt Rd. Sec. 4, Taipei, Taiwan
eajeong@agents.csie.ntu.edu.tw

Chun-Nan Hsu
Institute of Information Science
Academia Sinica
Nankang, Taipei, Taiwan
chunnan@iis.sinica.edu.tw

ABSTRACT

This paper proposes a novel approach to integrating heterogeneous XML DTDs. With this approach, an information agent can be easily extended to integrate heterogeneous XML-based contents and perform federated search. Based on a tree grammar inference technique, this approach derives an integrated view of XML DTDs in an information integration framework. The derivation takes advantages of naming and structural similarities among DTDs in similar domains. The complete approach consists of three main steps. (1) *DTD clustering* clusters DTDs in similar domains into classes. (2) *Schema learning* applies a tree grammar inference technique to generate a set of tree grammar rules from the DTDs in a class from the previous step. (3) *Minimization* optimizes the rules generated in the previous step and transforms them into an integrated view. We have implemented the proposed approach into a system called *DEEP* and tested the system on artificial and real domains. The experimental results reveal that this system can effectively and efficiently integrate radically different DTDs.

Keywords

XML DTD, Semistructured data, Federated search, Distributed databases, Intelligent Agent, Mark-up schemes

1. INTRODUCTION

Software agents [7, 13] and integration systems of heterogeneous databases [3, 11, 12, 6] are widely studied and developed to allow users to find, collect, filter and manage information sources spread on the Internet. The design concerns of these systems vary for different domains, but all share a common need for a layer of an *integrated view* and *source descriptions* to seamlessly integrate heterogeneous information sources. The integrated view must be designed for each application domain. The source descriptions are needed to map source schemas to the integrated view. However, previous work in information integration requires both of them be constructed manually in a laborious and time-consuming manner. Constructing integrated view becomes one of the major bottlenecks for building large-scaled information integration agents(IIAs).

The approach presented in this paper is based on the framework of previous work in information integration. In particular, this approach addresses the problem of automatic derivation of the integrated view for XML DTDs(Document Type Definition) [1]. Although XML is becoming an industrial standard for exchanging data on the Internet, when the maintenance of the information sources is independent of the integrator, it is difficult and sometimes impossible to have such a common DTD. This paper reports our preliminary result for resolving this problem.

1.1 XML Information Integration

Figure 1 shows the diagram of an XML information integration agent. When the user submits a request to the agent through a user interface, the request will be translated into an XML-QL [4] query based on an integrated view, which is a query template that helps formulate queries. Given the translated query, the *query decomposer* transforms the query into a set of subqueries against each integrated XML information source based on source descriptions. Finally, *query executor* issues the subqueries to each information source, combines the answers, and returns the result as an XML document to the user.

This paper describes how to automatically derive the integrated view and source descriptions by a *view inference* system in Figure 1. The derivation is conducted offline before the IIA can provide service. The view inference system is to automatically discover the association between closely related DTDs, identify elements with similar underlying semantics, and generate an *integrated view* that covers these semantically similar elements.

EXAMPLE 1. *Suppose we have four DTDs as shown in Table 1. They are extracted from published papers and documents: (a) and (c) from [4], (b) from [8], and (d) from [2]. This set of DTDs will serve as the example input and will be used to explain our view integration approach throughout the paper. Table 2 shows an integrated view that may be derived from two DTDs in Table 1(a) and (b). Reformulated subqueries for these two DTDs are shown in Table 3. In this case, we assume that there are two related XML documents,* `COOKBOOK.xml` *with* `COOKBOOK` *DTD and* `BIB.xml` *with* `BIB` *DTD.* □

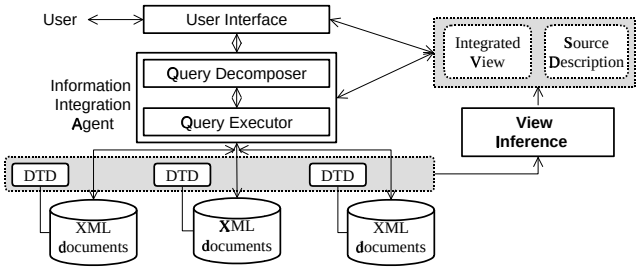


Figure 1: XML information integration agent

```
  (a) COOKBOOK DTD
 1 ⟨!ELEMENT cookbook (title, author+, year, isbn, publisher)⟩
 2 ⟨!ELEMENT author (authorname)⟩
 3 ⟨!ELEMENT authorname (firstname, lastname)⟩
 4 ⟨!ELEMENT publisher (name, address)⟩
  (b) BIB DTD
 5 ⟨!ELEMENT bib (title, author+, publisher, price)⟩
 6 ⟨!ATTLIST bib year CDATA #REQUIRED⟩
 7 ⟨!ELEMENT author (last, first)⟩
 8 ⟨!ELEMENT publisher (name, email)⟩
  (c) MOVIE DTD
 9 ⟨!ELEMENT movie (title, year, director*, featuring, genre)⟩
10 ⟨!ELEMENT featuring (credit_actor*, actors*)⟩
  (d) LIST DTD
11 ⟨!ELEMENT list (movie*)⟩
12 ⟨!ELEMENT movie (title, year, directed_by, genres, featuring)⟩
13 ⟨!ELEMENT directed_by (director*)⟩
14 ⟨!ELEMENT genres (genre*)⟩
15 ⟨!ELEMENT actor (firstname, lastname)⟩
```

Table 1: Example DTDs

1.2 Illustrative Examples

XML data is an instance of semistructured data. In XML, each *element* represents a logical component of a document. Each element has a *tag* to indicate its semantics. Elements can be atomic (i.e., character strings) or contain other subelements, which allow us to encode structural semantics in an application domain. The definitions of the role of tags are formally given in a DTD. A DTD establishes a set of constraints for XML documents. XML with a DTD is self-descriptive and provides a semistructured data model.

