

# View Inference for Heterogeneous XML Information Integration

Euna Jeong (eajeong@munhak.inha.ac.kr)  
School of Computer Science and Engineering, Inha University  
253 YOUNHYUN-DONG Nam-Gu Incheon 402-751, South of Korea

Chun-Nan Hsu (chunnan@iis.sinica.edu.tw)  
Institute of Information Science, Academia Sinica  
Nankang 115 Taipei, Taiwan

**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 learner* 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) *Minimizer* optimizes the rules generated in the previous step, transforms them into an integrated view, and generates source descriptions. We have implemented the approach into a system called *DEEP* and tested the system on several domains. Experimental results reveal that this system can effectively and efficiently integrate radically different DTDs.

**Keywords:** XML DTD; Mark-up schemes; Semistructured data; Federated search; Distributed databases; Intelligent Agents;

## 1. Introduction

### 1.1. MOTIVATION

Information integration agents have been receiving a great deal of attention due to the growing number of structured information sources available online. Information integration agents interact with information resources across distributed and heterogeneous environments, thereby freeing the users from having to locate, query, and combine the information from the different sources.

Over the last several years, information integration agents (e.g., (Chawathe et al., 1995; Etzioni and Weld, 1994; Kwok and Weld, 1996; Knoblock et al., 1994; Kirk et al., 1995; Duschka and Genesereth, 1997)) are widely studied and developed. The design concern of these systems vary for different domains, but all share a common need for a layer of domain knowledge, usually referred to an *integrated view* and



© 2002 Kluwer Academic Publishers. Printed in the Netherlands.

*source descriptions*, to seamlessly integrate heterogeneous information sources. The user poses queries in terms of an integrated view as opposed to the schema of the individual sources. The integrated view of underlying sources must be designed for each application domain. The actual data is stored in external sources, called the source schema. In order for the system to be able to answer queries, mappings (a.k.a source descriptions) between the integrated view and the source schema are needed.

However, previous work in information integration requires to construct the integrated view and these mappings manually in a time-consuming and error-prone manner. As the number of application areas and databases to be integrated is rapidly increased, manually designing integrated views and mappings is becoming one of serious bottlenecks to develop large-scaled information integration agents.

The approach presented in this paper is based on previous work in information integration. In particular, this approach addresses the problem of automatic derivation of the integrated views of XML (eXtensible Markup Language) DTDs (Document Type Definition) (Bray et al., 1998) and the source descriptions. Although XML is becoming an industrial standard for exchanging data on the Internet and is considered as a potential solution to data-sharing problems (Seligman and Rosenthal, 2001), it is difficult and sometimes impossible to have a reconciliated DTD when the maintenance of the information sources is independent of the integrator. This paper reports our preliminary results for resolving this problem.

## 1.2. XML INFORMATION INTEGRATION

Figure 1 shows the diagram of an XML information integration agent. The user submits a request to the agent through a user interface. The request is then translated into an XML-QL (Deutsch et al., 1998) query based on an integrated view, which is a query template that helps formulate queries. Given the translated query, *query decomposer* transforms the query into a set of subqueries against each integrated XML information source based on the source descriptions. Finally, *query executor* issues the subqueries towards each information source, combines the answers, and returns the result data to the user as an XML document.

This paper describes how to automatically derive the integrated view and the source descriptions by a *view inference* system as shown in Figure 1. The derivation is conducted offline before the information integration agent is able to provide service. The view inference system serves to automatically discover the association between closely related

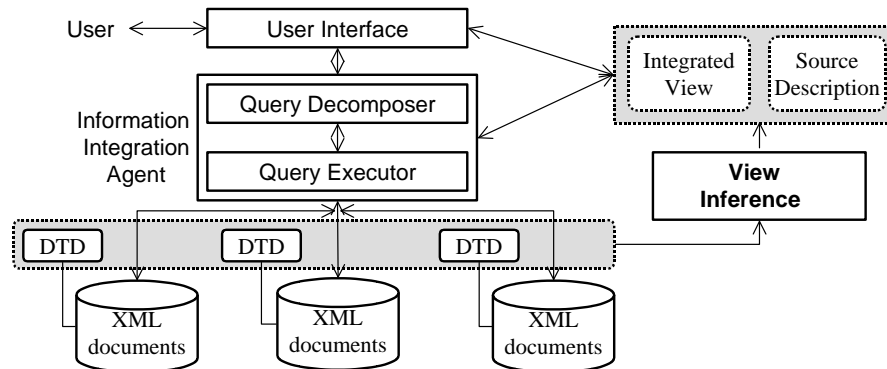


Figure 1. XML Information Integration Agent

DTDs, identify elements with similar underlying semantics, and generate an *integrated view* that covers these elements, as well as the source descriptions.

### 1.3. ILLUSTRATIVE EXAMPLES

XML data is an instance of semi-structured 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 or contain other sub-elements, 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. With a DTD, XML is self-descriptive and provides a semi-structured data model.

Distinct DTDs in similar application domains may use different labels for the same concept, or use the same label for different concepts. The same concept 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, they will reveal structural and naming similarities.

Table I shows several example cases that our approach can handle. The elements of pairs of input DTDs are shown in the second column. Their corresponding subsumption relationships are illustrated as input trees in the left of the third column. For each case, the view inference system can integrate the pair of trees into a single output tree as shown in the right of the third column. We will describe how the view inference system resolves these cases in the following sections.

