WPI-CS-TR-01-13 May 2001

Automatic Transformation of XML Documents

by

Hong Su

Harumi Kuno and Elke A. Rundensteiner

Automatic Transformation of XML Documents[*]

Hong Su[†], Harumi Kuno[‡] and Elke A. Rundensteiner[†]

(†)Computer Science Department, Worcester Polytechnic Institute
Worcester, MA 01609
{suhong, rundenst}@cs.wpi.edu

(‡) Hewlett-Packard Labs
Palo Alto, CA 94304
harumi_kuno@hp.com

September 16, 2003

Abstract

XML documents exchanged between two E-Services partners may need to be reconciled to conform to the receiver's expecting structure. Proprietary manual translation of XML documents is not only laborious but also error-prone for the quickly evolving E-Services world. We propose an approach that automatically identifies most likely matching choices between the schemas and then generates the corresponding XSLT script to perform the appropriate data transformation. For this, we introduce a set of transformation operations on XML's hierarchical structure. Our system will set up the semantic relationship between two schemas by discovering the operations that transform the source schema to the target one. We report the experimental studies on real DTDs.

[*]We thank HP for partial support of Hong Su in terms of a grassroot grant.

1 Introduction

1.1 Motivation.

In the vision of E-Services [Kun00], services effortlessly and dynamically discover, connect to, and conduct business with each other, regardless of underlying platforms and transports. Currently, technologies such as E-Speak [BKGD00], BizTalk [Biz01], RosettaNet PIPs [Ros01], and CommerceXML (cXML) [cXM01] enable E-Services running on heterogeneous devices to discover and exchange messages with each other. However, these technologies do not address the problem of how to reconcile structural differences between the types of documents the two E-Services might expect. For example, let there be two E-Services, Service A and Service B, which front different companies. Suppose that Service A wanted to purchase something from Service B, and that Service B requires Service A to submit a purchase order. The two services might structure their purchase order documents differently. Figure 1 shows two example document structures described by DTDs used by Service A and Service B respectively. Issues we propose to address include the identification of the most likely semantic relationships between the two structures and then the corresponding transformation of an XML document from one structure to the other.

```
<!ELEMENT company (address, cname, personnel)>
<!ATTLIST comapny id ID #REQUIRED>
<!ELEMENT address (street, city, state, zip)>
<!ELEMENT personnel (person)+>
<!ELEMENT person (name, email?, url?, fax+)>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT name (family|given|middle?)*>
<!ELEMENT cname (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
```

Figure 1: DTD1 of E-Service A's Purchase Order

```
<!ELEMENT company (cname,(street, city, state, zip), personnel)>
<!ATTLIST comapny id ID #REQUIRED>
<!ELEMENT personnel (person)+>
<!ELEMENT person (name, email+, url?, fax, fax, phonenum)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT name (first, last)>
<!ELEMENT cname (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
<!ELEMENT phonenum (#PCDATA)>
```

Figure 2: DTD2 of E-Service B's Purchase Order

1.2 Our Approach.

In order to represent the semantic relationships between two XML documents, we represent the documents' schemas as trees. Then we propose a set of *schema transformation operations* that establish semantic relationship between one tree and the other. We also define a cost model on the operations. We introduce an algorithm that discovers a sequence of such operations that transform the source schema tree to a target schema tree. We then generate the corresponding eXtensible Stylesheet Language Transformations (XSLT) [Gro] scripts, and perform them on the source XML document to produce the target XML document. Our approach can be applied to either XML-Schema [W3C01] or DTDs [W3C98]. However, since DTDs are still the dominant industry standard, in the remainder of this paper we use DTDs as the representation format of the schema of XML documents.

2 Related Work

2.1 Schema Translation

Schema Restructuring Thoery. How to build new schemata from existing ones using various structural manipulations has been a long studied topic in the area of relational databases. [Hul84] first proposed the notion of *relative information capacity*. Intuitively, a schema S_2 has more information capacity than a schema S_1 if every instance of S_1 can be mapped to an instance of S_2 without loss of information. Some work [MS92] [RR87] on translation and integration has used information capacity equivalence as a basis for judging the correctness of transformed schemas. [MIR93] presents a classification of common integration and translation tasks and derive from them the relative information capacity requirements of the original and transformed schemas.

Schema Matching. Before a schema can be restructured, a semantic relationship between the source schema and target schema must be set up. Several methodologies have been proposed in relational schema matching. *ARTEMIS* [CA99] [BCV] [BCVV98] is a tool that supports analyzing and reconciling sets of heterogeneous data schemas. Schema analysis in ARTEMIS is performed according to the concept of affinity. The system evaluates the schema affinity by measuring similarity of their names (based on a thesauri), data type (based on compatibilities) and structures (based on the similarity of relationships within the entities). [PSU98] uses a similar idea as [BCVV98] to discover common properties holding among objects in different schemes to achieve schema integration. However, a relational schema is flat while the schema of XML is hierarchical.