Distinct DTDs in a similar application domain may use different labels for the same concept, or use the same label for different concepts. The same concepts can also be defined with different structures. In spite of the possible diversity, if the underlying concepts of a set of DTDs are closely related, though they are created by different authors, they will reveal structural and naming similarities.

Table 4 shows several example cases that our approach can handle. The elements of pairs of input DTDs are shown in the second column and their corresponding subsumption relationships are illustrated as trees in the third column. For each case, the view inference system can integrate the pair of trees into a single tree as shown in the fourth column.

We will describe how the view inference system resolves these cases in the following Sections. The view inference system consists of three major components shown in Figure 2. A brief description of each module is as follows:

DTD cluster takes a collection of source DTDs as input and clusters similar DTDs into DTD classes. We merge similar element types as a preprocessing step to speed up the clustering. The DTD cluster uses hierarchical agglomerative clustering method [16] to cluster DTDs. We define a generic tree matching method [14] to compute the distance between DTDs for clustering.

```
 1  WHERE <cookbook|bib>
 2          <title>$title</>
 3          <*>
 4              <author|authorname>
 5                  <firstname|first>$first</>
 6                  <lastname|last>$last</>
 7              </>
 8          </>
 9          <publisher>
10              <address>$address</>
11              <name>$name</>
12              <email>$email</>
13          </>
14          <year>$year</>
15          <isbn>$isbn</>
16          <price>$price</>
17      </>
18 CONSTRUCT result statements
```

Table 2: Integrated view for BOOK domain

```
  (a) For COOKBOOK.xml                (b) For BIB.xml
 1 WHERE                              WHERE
 2   <cookbook>                         <bib year = $year>
 3     <title>$title</>                   <title>$title</>
 4     <author>                           <*>
 5       <authorname>                       <author>
 6         <firstname>$first</>               <first>$first</>
 7         <lastname>$last</>                 <last>$last</>
 8       </>                                </>
 9     </>                                </>
10     <publisher>                        <publisher>
11       <address>$address</>               <email>$email</>
12       <name>$name </>                    <name>$name </>
13     </>                                </>
14     <year>$year</>                     <price>$price</>
15     <isbn>$isbn</>                   </> IN "BIB.xml"
16  </> IN "COOKBOOK.xml"               CONSTRUCT result statements
17 CONSTRUCT result statements
```

Table 3: Reformulated queries for BOOK domain

Schema learner infers the general rules describing source DTDs in each DTD class. Our approach is based on a tree grammar inference technique [9], which takes a finite sample of trees as input and induces a tree grammar that describes the sample trees. In our approach, the input trees correspond to XML DTDs and the tree grammar corresponds to an integrated schema.

Minimizer optimizes the learned tree grammar rules. The learned rules are adjusted to fit the characteristics of DTD and transformed into the integrated view as well as the source descriptions to be used in IIAs.

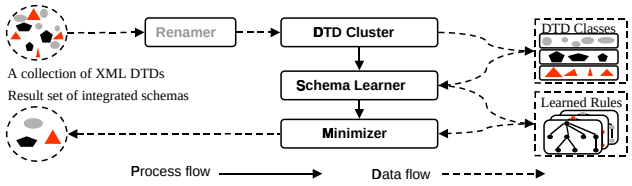


Figure 2: Diagram of the view inference system

The reminder of the paper is organized as follows. Section 2 defines the problem of view inference for XML DTDs.

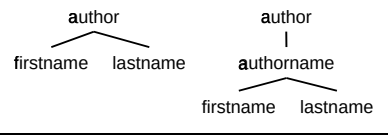
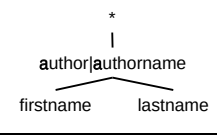
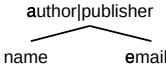
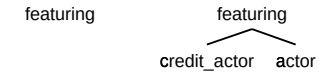
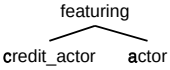
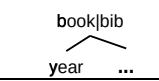
	Example DTDs	Input	Output
1	eg1:<ELEMENT author (firstname,lastname)> eg2:<ELEMENT author authorname> <ELEMENT authorname (firstname,lastname)>		
	The same concept may have the same element name, but has different hierarchical structures.		
2	eg1:<ELEMENT publisher (name,email)> eg2:<ELEMENT author (name,email)>		
	The two different concepts have the same sub-elements list.		
3	eg1:<ELEMENT featuring #PCDATA> eg2:<ELEMENT featuring (credit_actor,actor)>		
	The same concept may have the same element name, but has different sub-elements list in different DTDs.		
4	eg1:<ELEMENT book (year, ...)> eg2:<ATTLIST bib year #CDATA>		
	The same concept can be defined as an element in one DTD, and an attribute in another DTD. Because attribute names are handled the same as element names, the tree structures are equal. The difference will be described in Section 2.		

Table 4: Several example cases that our approach can handle: *Example DTDs:* **pairs of input DTDs;** *Input:* **tree structure of DTDs;** *Output:* **tree structure of integrated view**

Section 3 describes the details of our view inference approach. The approach has been implemented into a system called DEEP and tested in several domains. We report the experimental results in Section 4. Section 5 reviews related work. Finally, Section 6 draws conclusions.

2. A MODEL OF XML VIEW INFERENCE

2.1 Types and DTD Class

We model a DTD as a labeled, directed tree. The nodes in the tree represent objects and are labeled with an element or attribute name. We assume that each tree has a distinguished root object, and all objects are reachable from the root. Each node in the tree has its own type. The *type* of an object is defined by its label and its immediately adjacent child nodes (objects). XML attributes are treated in the same way as element tags.

Each type is denoted by t_i, where i is the type id. The type id of all leaf nodes (i.e., #PCDATA type) is 0, which implies that the value of atomic objects is ignored in this paper. Each type has a type definition of the form [*label*: *Type(label)*], where *label* is a regular expression over a finite set $\mathcal{N}$ of names and *Type(label)* is either #PCDATA for leaf nodes or a regular expression over $\mathcal{N}$ with type id as the subscription.