The remainder of the paper is organized into the following. We begin by defining the problem of heterogeneous XML view inference in Section 2. The details of the view inference system are described

Table I. 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

	Example DTDs	Input $\rightarrow$ Output
1	$D_1: \langle \text{ELEMENT author (firstname,lastname)} \rangle$ $D_2: \langle \text{ELEMENT author authorname} \rangle$ $\langle \text{ELEMENT authorname (firstname,lastname)} \rangle$	
	The same concept may have the same element name, but has different hierarchical structures.	
2	$D_1: \langle \text{ELEMENT publisher (name,email)} \rangle$ $D_2: \langle \text{ELEMENT author (name,email)} \rangle$	
	The two different concepts have the same sub-elements list.	
3	$D_1: \langle \text{ELEMENT featuring \#PCDATA} \rangle$ $D_2: \langle \text{ELEMENT featuring (credit_actor,actor)} \rangle$	
	The same concept may have the same element name, but has different sub-elements list.	
4	$D_1: \langle \text{ELEMENT book (year, ...)} \rangle$ $D_2: \langle \text{ATTLIST bib year \#CDATA} \rangle$	
	<p>The same concept can be defined as an element in one DTD, and an attribute in another DTD.</p> <p>Because attribute names are handled the same as element names, the tree structures are equal.</p> <p>The difference will be described in Section 2.2.</p>	

in Section 3. In Section 4 we report the experimental results which demonstrate the feasibility of our approach. Section 5 reviews closely related work. Section 6 summarizes conclusions.

## 2. Heterogeneous XML View Inference

### 2.1. DTD TREE AND GENERATIVE DTD TREE

We model a DTD as a labeled, directed tree, called *DTD tree*. The nodes in the DTD tree represent objects and are labeled with an element or attribute name. We assume that each DTD has a distinguished root object, and all objects are reachable from the root.

EXAMPLE 1. *Suppose we have four DTDs as shown in Table II. These example DTDs are extracted from published papers and documents:  $D_1$  and  $D_3$  from (Deutsch et al., 1998),  $D_2$  from (Fernandez et al., 1999), and  $D_4$  from (Buneman et al., 1996). This set of DTDs will serve as the example input and will be used to explain our view integration approach throughout the paper.*  $\square$

Table II. Example DTDs

---

```

(D1) COOKBOOK DTD
1 <!ELEMENT cookbook (title, author+, year, isbn, publisher)>
2 <!ELEMENT author (authorname)>
3 <!ELEMENT authorname (firstname, lastname)>
4 <!ELEMENT publisher (name, address)>
(D2) 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)>
(D3) MOVIE DTD
9 <!ELEMENT movie (title, year, director*, featuring, genre)>
10 <!ELEMENT featuring (credit_actor*, actor*)>
11 <!ELEMENT actor (firstname, lastname)>
(D4) LIST DTD
11 <!ELEMENT list (movie*)>
12 <!ELEMENT movie (title, year, directed_by, genres, featuring)>
13 <!ELEMENT directed_by (director*)>
14 <!ELEMENT genres (genre*)>

```

---

Each internal node in the tree can be represented as a *type* whose definition is as follows.

DEFINITION 2. (**Type**) Let  $t = [label(t): children(t)]$ , where

- $t$  is the type id,
- $label(t)$  is the label of  $t$ , that is, the element or attribute name of the corresponding object in the DTD, and
- $children(t)$  is “#PCDATA” if  $t$  is a leaf node, or a list  $(\dots, c, \dots)$  representing the child nodes of  $t$ .  $c$  is a regular expression of element or attribute names if  $t$  is an internal node.  $\square$

DEFINITION 3. (**DTD tree**) A DTD tree is a set of types.  $\square$

EXAMPLE 4. Given the set  $\mathcal{D}$  of source DTDs in Table II, the DTD trees for  $\mathcal{D}$  are shown in Table III. A node representing the element definition  $\langle !ELEMENT bib (title, author+, publisher, price) \rangle$  in line 5 of Table II has the following type definition:  $t_5 = [bib: (title, author+, publisher, price, \underline{year})]$ . Note that in the list of the child nodes, “author+” is a regular expression with “+” representing that there can be one or more “author” elements as the children. The underlined label, such as year of  $t_5$ , corresponds to an XML attribute. For brevity, we abbreviate the type definitions of the leaf nodes in Table III. The type of the leaf nodes has the type id 0. Leaf nodes, such as  $[title : \#PCDATA]$ ,  $[year : \#PCDATA]$ ,  $[isbn : \#PCDATA]$ , etc, has the same type id  $t_0$ .  $\square$

Table III. Example DTD Trees

---

(D <sub>1</sub> )	COOKBOOK DTD
$t_1$	$= [\text{cookbook} : (\text{title}, \text{author}+, \text{year}, \text{isbn}, \text{publisher})];$
$t_2$	$= [\text{author} : (\text{authorname})];$
$t_3$	$= [\text{authorname} : (\text{firstname}, \text{lastname})];$
$t_4$	$= [\text{publisher} : (\text{name}, \text{address})].$
(D <sub>2</sub> )	BIB DTD
$t_5$	$= [\text{bib} : (\text{title}, \text{author}+, \text{publisher}, \text{price}, \underline{\text{year}})];$
$t_6$	$= [\text{author} : (\text{first}, \text{last})];$
$t_7$	$= [\text{publisher} : (\text{name}, \text{email})].$
(D <sub>3</sub> )	MOVIE DTD
$t_8$	$= [\text{movie} : (\text{title}, \text{year}, \text{director}*, \text{featuring}, \text{genre})];$
$t_9$	$= [\text{featuring} : (\text{credit\_actor}*, \text{actor}*)];$
$t_{10}$	$= [\text{actor} : (\text{firstname}, \text{lastname})].$
(D <sub>4</sub> )	LIST DTD
$t_{11}$	$= [\text{list} : (\text{movie}*)];$
$t_{12}$	$= [\text{movie} : (\text{title}, \text{year}, \text{directed\_by}, \text{genres}, \text{featuring})];$
$t_{13}$	$= [\text{directed\_by} : (\text{director}*)];$
$t_{14}$	$= [\text{genres} : (\text{genre}*)].$

---

DEFINITION 5. (**DTD class**) A DTD class is a set of DTD trees.  $\square$

The concept of *DTD class* is designed to group DTD trees that represent the same domain together. 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 III, the output should be two classes. One class contains COOKBOOK and BIB DTDs, and the other contains MOVIE and LIST DTDs.