TranScm [MZ98] deals with a more general data model rather than being limited to the relational model only. It defines a common schema model and data model. And it also offers a set of "rules" (i.e., matcher) that describe how to match a component in the source schema with a corresponding component in the target schema. The matching is a performed node by node starting at the top. Rules are checked in a fixed order based on their priorities. However being a system that aims at providing a general approach, it may not be efficient nor even powerful enough in solving the specific schema matching problem in the domain of XML. How to assign the priority for each rule or how to efficiently find an appropriate rule in the rule bases are all critical unsolved issues while this approach is put in the specific context.

[DDL00] can handle hierarchical schema structures. It is a machine-learning approach to match a new schema to a predetermined global schema. The learner is trained by a set of user-provided mapping from a data source to the global schema and then discovers the characteristic instance patterns. Hence given a new data source, by applying the discovered matching patterns, it can determine all one-to-one mappings between the leaf nodes of two schema trees . However it does not match source-schema elements at higher levels as it would require learning methods that deal with structures. And in our scenario, if there are no example data sets of both source and target XML documents, this approach cannot be applied.

2.2 Tree Matching

Since XML's schema can be modeled as a tree, XML's schema matching has some similarity with tree matching. Using insertion, deletion and relabelling as the edit operations, [ZS89, SWZS90] defines a change detection problem for ordered trees while [ZWS95] presented the approach for unordered tress. In all above work, the matching is a prurely structural matching. It treats the label as a second-concern, i.e., the cost of relabelling is always cheaper than that of deleting a node with the old label and inserting a node with the new label. This assumption will not hold in our domain.

[CRGMW96] introduces one more edit operation *move*. It adapts a simple cost model in which *insert*, *delete* and *move* are all unit cost operations, i.e., cost is 1, while the cost of relabelling operation is given by a function evaluating how different the new label is from the old label. Also it makes an important assumption to help matching the nodes. It assumes that each node of the input trees has a special tag that describes its semantics and in the output tree there is no duplicates (or near duplicates) in the labels found in the input tree. [CGM97] introduces more edit operations, allows flexible cost models and drops the assumption in [CRGMW96]. However it takes time quadratic in the size of the input.

There are two differences in our tree model that disables the applicability of their approaches. First, some of their operations are not meaningful in our model. We need some other XML-Specific edit operations. Second, the assumption made in [CRGMW96] only hold for part of the nodes in the model. Hence it is not suitable to use the assumption to direct the mapping, neither is it suitable to completely discard the assumption which results in a high time complexity.

Our approach is more of a tree matching flavor. However we are exploring an approach further than traditional tree matching (graph isomorphism) which will incorporate the domain characteristics of XML's schema.

3 DTD Data Model

Document Type Definition (DTD) [W3C98] enforces the structure of XML documents. DTD allows for properties or constraints to be defined on elements and attributes. A DTD defines a document's structure as a list of element type declarations. Elements represent the tag names that can be used in an XML document. Elements can in turn have content particles or attributes or be empty. The structure of elements is defined via a *content-model* built out of operators applied to its content particles. Content particles can be grouped as sequences (e.g., *a,b*) or as choices (e.g., $a|b$) to be a content particle again. For every content particle, the content-model can specify its occurrence in its parent content particle using regular expression operators (i.e., $?, *, +$). There are also some special cases of the content-model: *EMPTY* for an element with no content particles; *ANY* for an element that can contain any content particles; *PCDATA* for an element that can contain only text; *MIXED* for an element that can contain content particles mixed with text; *CHILDREN* for an element that contains only content particles.

Attributes can be of various types such as *ID* for a unique identifier or *CDATA* for text. They can be optional (*#IMPLIED*) or mandatory (*#REQUIRED*). Optionally, attributes can have a default or a constant value (*#FIXED*).

We model an element type declaration as a tree, denoted as $T = (N, p, l)$, where N is the set of nodes, p is the parent function representing the parent[1] relationship between two nodes, l is the labeling function representing a tuple of node's properties including the node's name and other properties if any.

A node $n \in N$ can be categorized based on its label $l(n)$.

1. **Tag node**: The names of tag nodes appear as tags in the XML documents.

   (a) **Element node**: Each element node n is associated with an element type T. $l(n)$ is a pair in the format of $[Name, CMType]$ where $Name$ is T's name and $CMType$ is the T's content model's type, i.e., *CHILDREN*, *MIXED*, *PCDATA*, *EMPTY*, *ANY*.

   (b) **Attribute node**: Each attribute node n is associated with an attribute type T defined within an element type. $l(n)$ is quadriple in the format of $[Name, Type, Def, Val]$ where $Name$ is T's name, $Type$ is T's data type (e.g., CDATA, ID, IDREF, IDREFS, ENUMERATION etc.), Def is T's default property (i.e., *#REQUIRED*, *#IMPLIED*, *#FIXED*, *#DEFAULT*), and Val is T's default or fixed value if any.

2. **Constraint node**: The names of constraint nodes do not appear in the XML documents. The label of a constraint node is a singlet comprising of the node's name.