A *DTD schema* consists of a sequence of type definitions.

EXAMPLE 2. *Given the set $\mathcal{D}$ of source DTDs in Table 1, the following type set Γ can be constructed. The underlined labels, such as $\underline{year_0}$ of t_5, correspond to XML attributes. In this example, $t_1 - t_4$ comprise type definitions for* COOKBOOK *DTD, $t_5 - t_7$ for* BIB *DTD, $t_8 - t_9$ for* MOVIE *DTD, and the others for* LIST *DTD. For brevity, the #PCDATA type elements are abbreviated.* □

$t_1 = [cookbook: (title_0, (author_2)+, year_0, isbn_0, publisher_4)];$
$t_2 = [author: (authorname_3)];$
$t_3 = [authorname: (firstname_0, lastname_0)];$
$t_4 = [publisher: (name_0, address_0)];$
$t_5 = [bib: (title_0, (author_6)+, publisher_7, price_0, \underline{year_0})];$
$t_6 = [author: (first_0, last_0)];$
$t_7 = [publisher: (name_0, email_0)];$
$t_8 = [movie: (title_0, year_0, (director_0)*, featuring_9, genre_0)];$
$t_9 = [featuring: ((credit\_actor_0)*, (actor_{14})*)];$
$t_{10} = [list: (movie_{11})*];$
$t_{11} = [movie: (title_0, year_0, directed\_by_{12}, genres_{13}, featuring_0)];$
$t_{12} = [directed\_by: (director_0)*];$
$t_{13} = [genres: (genre_0)*];$
$t_{14} = [actor: (firstname_0, lastname_0)];$

A *DTD class* is a set of similar DTD schemas. Since we make no assumption that the input DTDs must describe the same domain, the input DTDs may describe drastically different domains. Therefore, DTDs need to be clustered into classes of similar domains so that it is meaningful for the system to derive an integrated view. This task is the goal of the DTD clustering in our approach. Given the DTDs in Table 1, the output should be two classes. One class contains COOKBOOK and BIB DTDs, and the other contains MOVIE and LIST DTDs.

2.2 From Tree Grammar to Integrated Schema

Given a DTD class, *schema learner* generates a tree automaton that describes DTDs (as trees) in the input DTD class. The corresponding tree grammar of the tree automaton generates an infinite language that expresses all input trees in the DTD class. The learned rules will then be optimized by *minimizer*, which will insert the wildcard symbol \$ to the tree grammar for common labels and subtrees.

A *tree grammar* $M = \langle \mathcal{Q}, \mathcal{N}, \delta, \mathcal{F} \rangle$ is a quadruple: a set $\mathcal{Q}$ of states, a set $\mathcal{N}$ of labels, the state-transition function δ, and a set $\mathcal{F}$ of final states, $\mathcal{F} \subseteq \mathcal{Q}$.

Since M reduces frontier nodes first and the root node last, it is also known as a *frontier-to-root automaton*. We define δ, which operates on trees of depth 0 or 1 as follows:

- a tree which consists of single node labeled by a (i.e., tree primitive) belongs to a state q, denoted as $\delta_a = q$.
- a tree with m subtrees $q_1, q_2, \ldots, q_m$ and the root labeled by a belongs to a state q, denoted as $\delta_a(q_1, q_2, \ldots, q_m) = q$, where a is a regular expression over $\mathcal{N} \cup \{\$\}$ and

$q_1, q_2, \ldots, q_m$ are also regular expressions over each subtree. The wildcard \$ matches any label.

For example, a state transition function $\delta_{\texttt{firstname}} = q_1$ generates a single node tree with label `firstname` and belongs to state q_1. If a tree has a root node labeled by `author` and two child subtrees, q_1 and q_2, and it belongs to state q_3, then it is represented as $\delta_{\texttt{author}}(q_1, q_2) = q_3$.

An *integrated schema* is a DTD schema generated from optimized state-transition functions of a tree grammar. We define a mapping function θ from a state-transition function to a type definition in a DTD schema as follows:

- if $\delta_a = q$ is a function for starting states, then $\theta(\delta_a = q)$ maps to t_0 type with character value.
- if $\delta_a(q_1, q_2, \ldots, q_m) = q$ is a function for internal states, then $\theta(\delta_a(q_1, q_2, \ldots, q_m) = q)$ is a type t whose *label* is a and $Type(a)$ is a regular expression over labels of q_i with type id of q_i as the subscription, where $1 \leq i \leq m$.

2.3 Integrated View and Source Description

Before we proceed to describe the integrated view, we provide formal criteria against which the integrated schema should be evaluated. The first one, called *coverage*, guarantees that the integrated schema derives all DTD schemas in the input DTD class.

DEFINITION 1. *Given a regular expression R over some alphabet Σ, let $L(R)$ denote the regular language defined by R. We assume that $L(\#PCDATA)$ is a subset of any regular language. Let $D_c = \{\ldots, t_c, \ldots\}$ be a DTD schema in the DTD class C, $D_s = \{\ldots, t_s, \ldots\}$ be the integrated schema for C, and l_c and l_s are labels of t_c and t_s, respectively. We say that D_s covers C if there exists a mapping σ from types of each D_c in C to types in D_s such that:*

- *if $L(l_c) \subseteq L(l_s)$ and $L(Type(l_c)) \subseteq L(Type(l_s))$, then $\sigma(t_c) = t_s$.*
- *if all types in $\sigma(D_c) = \{.., \sigma(t_c), ..\}$ can construct a DTD schema.* □

Secondly, we require that the integrated schema be *compact*. Intuitively, the integrated schema should be the smallest type set that covers the input DTD class so that similar types in different DTDs should be mapped to the same type in the integrated schema.