DEFINITION 6. (**Generative DTD tree**) A generative DTD tree is similar to a DTD tree except that its type definition  $t = [\text{label}(t) : \text{children}(t)]$  may contain “\*” symbols in both  $\text{label}(t)$  and  $\text{children}(t)$ , and that  $\text{label}(t)$  may be a regular expression of element or attribute names, including “\*”.  $\square$

EXAMPLE 7. An example of a generative DTD tree is shown in Table IV.  $\square$

A generative DTD tree characterizes a DTD class when it satisfies two constraints: *coverage* and *compactness*.

DEFINITION 8. (**Coverage**) Let  $D = \{\dots, d, \dots\}$  be a DTD tree, and  $d = [\text{label}(d) : \text{children}(d)]$  be a type definition. Let  $G = \{\dots, g, \dots\}$  be a generative DTD tree, and  $g = [\text{label}(g) : \text{children}(g)]$ .

- A generative DTD tree  $G$  **covers**  $D$  if there exists a mapping  $\sigma$  from  $D$  to  $G$  such that for all  $d$  in  $D$ ,  $\sigma(d) = g$  iff  $L(\text{label}(d)) \subseteq L(\text{label}(g))$  and  $L(\text{children}(d)) \subseteq L(\text{children}(g))$ , where  $L$  is a function that maps a regular expression to its corresponding regular language.
- Similarly, a generative DTD tree  $G$  **covers** a DTD class  $C = \{\dots, D, \dots\}$  iff for all  $D$  in  $C$ ,  $G$  covers  $D$ .  $\square$

Table IV. An example of a generative DTD tree

---

$g_1$	= [cookbook bib : (title, *, publisher, year, isbn?, price?)];
$g_2$	= [* : (author authorname)];
$g_3$	= [author authorname : (firstname first, lastname last)];
$g_4$	= [publisher : (name, address?, email?)];

---

Given a DTD class, there can be a large number of generative DTD trees that cover the DTD class. Ideally, the generative DTD tree should be the smallest one covering the input DTD class so that similar types in different DTDs may be mapped to the same type in the generative DTD tree.

**DEFINITION 9. (Compactness)** A generative DTD tree  $G$  is more **compact** than a generative DTD tree  $G'$  if  $G$  covers a given DTD class which is covered by  $G'$  and has a smaller set of type definitions than  $G'$ . A generative DTD tree  $G$  is the **most compact** one for a given DTD class if there is no generative DTD tree  $G'$  more compact than  $G$ .  $\square$

## 2.2. INTEGRATED VIEW AND SOURCE DESCRIPTION

An *integrated view* is an XML-QL query template transformed from a generative DTD tree. The body statements<sup>1</sup> of the query template are transformed directly from the types in the generative DTD tree.

**DEFINITION 10. (Integrated view)** An *integrated view* is a result of a mapping  $\nu$  from a type in a generative DTD tree to a body statement as follows:

- if  $t = [\text{label}(t) : \#PCDATA]$  is a type for a leaf node, then  $\nu(t)$  maps to “ $\langle \text{label}(t) \rangle \$var \langle / \rangle$ ” where  $\$var$  is a variable for  $\text{label}(t)$ .
- if  $t = [\text{label}(t) : (t_1, \dots, t_m)]$  is a type for an internal node with  $m$  child nodes, then  $\nu(t)$  maps to “ $\langle \text{label}(t) \rangle p_1 \dots p_m \langle / \rangle$ ” where  $p_i$  is a statement for  $t_i$ ,  $1 \leq i \leq m$ .  $\square$

<sup>1</sup> In this paper, we consider queries of the form as XML-QL under the *unordered* data model: **WHERE** *body statements* **CONSTRUCT** *result statements*.

Table V. Integrated view for a book domain

---

```

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

```

---

EXAMPLE 11. Consider the generative DTD tree in Table IV, the mapping function  $\nu$  generate the integrated view as shown in Table V.  $\square$

Given a query based on the integrated view, the system must translate it into subqueries against the related source DTDs by referring to the source descriptions.

DEFINITION 12. (**Source description**) A source description is a function from types in  $G_D$  to types in  $D$  where, given a generative DTD tree  $G$  and a DTD class  $C$ ,  $G_D = \{g | g \in G, \exists d \in D, D \in C, \text{ and } \sigma(d) = g\}$ .  $\square$

$G_D$  is the set of type definitions  $g$  in  $G$  such that we can find a type definitions  $d$  in one of the DTDs in  $C$  such that  $d$  and  $g$  can be associated.

DEFINITION 13. (**Subquery**) A subquery is a result of a function  $\nu'$  from a type in each DTD tree  $D$  in  $C$  to a body statement that is the same as  $\nu$  in Definition 10 with the following additional condition:

- if  $t = [\text{label}(t) : (t_1, \dots, t_m)]$  and  $t_{i+1}, \dots, t_m$ , are types of nodes corresponding to XML attributes, then  $\nu'(t)$  maps to “ $\langle \text{label}(t) p_{i+1} \dots p_m \rangle p_1 \dots p_i \langle / \rangle$ ”, where  $p_k, i+1 \leq k \leq m$ , is a statement of the form: “ $\text{label}(p_k) = \$var$ ” and  $\$var$  is a variable of  $\text{label}(p_k)$ .  $\square$

EXAMPLE 14. Given the generative DTD tree in Table IV and DTD trees  $D_1$  and  $D_2$  in Table III, executable XML-QL subqueries for the

two sources are given in Table VI. In this case, we assume that there are two related XML documents, `COOKBOOK.xml` with `COOKBOOK DTD` and `BIB.xml` with `BIB DTD`. Therefore, there are two subqueries for these two DTDs.  $\square$

Table VI. Reformulated queries 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	

Finally, we give a formal model of heterogeneous XML view inference problem. *Heterogeneous XML view inference* is a function from a set of DTD trees (in a DTD class) to an integrated view derived from the *most compact* generative DTD tree that *covers* the given DTD trees, as well as source descriptions for the given DTD trees.