DEFINITION 2. *An integrated schema D_s is more compact than an integrated schema D'_s if D_s covers given DTD class which is covered by D'_s and has a smaller type set than D'_s. An integrated schema D_s is the most compact one for a given DTD class if there is no integrated schema D'_s more compact than D_s.* □

An *integrated view* is an XML-QL query template transformed from the most compact integrated schema. Different orderings of the body statements are considered equivalent. We define a mapping ν from a type in the integrated schema to a body statement as follows:

- if $t = [label : (t_1, \ldots, t_m)]$ is a type with m sub-elements, then $\nu(t)$ maps to "$\langle label \rangle p_1 \ldots p_m \langle / \rangle$" where $p_1, \ldots, p_m$ are statements of types $t_1, \ldots, t_m$, resp.
- if $t = [label : (\#PCDATA)]$ is a type with no sub-element, then $\nu(t)$ maps to "$\langle label \rangle$ \$var $\langle / \rangle$".

In the case of complex element types, the type of the label (i.e., $Type(label)$) is mapped to a sequence of statements. But in the case of atomic element types, it is mapped to a variable. If the label is a wildcard ("\$"), we denote the statement as "$\langle * \rangle$", which means it matches *any* element. The result of the query is defined by the *result* statements, following the specification of XML-QL. Note that variables in XML-QL are preceded by "\$" symbol.

Given a query based on the integrated view, the system must translate it into subqueries against the related source DTDs with assigned URLs. The *source description* for each DTD is generated from types in $\sigma^{-1}(D_c)$ (the inverse function of σ) in Definition 1. The mapping is the same as ν from a type in each DTD schema to a body statement with the following additional condition:

- if $t = [label : (t_1, \ldots, t_m)]$ is a type with m sub-elements and sub-elements $t_{i+1}, \ldots, t_m$, $0 < i+1 \leq m$, are types corresponding to XML attributes, then $\nu(t)$ maps to "$\langle label\, p_{i+1} \ldots p_m \rangle p_1 \ldots p_i \langle / \rangle$", where $p_{i+1}, \ldots, p_m$ are statements of the form: "$label_p$ = \$var" and $label_p$ is the label of corresponding type.

For example, $t_5 = [bib : (title_0, year_0)]$ will be transformed to a pattern "$\langle bib\ year=\ \$year \rangle \langle title \rangle$ \$title $\langle / \rangle \langle / \rangle$".

Finally, we give a model of XML view inference problem. *View inference* is a function from a set of DTD schemas(in a DTD class) to an integrated view derived from the most compact integrated schema that covers the given DTD schemas, and source descriptions for the given DTD schemas.

3. VIEW INFERENCE SYSTEM

In this section, we describe the details of our approach to XML view inference in the data-flow order.

3.1 Renamer

Renamer as a preprocessing step is an optional module that requires human intervention. The internal nodes in XML DTDs offer both naming and structural hints for the system to associate related elements in different DTDs, while leaf nodes offer very limited information to the system. Renamer module is designed to allow human users to provide additional hints for the system to associate related leaf nodes. In the case of leaf nodes, the element name can be renamed manually to another internal/leaf element name in different DTDs so that they will be considered to share the same underlying concept. For example, in Example 2, element name `first` in t_6 may be changed to `firstname`.

3.2 DTD CLUSTER

This module contains three steps to complete clustering a collection of source DTDs into several DTD classes. As the first step, we merge element types. The purpose is to reduce the number of element types as well as the distance between DTD trees, so that DTDs of similar domains can have a better chance to be clustered together. We also describe the method of computing distance between DTD trees and clustering method in detail.

3.2.1 Type Merging

The type merging method is a preprocessing step before clustering. First, we consider types that have the same label. If t_i and t_j in two different DTDs have the same label, then we compare their sub-element lists. Let Sub_i/Sub_j be the set of t_i/t_j's sub-elements, respectively. If Sub_i and Sub_j have common elements, then these two types are merged and the result type definition consists of the common label and the

union of Sub_i and Sub_j. If one of two types is an atomic type, it will be merged to another. Case 3 in Table 4 is resolved in this step. Occurrence indicators (?, *, +) in type definitions are merged according to the precedence order: ? < * < +. For example, `author+` and `author*` are merged to `author*`.

EXAMPLE 3. *Given the type set* Γ *in Example 2, the merged type set* Γ^m *is generated as follows:*

$s_1 =$ [*cookbook* : (*title*$_0$, (*author*$_2$)+, *year*$_0$, *isbn*$_0$, *publisher*$_4$)];
$s_2 =$ [*author* : (*authorname*$_3$)];
$s_3 =$ [*authorname* : (*firstname*$_0$, *lastname*$_0$)];
$s_4 =$ [*publisher* : (*name*$_0$, (*email*$_0$)?, (*address*$_0$)?)];
$s_5 =$ [*bib* : (*title*$_0$, (*author*$_6$)+, *publisher*$_4$, *price*$_0$, *year*$_0$)];
$s_6 =$ [*author* : (*first*$_0$, *last*$_0$)];
$s_7 =$ [*movie* : (*title*$_0$, *year*$_0$, (*director*$_0$)*, *featuring*$_8$, (*genre*$_0$)?, (*directed_by*$_{10}$)?, (*genres*$_{11}$)?)];
$s_8 =$ [*featuring* : ((*credit_actor*$_0$)*, (*actor*$_{12}$)*)];
$s_9 =$ [*list* : (*movie*$_7$)*];
$s_{10} =$ [*directed_by* : (*director*$_0$)*];
$s_{11} =$ [*genres* : (*genre*$_0$)*];
$s_{12} =$ [*actor* : (*firstname*$_0$, *lastname*$_0$)];

Types t_8 *and* t_{11} *in Example 2 are merged to new type* s_7 *and types* t_4 *and* t_7 *to* s_4. *An atomic type* [*featuring* : #*PCDATA*] *is merged to* s_8. *Based on the merged types, we modify the input DTD schemas. Let* Γ^m *be the resulting merged type set. For each root type (root element) in* Γ^m, *the set of all reachable types will constitute the modified DTD schema. Continuously in Table 1, all DTD schemas are redefined according to the merged type set* Γ^m. *For example,* `MOVIE` *DTD schema will be redefined with five types,* s_7, s_8, s_{10}, s_{11}, *and* s_{12}. □

3.2.2 Tree Matrix

In order to cluster a collection of source DTDs, a distance measure quantifying the degree of association between DTDs is necessary. We use a generic matching method between two trees proposed by Lu [14] as the distance measure. Basically, the idea is to compute the distance by calculating the minimum number of modifications required to transform the input tree into a reference tree. We extend Lu's algorithm to compute the distance between two labeled trees.

There are five types of operations, (1)parent-child splitting, (2)sibling splitting, (3)parent-child merging, (4)sibling merging, and (5)substituting, as illustrated in Figure 3.

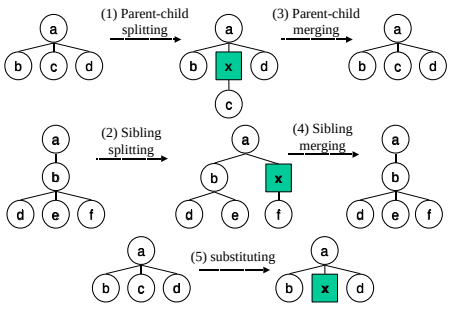


Figure 3: The five types of operations on tree nodes

To explain the algorithm clearly, we need to introduce some definitions and terminologies as follows (following [14]):

- All nodes in a tree T are labeled with postfix-ordered.
- a/T represents the subtree of a tree T at node a.
- $\tau_T(a, b)$ represents a region in a tree T, where $a \leq b$, which is a set of nodes of T such that x is a node in $\tau_T(a, b)$ if $a \leq x \leq b$.
- The n regions in Y, $\tau_Y(p_i, r_i)$, for $i = 1, \ldots, n$ are said to be *consistent* if the following holds true:
  1. $\tau_Y(p_i, r_i)$ and $\tau_Y(p_j, r_j)$ do not overlap for $1 \leq i, j \leq n, r_i < p_j$.
  2. Any node in $\tau_Y(p_i, r_i)$ does not have predecessor-descendant relation with any node in $\tau_Y(p_j, r_j)$, $i \neq j$.

The tree matching algorithm performs the operations in an efficient way by a *divide-and-conquer* strategy. In "divide", the matching between a subtree a/X and a region of Y is reduced to the problem of matching the children of the subtree a/X to each one of the regions of Y such that they are consistent. This process is illustrated in Figure 4 where the n regions are consistent in the tree Y.

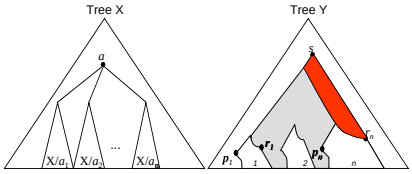


Figure 4: Tree matching method

With all the possible matchings between subtrees a/X and regions of Y identified, the last problem is to find the minimum subtree which covers all the n regions in Y. The "conquer" part resolves this problem. First, let s be the nearest common predecessor for the n regions. The second step is to compute the minimum number of operations to match a/X into the subtree s/T in Y. The distance for $\tau_Y(p_1, r_i)$, $i = 1, \ldots, n$ can now be computed according to the following equation:

$$d = \sum_{i=1}^{n} d_i + \left\{ |\tau_Y(p_1, r_n)| - \sum_{i=1}^{n} |\tau_Y(p_i, r_i)| \right\} + (s - r_n + 1).$$

In this equation, the first term is the total distance of mapping from n subtrees of a/X to n regions $\tau_Y(p_i, r_i)$, $i = 1, \ldots, n$. The second term is the cost of merging needed to remove the nodes not belonging to the n regions in $\tau(p_1, r_n)$ (the pale gray part in Figure 4). Finally, the third term is the number of merging operations used to remove nodes from r_n to s excluding r_n(the dark gray part in Figure 4).

3.2.3 Hierarchical Clustering Method

We employ a hierarchical clustering method [16], which is used widely in information retrieval. The basic idea is: initially start with a separate class for each DTD; successively merge the classes closest to one another, until the number of classes is sufficiently small. We use the average distance for computing the distance between classes.

The hierarchical clustering method computes a hierarchy of classes whose leaves are individual DTDs and internal nodes corresponding to classes formed by merging classes from lower levels.

3.3 Schema Learner

Our technique for generating integrated schema is based on tree grammar inference. *Grammatical inference* is the task to induce hidden grammatical rules from a set of examples. In a tree grammar, its terminal alphabet is composed of a set of primitive trees, then the induced rules will produce trees by assembling the primitive elements. The

problem of deriving an integrated schema from similar DTD schemas is reduced to this problem. We first describes the tree grammar inference framework. We then show how to treat integrated schema generation as an inference task.

We adopt the k-follower method [9], which applies a simple heuristic of state-merging process. We say that two states of a finite automaton whose last k words match have a tail in common. The two states are k-equivalent where k is a nonnegative integer.

Some additional definitions and terminologies for trees are given as follows (following [9]):

- $T(a \Leftarrow U)$ is the replacement of the subtree of T at a with a subtree U.
- $Depth_T(a)$ is the number of nodes on the path from the root of T to a, excluding a.

DEFINITION 3. *Let $\mathcal{S}$ be a given finite set of trees, $\mathcal{S}_{sub}$ be the set of all subtrees of the member trees in $\mathcal{S}$, and $\hat{\mathcal{S}}$ be a union of $\mathcal{S}$ and $\mathcal{S}_{sub}$. Also, let k be a nonnegative integer and "\$" be a special character not in the set of node labels in $\mathcal{S}$. The k-follower $H^k_{\mathcal{S}}(T)$ of a tree T with respect to $\mathcal{S}$ is defined by*

$$H^k_{\mathcal{S}}(T) = \{U(b \Leftarrow \$)\}$$

where tree U and node b satisfy the following conditions:

- $U \in \hat{\mathcal{S}}$ *and* $b/U = T$;
- *If there exists a tree $U \in \mathcal{S}$, then $Depth_U(b) \leq k$, or if there exists a tree $U \in \mathcal{S}_{sub}$, then $Depth_U(b) = k$.* □

Hence, the *k-follower* of tree T is the trees that are the replacements of T in each tree U in $\hat{\mathcal{S}}$ with \$.