### 3. View Inference System

Figure 2 illustrates the overall approach to generating integrated views and source descriptions from a set of DTDs. As shown in the figure, the system consists of three main components: *DTD clustering*, *schema learner*, and *minimizer*. We summarize our approach to heterogeneous XML view inference in the data-flow order. For details, see (Jeong and Hsu, 2001; Jeong, 2002).

#### 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

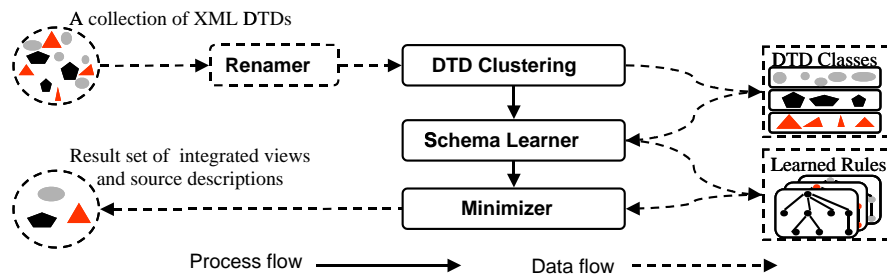


Figure 2. Diagram of the View Inference System

and structural hints in order for the system to conveniently associate related elements in the different DTDs, while leaf nodes offer very limited information to the system. The 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 manually renamed to a label of an internal or leaf element in different DTDs so that they will be considered as sharing the same underlying concept. This renaming is not to physically modify the original DTDs, but used to construct *source descriptions*. Therefore, an information integration agent only needs to post-process the query-results returned from the subqueries. For instance, in Table III, element name `first` in  $t_6$  may be renamed to `firstname` and `last` to `lastname`, or vice versa.

### 3.2. DTD CLUSTERING

This module contains three steps to cluster a collection of source DTD trees into DTD classes. As the first step, we merge similar types. The purpose is to reduce the number of types as well as the distance between DTD trees. This will allow DTDs of similar domains to have a better chance of being clustered together. The second step is to compute the distance between DTD trees. Based on the distances, the whole clustering process can be completed with a hierarchical clustering algorithm.

We use a simple heuristic to merge types. First, we consider types that have the same label. If two types in different DTDs have the same label and common children, then these two types are merged into new one with the common label and the union of children.

**EXAMPLE 15.** *Given the DTD trees in Table III, The modified DTD trees are given in Table VII. Types  $t_8$  and  $t_{12}$  are merged into  $s_7 = [\text{movie} : (\text{title}, \text{year}, \text{director}^*, \text{featuring}, \text{genre}?, \text{directed\_by}?, \text{genres}?)]$  and  $t_4$  and  $t_7$  into  $s_4 = [\text{publisher} : (\text{name}, \text{address}?, \text{email}?)]$ . An atomic type  $t_0 = [\text{featuring} : \#PCDATA]$  in  $D_4$  DTD tree is merged*

to  $t_9 = s_8 = [\textit{featuring} : (\textit{credit\_actor*}, \textit{actor*})]$ . The DTD trees are modified based on the merged types. The merging will introduce children types from different DTD trees due to the union applied in the merging step. As a result, the modified DTD trees may contain more types (nodes) than their originals.  $\square$

Table VII. Modified DTD trees

---

(D <sub>1</sub> ) COOKBOOK DTD
$s_1 = [\textit{cookbook} : (\textit{title}, \textit{author}+, \textit{year}, \textit{isbn}, \textit{publisher})];$
$s_2 = [\textit{author} : (\textit{authorname})];$
$s_3 = [\textit{authorname} : (\textit{firstname} \textit{first}, \textit{lastname} \textit{last})];$
$s_4 = [\textit{publisher} : (\textit{name}, \textit{address}?, \textit{email}?)].$
(D <sub>2</sub> ) BIB DTD
$s_5 = [\textit{bib} : (\textit{title}, \textit{author}+, \textit{publisher}, \textit{price}, \underline{\textit{year}})];$
$s_6 = [\textit{author} : (\textit{firstname} \textit{first}, \textit{lastname} \textit{last})];$
$s_4 = [\textit{publisher} : (\textit{name}, \textit{address}?, \textit{email}?)].$
(D <sub>3</sub> ) MOVIE DTD
$s_7 = [\textit{movie} : (\textit{title}, \textit{year}, \textit{director*}, \textit{featuring}, \textit{genre}?, \textit{directed\_by}?, \textit{genres}?)];$
$s_8 = [\textit{featuring} : (\textit{credit\_actor*}, \textit{actor*})];$
$s_9 = [\textit{actor} : (\textit{firstname} \textit{first}, \textit{lastname} \textit{last})];$
$s_{10} = [\textit{directed\_by} : (\textit{director*})];$
$s_{11} = [\textit{genres} : (\textit{genre*})].$
(D <sub>4</sub> ) LIST DTD
$s_{12} = [\textit{list} : (\textit{movie*})];$
The rest is the same as $s_7 - s_{11}$ .