EXAMPLE 4. *Given a set $\mathcal{S}$ of two trees, let T be a single node tree labeled fn, the* k-follower *of T is shown in Figure 5. The label author is abbreviated to a, authorname to an, firstname to fn, and lastname to ln.* □

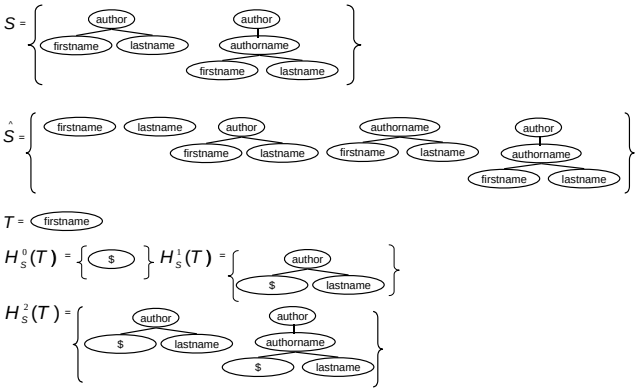


Figure 5: Example of k-follower set

We define the equivalence relation between trees based on the concept of k-follower.

DEFINITION 4. *Let R^k be an equivalence relation on trees in $\mathcal{S}$. $(T, U) \in R^k$ iff $H^k_{\mathcal{S}}(T) = H^k_{\mathcal{S}}(U)$ for $k \geq 0$.* □

Given a finite set $\mathcal{S}$ of trees and any nonnegative integer k, the tree grammar inference algorithm generates a nondeterministic tree automaton. At first, the automaton contains one state for each subtree of $\mathcal{S}$. The initial equivalence relation consists of the distinct subtrees in $\mathcal{S}$. The equivalence classes are induced from the equivalence relation and defined as follows: $[T]_{R^k} = \{U \in \hat{\mathcal{S}} | (T, U) \in R^k\}$. The equivalence classes will become the states of a nondeterministic tree automaton M. If subtrees of $\mathcal{S}$ not belonging to the same equivalent class have the same set of *k-follower*, then their classes are merged to a new one. The set of final states $\mathcal{F}$ is a set of equivalence relations whose k-follower has \$ as an element, i.e., $\mathcal{F} = \{[T]_{R^k} | \$ \in H^k_{\mathcal{S}}(T)\}$. Formally, our tree grammar inference algorithm is given as follows:

ALGORITHM 1. *Given a set $\mathcal{S}$ of source DTDs,*

Step 1. Generate the set $\hat{\mathcal{S}}$ and initialize k to 0.

Step 2. For each subtree T in $\hat{\mathcal{S}}$, generate the k-follower with respect to the set $\mathcal{S}$.

Step 3. If $H^k_{\mathcal{S}}(T) = H^k_{\mathcal{S}}(U)$, then the states of the automaton corresponds to the same equivalence class.

Step 4. If the equivalence classes have changed, then go to Step 2 with k increased by 1. Otherwise go to Step 5.

Step 5. Generate state-transition functions.

EXAMPLE 5. *Suppose we are given source DTDs, (a) and (b) in Table 1. The set $\mathcal{S}$ has two DTDs and the generated set $\hat{\mathcal{S}}$ has 15 subtrees. The Algorithm 1 is terminated when k = 2, because its equivalence classes are the same with k = 1. The inferred tree automaton is $M = \langle\{F, q_1, \ldots, q_{13}\rangle, \mathcal{N}, \delta, \{F\}\}$ where the state transition functions are as follows. The corresponding tree grammar is shown in Figure 6.* □

$\delta_{\texttt{email}} = q_1$, $\delta_{\texttt{price}} = q_2$, $\delta_{\texttt{isbn}} = q_3$, $\delta_{\texttt{address}} = q_4$,
$\delta_{\texttt{name}} = q_5$, $\delta_{\texttt{title}} = q_6$, $\delta_{\texttt{year}} = q_7$, $\delta_{\texttt{firstname|first}} = q_8$,
$\delta_{\texttt{lastname|last}} = q_9$, $\delta_{\texttt{author}}(q_8, q_9) = q_{10}$,
$\delta_{\texttt{publisher}}(q_5, q_4?, q_1?) = q_{11}$, $\delta_{\texttt{authorname}}(q_8, q_9) = q_{12}$,
$\delta_{\texttt{author}}(q_{12}) = q_{13}$, $\delta_{\texttt{cookbook}}(q_6, q_{13}, q_7, q_3, q_{11}) = F$,
$\delta_{\texttt{bib}}(q_6, q_{10}, q_{11}, q_2, q_7) = F$.

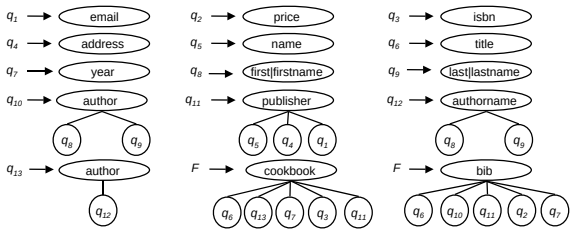


Figure 6: Learned tree grammar

3.4 Minimizer

The *minimizer* subsystem optimizes the state-transition functions generated by the schema learner subsystem and transforms the optimized functions into an integrated schema. The optimization strategy is to merge/modify functions that have parent-child relationships or common labels/subtrees. The algorithm is given as follows:

ALGORITHM 2. *Given a set $\mathcal{P}$ of states,*

Step 1. $\mathcal{P}' = \mathcal{P}$,

Step 2. For every two states $p_i, p_j \in \mathcal{P}$, (1) if they have common root label and p_i is a subtree of p_j, then the root label of p_j is changed to \$. (2) if they have common root label and subtrees, then merge their labels and subtrees. (3) if they have the same subtrees, then merge their labels.