---

We employ a hierarchical clustering method (Frakes and Baeza-Yates, 1992) used widely in information retrieval. The basic idea is the following: 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 extend Lu's algorithm (Lu, 1984) to compute the distance between two labeled trees. The distance is computed by calculating the minimum number of modifications required to transform the input tree into a reference tree. For computing the distance between classes during a hierarchical clustering process, the average distance is used.

### 3.3. SCHEMA LEARNER

Once the DTD trees have been clustered into DTD classes, *schema learner* is ready to create a generative DTD tree for each DTD class. We address this problem with a tree grammar inference approach. *Grammatical inference* is the task for inducing hidden grammatical rules from a set of examples. The problem of deriving a generative DTD tree from similar DTD trees can be reduced to this task. We adopt the

$k$ -follower method (Fukuda and Kamata, 1984), which applies a simple state-merging heuristic process. Given a DTD class, *schema learner* generates a tree automaton to describe DTD trees in the DTD class. The corresponding tree grammar of the tree automaton describes an infinite language, containing input trees in the DTD class.

**DEFINITION 16. (k-follower)** 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, “\$” be a special character not in the set of node labels in  $\mathcal{S}$ ,  $U(b \Leftarrow \$)$  be the replacement of the subtree of tree  $U$  at node  $b$  with \$,  $b/U$  be the subtree of tree  $U$  with node  $b$  as the root, and  $Depth_U(b)$  be the number of nodes on the path from the root of  $U$  to  $b$ , excluding  $b$ . The  $k$ -follower  $H_{\hat{\mathcal{S}}}^k(T)$  of a tree  $T$  with respect to  $\mathcal{S}$  is defined by

$$H_{\hat{\mathcal{S}}}^k(T) = \{U(b \Leftarrow \$)\}$$

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  $Depth_U(b) = k$ .  $\square$

**DEFINITION 17. (Equivalence relation)** Let  $R^k$  be an equivalence relation on trees in  $\mathcal{S}$ .  $(T, U) \in R^k$  iff  $H_{\hat{\mathcal{S}}}^k(T) = H_{\hat{\mathcal{S}}}^k(U)$  for  $k \geq 0$ .  $\square$

The equivalence classes are induced from the equivalence relation and defined as follows:  $[T]_{R^k} = \{U \in \hat{\mathcal{S}} \mid (T, U) \in R^k\}$ . The equivalence classes will become the states of a nondeterministic tree automaton.

**DEFINITION 18. (Nondeterministic tree automaton)** A nondeterministic tree automaton is a quadruple  $\langle \mathcal{Q}, \mathcal{N}, \delta, \mathcal{F} \rangle$  where

- $\mathcal{Q}$  is the set of states,
- $\mathcal{N}$  is the set of labels,
- $\delta$  is the set of state-transition functions, which is a mapping  $\mathcal{N} \times 2^{\mathcal{Q}} \rightarrow \mathcal{Q}$  defined as follows:
  - if  $T$  is a tree whose root label is “ $a$ ” with  $m$  child subtrees  $t_1, t_2, \dots, t_m$ , then  $q = \delta_a(q_1, q_2, \dots, q_m) = \{[T]_{R^k} \mid T \in \hat{\mathcal{S}}, t_i \in q_i = [T_i]_{R^k}, \text{ and } 1 \leq i \leq m\}$
  - if  $T$  is a tree with a single node “ $a$ ”, then  $q = \delta_a = \{[a]_{R^k}\}$ ,
- $\mathcal{F}$  is the set of final states,  $\mathcal{F} \subseteq \mathcal{Q}$ , whose  $k$ -follower contains “\$” as an element, i.e.,  $\mathcal{F} = \{[T]_{R^k} \mid \$ \in H_{\hat{\mathcal{S}}}^k(T)\}$ .  $\square$

The inference algorithm of tree automata using  $k$ -follower is formally presented as follows:

ALGORITHM 19. *Given a set  $\mathcal{S}$  of trees,*

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

**Step 2.** *For all  $T$  in  $\hat{\mathcal{S}}$ , generate  $H_{\hat{\mathcal{S}}}^k(T)$ .*

**Step 3.** *For all pairs of trees,  $T$  and  $U$  in  $\hat{\mathcal{S}}$ , not belonging to the same equivalence class, if  $H_{\hat{\mathcal{S}}}^k(T) = H_{\hat{\mathcal{S}}}^k(U)$ , then make their equivalence classes identical.*

**Step 4.** *If the equivalence classes are changed, then go to Step 2 with  $k$  increased by 1.*

**Step 5.** *Generate state-transition functions.* □

EXAMPLE 20. *Suppose we are given DTD trees,  $D_1$  and  $D_2$  in Table III. Let  $\mathcal{S} = \{D_1, D_2\}$ . Therefore,  $\hat{\mathcal{S}}$  contains 15 subtrees. Algorithm 19 is terminated when  $k = 2$  because its equivalence classes are the same as  $k = 1$ . The inferred tree automaton is  $M = \langle \mathcal{Q}, \mathcal{N}, \delta, \mathcal{F} \rangle$ , where  $\mathcal{Q} = \{F, q_1, q_2, \dots, q_{13}\}$ ,  $\mathcal{N}$  is the set of labels in  $\mathcal{S}$ ,  $\delta$  is  $\{ \delta_{\text{email}} = q_1, \delta_{\text{price}} = q_2, \delta_{\text{isbn}} = q_3, \delta_{\text{address}} = q_4, \delta_{\text{name}} = q_5, \delta_{\text{title}} = q_6, \delta_{\text{year}} = q_7, \delta_{\text{firstname}} = q_8, \delta_{\text{lastname}} = q_9, \delta_{\text{author}}(q_8, q_9) = q_{10}, \delta_{\text{publisher}}(q_5, q_4, q_1) = q_{11}, \delta_{\text{authorname}}(q_8, q_9) = q_{12}, \delta_{\text{author}}(q_{12}) = q_{13}, \delta_{\text{cookbook}}(q_6, q_{13}, q_7, q_3, q_{11}) = F, \delta_{\text{bib}}(q_6, q_{10}, q_{11}, q_2, q_7) = F \}$ , and  $\mathcal{F} = \{F\}$ . □*

### 3.4. MINIMIZER

Finally, the *minimizer* optimizes the state-transition functions generated by the *schema learner* and transforms the optimized functions into a generative DTD tree. The optimization strategy is to merge state-transition functions that have precedent-descendant relationships or common labels/subtrees. If two states have a parent-child relationship and their labels are equal, the label of parent state is changed to “\*” symbol. Case 1 and 2 in Table I are resolved in this process.

EXAMPLE 21. *The state transition functions in Example 20 can be optimized as follows:  $\delta_{\text{email}} = q_1, \delta_{\text{price}} = q_2, \delta_{\text{isbn}} = q_3, \delta_{\text{address}} = q_4, \delta_{\text{name}} = q_5, \delta_{\text{title}} = q_6, \delta_{\text{year}} = q_7, \delta_{\text{firstname}} = q_8, \delta_{\text{lastname}} = q_9, \delta_{\text{author|authorname}}(q_8, q_9) = q_{10}, \delta_{\text{publisher}}(q_5, q_4, q_1) = q_{11}, \delta_*(q_{10}) = q_{12}, \delta_{\text{cookbook|bib}}(q_6, q_{12}, q_{11}, q_7, q_2, q_3) = F$ . The corresponding generative DTD tree was shown in Table IV.* □

The different combinations of state merging methods generate several generative DTD trees. The quality of these generative DTD trees will be evaluated by *coverage* and *compactness* in Definitions 8 and 9, respectively. Then the most compact generative DTD tree is selected as the integrated view.

EXAMPLE 22. Consider the given query (a) in Table VIII that retrieves titles of books/articles whose publisher is “Addison Wesley” and the published 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 subqueries, (b) and (c) in Table VIII. Case 4 in Table I is resolved in this step. Now, the information agent can provide a uniform interface for its user to retrieve the data from both XML data sources. □

Table VIII. Given Query and Reformulated Subqueries

(a) Given query	
1	WHERE <cookbook bib>
2	<title>\$title</>
3	<publisher><name> Addison Wesley</></>
4	<year>\$year</>
5	</> ,
6	\$year > 1998
7	CONSTRUCT <result><title>\$title</></>
(b) For COOKBOOK DTD	(c) For BIB DTD
8	WHERE
9	<cookbook>
10	<title>\$title</>
11	<publisher>
12	<name> Addison Wesley</></>
13	<year>\$year</>
14	</> IN "COOKBOOK.xml",
15	\$year > 1998
16	CONSTRUCT the same as given query
	WHERE
	<bib year = \$year>
	<title>\$title</>
	<publisher>
	<name> Addison Wesley</></>
	</> IN "BIB.xml",
	\$year > 1998
	CONSTRUCT the same as given query

#### 4. Analytical and Empirical Evaluation

This section provides both analytical and empirical evaluation of our approach.

##### 4.1. CORRECTNESS RESULTS

Recall that in Definitions 8 and 9, we presented the conditions, i.e., *coverage* and *compactness*, that must be held for a generative DTD tree.

Coverage requires that a generative DTD tree for a DTD class should cover all DTD trees of the DTD class. Given a DTD class  $\mathcal{C}$ , Algorithm 19 generates a tree automaton. The tree automaton has the following characteristic (Fukuda and Kamata, 1984).

**THEOREM 23.** *Let  $T(M)$  be the set of trees accepted by tree automaton  $M$ . For each DTD class  $\mathcal{C}$ , inferred tree automaton  $M$  for any nonnegative integer  $k$  accepts all DTD trees  $T_i \in \mathcal{C}$ , i.e.,  $\mathcal{C} \subseteq T(M)$ .*

*Proof of Theorem 23.* By Algorithm 19, Step 1 generates  $\hat{\mathcal{C}}$ . If  $T_i \in \mathcal{C}$ , (not  $\mathcal{C}_{sub}$ ) then  $\$ \in H_{\hat{\mathcal{C}}}^k(T_i)$  for all values of  $k$  by condition 2 of Definition 16. Since the set of final states is  $\mathcal{F} = \{[T]_{R^k} | \$ \in H_{\hat{\mathcal{C}}}^k(T)\}$ , we get  $[T_i]_{R^k} \in \mathcal{F}$ .  $\square$

Minimizer in Section 3.4 is the process of minimizing tree automaton  $M$  generated by Algorithm 19. Since minimized tree automaton  $M'$  is constructed by merging states of  $M$ , each state in  $M'$  may consist of one or more states of  $M$ .

**DEFINITION 24. (Minimized automaton)** *Let  $M = \langle \mathcal{Q}, \mathcal{N}, \delta, \mathcal{F} \rangle$  be a finite automaton and let  $\mathcal{P} = (P_1, P_2, \dots, P_m)$  be a partition of  $\mathcal{Q}$  such that  $\bigcup P_i = \mathcal{Q}$  and  $P_i \cap P_j = \{\}$ , for any  $1 \leq i, j \leq m$ . The automaton  $M' = \langle \mathcal{Q}', \mathcal{N} \cup \{*\}, \gamma, \mathcal{F}' \rangle$  derived from  $M$  such that*

- $\mathcal{Q}' = \mathcal{P}$ ,
- $\gamma_a(\dots, P_i, \dots) = P_j$  if  $\exists q_i \in P_i$  and  $q_j \in P_j$  such that  $\delta_b(\dots, q_i, \dots) = q_j$  and  $L(b) \subseteq L(a)$ , and
- $\mathcal{F}' = \{P_i | \exists q_j \in \mathcal{F}, q_j \in P_i, P_i \in \mathcal{P}\}$ ,

*is called the automaton minimized from  $M$  by partition  $\mathcal{P}$ .*  $\square$

The following theorem follows directly from the above definition.