Step 3. If $\mathcal{P}' \neq \mathcal{P}$, go to Step 1. Otherwise return states.

Case 1 in Table 4 is the case of *Step 2(1)* in Algorithm 2. *Step 4* will resolve Case 2. The different combinations of methods of *Step 2* generate several sets of states. The most compact integrated schema is selected as the integrated view.

EXAMPLE 6. *The transition functions in Example 5 can be optimized to the following functions.* □

$\delta_{\text{price}} = q_1$, $\delta_{\text{isbn}} = q_2$, $\delta_{\text{address}} = q_3$, $\delta_{\text{name}} = q_4$, $\delta_{\text{title}} = q_5$, $\delta_{\text{year}} = q_6$, $\delta_{\text{firstname|first}} = q_7$, $\delta_{\text{lastname|last}} = q_8$, $\delta_{\text{email}} = q_9$, $\delta_{\text{author|authorname}}(q_7, q_8) = q_{10}$, $\delta_{\text{publisher}}(q_3, q_4?, q_9?) = q_{11}$, $\delta_{\$}(q_{10}) = q_{12}$, $\delta_{\text{cookbook|bib}}(q_5, q_{12}+, q_{11}, q_6, q_2?, q_1?) = F$.

EXAMPLE 7. *The corresponding integrated schema for optimized functions in Example 6 is as follows:*

$s_F = [cookbook|bib : (title_0, (\$_1)*, publisher_3, year_0, (isbn_0)?, (price_0)?)]$;
$s_1 = [\$: ((author|authorname)_2)]$;
$s_2 = [author|authorname : ((firstname|first)_0, (lastname|last)_0)]$;
$s_3 = [publisher : (name_0, (address_0)?, (email_0)?)]$;

EXAMPLE 8. *Consider the Given query in Table 5(a) that retrieves titles of books/articles whose publisher is* Addison Wesley *and the publishing year is later than 1998. In this case, suppose there are two related XML documents,* `COOKBOOK.xml` *with* `COOKBOOK` *DTD and* `BIB.xml` *with* `BIB` *DTD. The system reformulates the query into two XML-QL queries in Table 5(b) and (c). Case 4 in Table 4 is resolved in this step.* □

```
 (a) Given Query
 1 WHERE <cookbook|bib>
 2         <title>$title</>
 3         <publisher><name> Addison Wesley</></>
 4         <year>$year</>
 5       </>,
 6       $year > 1998
 7 CONSTRUCT <title>$title</>
  (b) For COOKBOOK.xml            (c) For BIB.xml
 8 WHERE                          WHERE
 9  <cookbook>                      <bib year = $year>
10    <title>$title</>                <title>$title</>
11    <publisher>                     <publisher>
12      <name>Addison Wesley</>         <name>Addison Wesley</>
13    </>                             </>
14    <year>$year</>                  </> IN "BIB.xml",
15  </> IN "COOKBOOK.xml",            $year > 1998
16  $year > 1998
17 CONSTRUCT <title>$title</>    CONSTRUCT <title>$title</>
```

Table 5: Given query and reformulated subqueries

mod	Without Renamer				With Renamer				
	t	mt	pre.	accu.	t	mt	pre.	ren.	accu.
10	127	31	1.00	50%	127	31	1.00	25%	100%
20	123	38	1.00	27%	123	38	1.00	57%	93%
30	119	48	0.97	30%	119	48	0.92	58%	96%
40	116	56	0.91	27%	116	54	0.90	60%	91%
50	121	67	0.89	40%	119	61	0.87	43%	93%
60	120	70	0.88	25%	118	64	0.89	63%	88%
70	120	75	0.70	19%	117	66	0.77	67%	86%
80	128	81	0.66	20%	124	72	0.76	64%	85%
90	132	85	0.64	18%	126	74	0.75	68%	90%
100	133	91	0.62	26%	126	80	0.77	56%	82%

Table 6: Summary of experimental results: *mod:* modification rate; *t:* number of types; *mt:* number of merged types; *pre.:* average precision of clustering; *ren.:* number of being renamed elements/number of similar concepts; *accu.:* accuracy, in terms of percentage.

4. EXPERIMENTAL RESULTS

We have implemented a system called *DEEP* based on our approach and conducted some preliminary experiments. The main concern of the experiments is how to evaluate the quality of the results.

We tested DEEP on three domains, *book*, *play* and *movielist*. The test DTDs are prepared as follows. We started by collecting two to three seed DTDs in the test domains. The seed DTDs serve as the "golden rule" for performance evaluation. From these seed DTDs, we constructed 100 DTDs for each domain. These 100 DTDs were generated by various perturbation using tree modification operations in Figure 3 with different modification rates. The modification rate is the ratio of the number of nodes that are modified and the total number of nodes in a given tree. The modification rate starts from 10% and is increased by 10% in each iteration until it reaches 100%. Each iteration generates ten new DTDs by modifying seed DTDs. The modification is conducted by applying a randomly selected operator to the randomly selected node. Each data set has tested in two cases: with or without *renamer* described in Section 3.1.

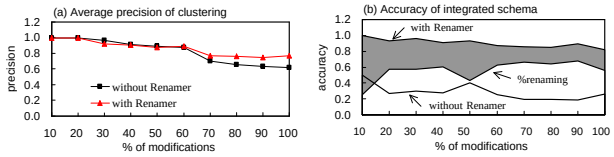


Figure 7: Quality of DEEP

The experimental results are shown in Table 6. The first performance measure is the *correctness* of the clustering. We compute the precision of clustering, pre_n, where n is a clustering level (i.e., the number of classes). The precision pre_n is obtained by the following equation:

$$pre_n = \frac{\sum_{i=1}^{n} \frac{|C_{D_i}|}{|C_i|}}{n}$$

where $|C_{D_i}|$ is the number of correctly clustered DTDs in C_i and $|C_i|$ is the number of DTDs in a DTD class C_i. The fourth and eighth columns of Table 6 show the average precision of clustering without and with renamer, resp., at $n = 3$. As the modification rate increases, the precision degrades gracefully from 100% to 75% with the renamer, as shown in the dotted line of Figure 7 (a). Without the process, we see that the degradation is 38% (from 100% to 62%), 13% worse than with the renamer, as shown in the straight line of Figure 7 (a).

The second measure is the *accuracy* of integrated schema. The result was achieved without DTD Cluster. The fifth and tenth columns of Table 6 show the accuracy for the system to correctly find out similar concepts. In both columns, a/b means that a is the number of similar concepts discovered by the system and b is the total number of similar concepts in the data set. Figure 7(b) shows accuracies ranging from 50% to 18% without renamer and ranging from 100% to 82% with renamer. In the figure, *%renaming* is the percentage of the accuracy achieved by renamer. The shaded area of Figure 7(b) shows the additional associations identified by the system. This illustrates that, with help of renamer, DEEP can recognize up to 25% more of associations.

We now analyze some cases that DEEP fails to find associations between similar concepts. First, if two element types have no common labels and child nodes, they cannot be matched to the same type, though they are conceptually similar. Another problem is that users may provide misleading renaming that confuses our system. To solve this kind

of problems, we provide a tool to correct minor mistakes manually in the DEEP system.

5. RELATED WORK

The most closely related work is a system that learns mappings between source schema and the integrated schema, called LSD [5]. Given an integrated schema, first a set of data sources have been manually mapped to the integrated schema. Then the system learns from these mappings to propose mappings for new data sources. Since the focus of LSD is on how to find one-to-one mappings for leaf elements (i.e., #PCDATA type) of source schema, it can not learn hierarchical or nested structures of source schema. Another difference between LSD and our work is that the integrated schema of their system is given by human, while ours is generated dynamically and automatically by the system.

Another related work is XTRACT [10], a system that extracts a DTD from XML documents. Input XML documents are assumed to conform to the same DTD. From each element in XML documents, XTRACT tries to derive a regular expression that describes the sub-element sequences for the element. Since DTDs are not mandatory, an XML document may not always have an accompanying DTD. Tools that can infer an accurate DTD for given XML documents are useful. It is straightforward to extend our system to extract a DTD from XML documents using *schema learner* subsystem. In this case, the set of sample trees consists of XML documents and the inferred rules will generate a DTD which can cover all input documents.

Extracting schema from semistructured data is considered in [15]. This work focuses on finding a typing for semistructured data. They use an approximately typing method to merge types. Therefore, the generated types are expressed using incoming and outgoing labeled edges. The extracted schema is plain sequences of types instead of arbitrary regular expressions.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a view inference approach that automatically derives integrated view for an IIA to access XML-based sources. This approach enhances the extensibility of an IIA. This problem arises because manually constructing an integrated view for each application domain is error-prone and labor-intensive. To make the development of an extensible IIA more efficient, it is desirable to have a tool that automates this task.

Our solution to this problem is a novel approach that applies hierarchical agglomerative clustering and tree grammar inference technique to generate an integrated view so that an IIA can integrate a new XML document source easily. The experimental results show that our method can effectively and efficiently generate appropriate integrated view for the test domains. The performance is further improved with renamer. We conclude that our view inference approach is a feasible solution to alleviate engineering bottlenecks for the development of extensible IIA.

Our future work includes applying this approach to large-scaled real-world applications. We are participating in a project aiming at building a digital museum of historical photographs in Taiwan. In this project, we need to integrate a variety of photograph collections maintained independently by a variety of agencies. We also planned to investigate how to minimize human intervention in renamer.

Acknowledgements

The research reported here was supported in part by the National Science Council in Taiwan under Grant No. NSC 89-2218-E-002-014, 89-2750-P-001-007, and 89-2213-E-001-039.

7. REFERENCES

[1] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language(XML) 1.0, 1998. W3C Recommendation.

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

[3] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of the Information Processing Society of Japan Conference*, pages 7–18, Tokyo, Japan, October 1995.

[4] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: a query language for XML, 1998.

[5] A. Doan, P. Domingos, and A. Levy. Learning source descriptions for data integration. In *3rd International Workshop on the Web and Databases*, 2000.

[6] O. Duschka and M. Genesereth. Query planning in infomaster. In *Proceedings of the ACM Symposium on Applied Computing*, San Jose, CA, February 1997.

[7] O. Etzioni and D. Weld. A softbot-based interface to the Internet. In *C. ACM*, 1994.

[8] M. Fernandez, J. Simeon, and P. Wadler. XML query languages:experiences and examplars, 1999. W3C Draft manuscript.

[9] H. Fukuda and K. Kamata. Inference of tree automata from sample set of trees. *International Journal of Computer and Information Sciences*, 13:177-196, 1984.

[10] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: a system for extracting document type descriptors from xml documents. In *Proceedings of the ACM SIGMOD*, 2000.

[11] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The information manifold. In *Proceedings of the AAAI Spring Symposium on Information Gathering in Distributed Heterogeneous Environments, Stanford, California*, March 1995.

[12] C. A. Knoblock, Y. Arens, and C. N. Hsu. Cooperating agents for information retrieval. In *Proceedings of International Conference on Cooperative Information Systems*, 1994.

[13] C. Kwok and D. Weld. Planning to gather information. In *Proceedings on 13th National Conference of AI*, 1996.

[14] S. Y. Lu. A tree matching algorithm based on node splitting and merging. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 6, pages 249–256, 1984.

[15] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM SIGMOD*, pages 295–306, Seattle, June 1998.

[16] E. Rasmussen. *Clustering Algorithms*, chapter 16. Prentice Hall, 1992.