**THEOREM 25.** *If  $M$  is a tree automaton and  $M'$  is a tree automaton minimized from  $M$  by an arbitrary partition of the states of  $M$ , then  $T(M) \subseteq T(M')$ .*

*Proof of Theorem 25.* By Definition 24.  $\square$

Thus we can establish the following theorem:

**THEOREM 26.** *If  $M$  is a tree automaton for DTD class  $\mathcal{C}$  and  $M'$  is a tree automaton minimized from  $M$  by an arbitrary partition of the states of  $M$ , then  $M'$  accepts all DTD trees  $T_i \in \mathcal{C}$ , i.e.,  $\mathcal{C} \subseteq T(M')$ .*

*Proof of Theorem 26.* By Theorems 23 and 25.  $\square$

In other words, if  $G$  is a generative DTD tree of DTD class  $\mathcal{C}$  generated by  $M'$ , then  $G$  covers all DTD trees in  $\mathcal{C}$ .

The other criteria for the output generative DTD trees is the compactness. Although we cannot guarantee that our algorithm always returns the most compact generative DTD trees, we still can empirically show that the output is small in most cases. On the other hand, in some cases, the most reasonable integrated view may not be the smallest one. Human intervention is still required for the judgement. In the following sub-section, we design experiments to show that our algorithm can generate a reasonable integrated view even though the input DTDs are drastically different.

#### 4.2. 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 *movie-list*. 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 construct 100 DTDs for each domain by using various perturbations 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 for each domain. 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.

The experimental results are shown in Table IX. The first performance measure is the *precision* of the clustering. The precision of clustering is the average of the ratio of the correctly clustered DTDs and the number of DTDs in each DTD class. The fourth and ninth columns of Table IX show the average precision of clustering without and with *renamer*, respectively, when the number of classes generated by our clustering algorithm is equal to three. As the modification rate increases, the precision degrades gracefully from 100% to 75% with the *renamer*, as shown in the dotted line of Figure 3 (a). Without the *renamer*, 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 3 (a). But the precisions with and without the *renamer* are similar until the modification level is increased by 60%.

The second measure is the *accuracy* of integrated schema. The result was achieved without DTD cluster. The fifth and eleventh columns of

Table IX. 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.*: number of correctly discovered similar concepts by DEEP/number of similar concepts *%accu.*: accuracy, in terms of percentage.

mod	Without Renamer					With Renamer					
	t	mt	pre.	accu.	%accu.	t	mt	pre.	%ren.	accu.	%accu.
10%	127	31	1.00	4/8	50%	127	31	1.00	25%	8/8	100%
20%	123	38	1.00	4/15	27%	123	38	1.00	57%	14/15	93%
30%	119	48	0.97	8/25	30%	119	48	0.92	58%	26/27	96%
40%	116	56	0.91	9/33	27%	116	54	0.90	60%	30/33	91%
50%	121	67	0.89	12/30	40%	119	61	0.87	43%	28/30	93%
60%	120	70	0.88	10/40	25%	118	64	0.89	63%	35/40	88%
70%	120	75	0.70	8/42	19%	117	66	0.77	67%	36/42	86%
80%	128	81	0.66	9/46	20%	124	72	0.76	64%	39/46	85%
90%	132	85	0.64	9/49	18%	126	74	0.75	68%	44/49	90%
100%	133	91	0.62	13/50	26%	126	80	0.77	56%	41/50	82%

Table IX 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. In the sixth column, without the renamer, the performance is not satisfactory, as accuracies range from 50% to 18%. In contrast, with the renamer, DEEP performed quite well with accuracies ranged from 100% to 82% in the twelfth column. Figure 3 (b) illustrates each accuracy in terms of percentage, with and without the renamer. In the figure, *%renaming* is the percentage of the accuracy achieved by the renamer. The shaded area of Figure 3 (b) shows the additional associations identified by the system. This illustrates that, with help of the renamer, DEEP can recognize up to 25% more of associations. Also, DEEP can correctly recognize more associations without help of the renamer until the rate of modification reaches 60%. With XML documents and renamer, our approach can handle real-world XML documents with DTDs that are different by 90%.

We now analyze some cases that DEEP fails to find associations between similar concepts. First, if two element types have no common

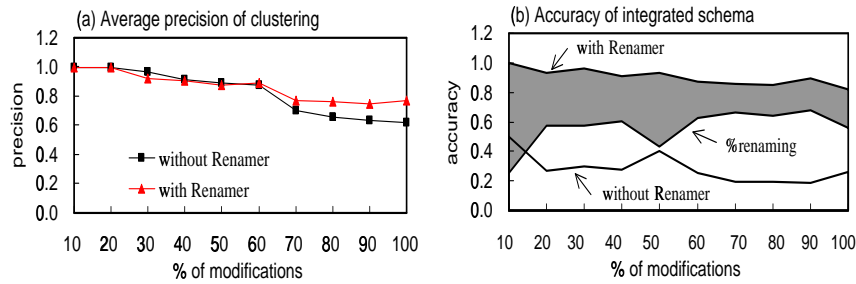


Figure 3. Quality of DEEP

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 this type of minor mistakes manually in the DEEP system.

## 5. Related Work

The most closely related work is LSD (Doan et al., 2000), a system that learns mappings between source schema and the integrated schema. The input includes a set of data sources and their schema information that has been manually mapped to an integrated schema. In the training phase, the system learns from these mappings using general and domain specific learners. Then in the mapping phase, the system can propose mappings for new data sources. In contrast to LSD, DEEP can capture domain-specific knowledge, not especially providing it, but by modeling DTDs based on parent-child relationships of elements. Another difference between LSD and DEEP is that the integrated schema is given by human in LSD, while DEEP can dynamically and automatically generate it.

Another related work is XTRACT (Garofalakis et al., 2000), 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 in order to extract a DTD from XML documents using the *schema learner*. In this case,

the set of sample trees consisting of XML documents would allow the inferred rules to generate a DTD which can cover all input documents.

Extracting schema from semi-structured data is considered in (Nestorov et al., 1998). This work focuses on finding a typing for semi-structured data. The generated types are expressed using incoming and outgoing labeled edges. Since the extracted schema is plain sequences of types instead of arbitrary regular expressions, it lacks the coverage property by missing some elements.

## 6. Conclusions and Future Work

We have proposed a view inference approach that automatically derives an integrated view and a source description in order for an information integration agent (IIA) to access XML-based sources. This approach enhances its extensibility. 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 clustering and tree grammar inference technique to generate an integrated view so that an IIA can integrate a new XML document source easily. We show that our algorithm can always generate an integrated view that covers all information sources. Experimental results show that our method can effectively and efficiently generate appropriate integrated views for the test domains. We conclude that our view inference approach is a feasible solution for alleviating engineering bottlenecks in 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. Another project is to build a virtual fab for a semiconductor manufacturing firm. In this project, we apply IIA to allow users to access information from their partners in the supply chain and remote fabs, and DEEP is applied to integrate these information sources. We also plan to investigate how to minimize human intervention in the renamer process.

### *Acknowledgements*

We wish to thank Dr. Jane Y.-J. Hsu for her valuable comments. This reported research 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.

## References

- Bray, T., J. Paoli, and C. M. Sperberg-McQueen: 1998, 'Extensible Markup Language(XML) 1.0'. W3C Recommendation.
- Buneman, P., S. Davidson, G. Hillebrand, and D. Suciu: 1996, 'A Query language and optimization techniques for unstructured data'. In: *Proceedings of the International Conference on Management of Data*.
- Chawathe, S., H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom: 1995, 'The TSIMMIS Project: Integration of Heterogeneous Information Sources'. In: *Proceedings of the Information Processing Society of Japan Conference*. Tokyo, Japan, pp. 7–18.
- Deutsch, A., M. Fernandez, D. Florescu, A. Levy, and D. Suciu: 1998, 'XML-QL: a query language for XML'. In: *QL'98 - The W3C Query Languages Workshop*. Boston, MA, USA.
- Doan, A., P. Domingos, and A. Levy: 2000, 'Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach'. In: *Proceedings of the International Conference on Management of Data*. Dallas, Texas, USA.
- Duschka, O. and M. Genesereth: 1997, 'Query planning in Infomaster'. In: *Proceedings of the ACM Symposium on Applied Computing*. San Jose, CA, USA.
- Etzioni, O. and D. Weld: 1994, 'A softbot-based interface to the Internet'. *Communication of the ACM* **37**(7), 72–76.
- Fernandez, M., J. Simeon, and P. Wadler: 1999, 'XML Query languages: Experiences and Exemplars'. W3C Draft manuscript.
- Frakes, F. and R. Baeza-Yates (eds.): 1992, *Information Retrieval : Data structures & Algorithms*, Chapt. 16 Clustering Algorithms. Prentice Hall.
- Fukuda, H. and K. Kamata: 1984, 'Inference of Tree Automata from Sample Set of Trees'. *International Journal of Computer and Information Sciences* **13**, 177–196.
- Garofalakis, M., A. Gionis, R. Rastogi, S. Seshadri, and K. Shim: 2000, 'XTRACT: a system for extracting document type descriptors from XML documents'. In: *Proceedings of the International Conference on Management of Data*. Dallas, Texas, USA.
- Jeong, E.: 2002, 'Ontology Integration in XML'. Doctoral Dissertation, Department of Computer Science and Information Engineering, National Taiwan University.
- Jeong, E. and C.-N. Hsu: 2001, 'Induction of Integrated Schema for XML Data with Heterogeneous DTDs'. In: *Proceedings of the Tenth International Conference on Information and Knowledge Management*. Atlanta, GA, USA, pp. 151–158.
- Kirk, T., A. Y. Levy, Y. Sagiv, and D. Srivastava: 1995, 'The Information Manifold'. In: *Proceedings of the AAAI Spring Symposium on Information Gathering in Distributed Heterogeneous Environments*. Stanford, CA, USA.
- Knoblock, C., Y. Arens, and C.-N. Hsu: 1994, 'Cooperating Agents for Information Retrieval'. In: *Proceedings of the 2nd International Conference on Cooperative Information Systems*. Toronto, Ontario, Canada.
- Kwok, C. T. and D. S. Weld: 1996, 'Planning to Gather Information'. In: *Proceedings of 13th AAAI National Conference on Artificial Intelligence*. Portland, Oregon, USA, pp. 32–39, AAAI/MIT Press.
- Lu, S. Y.: 1984, 'A tree matching algorithm based on node splitting and merging'. *IEEE Transactions on Pattern Analysis and Machine Intelligence* **6**, 249–256.
- Nestorov, S., S. Abiteboul, and R. Motwani: 1998, 'Extracting schema from semistructured data'. In: *Proceedings of the International Conference on Management of Data*. Seattle, WA, USA, pp. 295–306.
- Seligman, L. and A. Rosenthal: 2001, 'XML's Impact on Databases and Data Sharing'. *IEEE Computer* **34**(6), 59–67.