   (a) **List node**: Each list node n indicates the connector for composing its children to a content particle, that is, by sequence (i.e., $l(n)$ = [","]) or by choice (i.e., $l(n)$ = ["|"]).

   (b) **Quantifier node**: It represents whether its children occur in its parent's content model one or more (i.e., $l(q)$ = ["+"], called as plus quantifier node), zero or more (i.e., $l(q)$ = ["*"], called as star quantifier node), or zero or one times (i.e., $l(q)$ = ["?"], called as qmark quantifier node). The absence of a quantifier node between a non-quantifier child and its non-quantifier parent implies that the child occurs exactly once.

Except the attribute nodes, each node in the model represents a content particle. If for two nodes n_1 and n_2, there is $p(n_1) = n_2$, n_1 represents either a content particle in n_2's content model or an attribute type defined in n_2's element type.

A tree rooted at a node of element type T is called T's *type declaration tree.* We assume in our study, the XML documents conforming to a DTD have the same root element type, we call the root element type's type declaration tree a *DTD tree.* For example, *company* is the root element type in both DTDs in Figure 1

[1] In this paper, we use *parent* and *child* to refer to direct parent and direct child respectively, versus *ancestor* and *descendant.*

and Figure 2. The two DTDs are modeled as DTD trees in Figures 3 and Figure 4[2]. In the following paper, we represent a node in the trees by its name n with a subscript i indicating the number of the DTD it is within, i.e., $[n]_i$.

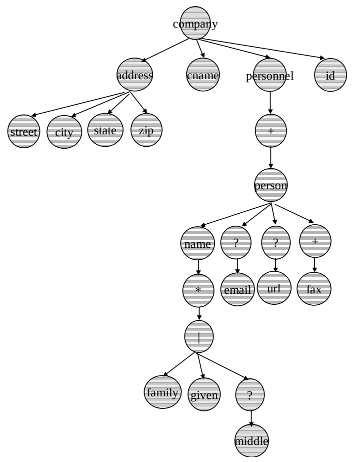

Figure 3: DTD1's DTD Tree

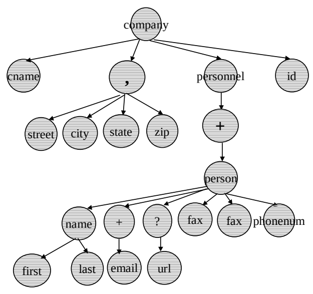

Figure 4: DTD2's DTD Tree

Since each element type declaration is composed of a list of content particles enclosed in a parenthesis followed by a quantifier or not, we do not explicitly model the outermost parenthesis construct as a sequence list node in the DTD trees.

One important aspect we want to emphasize here is that the content model is required to be deterministic. It is an error if an element in the XML document can match more than one occurrence of an element type in the content model. For example, the content model (a?, a?) is is non-deterministic and thus not allowed.

4 Transformation Operations

4.1 Taxonomy of the Transformation Operations

We now give the taxonomy of the transformation operations on the simplified element trees. We describe the taxonomy of the transformation operations in two aspects. One is the change to the tree structure, and the other one is the corresponding DTD semantic change.

1. *add(T, n)*: Add a new subtree T under node n. This corresponds to adding a new content particle T to n's content model.

2. *insert(n, p, C)*: Insert a new node n under node p with n a quantifier node or a sequence list node. C, a subset of p's children, now become n's children.

[2]For simplification, we only mark a node with its name instead of a complete label.

- If n is a quantifier node, it corresponds to changing the occurrence property of a set of nodes C in p's content model from "exactly once" to the one represented by n.
- If n is a sequence list node, the semantics are to put the nodes C in a group, literally i.e., add a parenthesis around the set of content particles C represents.

n can not be an attribute node since attribute node will not have any children. And we do not allow n to be an element node because it may cause undesirable matches. For example, in Figure 5, it is inappropriate to derive $[name]_2$ from $[name]_1$ by inserting a tag node *agency* between *company* and *name* since $[name]_2$ indicates company's agency's name while $[name]_1$ indicates company's name. We would rather first delete *name* and then insert a subtree rooted at *agency* with a leaf *name* to derive DTD2.

```
DTD1:  <!ELEMENT company (name)>
       <!ELEMENT name    (#PCDATA)>
```

```
DTD2:  <!ELEMENT company (agency)>
       <!ELEMENT agency  (name)>
       <!ELEMENT name    (#PCDATA)>
```

Figure 5: Two Example DTDs Showing a Possible Illegal Insertion Relationship

3. *delete(T)*: Delete subtree T. It corresponds to deleting a content particle T from a content model. This is the reverse operation of *add*.
4. *remove(n)*: Remove node n. All n's children now become $p(n)$'s children. This is the reverse operation of *insert*. The constraint that n can only be a quantifier node or a sequence list node also applies here.
5. *relabel(n, l, l')*: Change node n's original label l to l'. The relabelings we allow fall into the following categories:
   - *relabel within the same type*: the relabeling does not change the node's type. We only allow following relabeling:
     - Renaming between two element nodes, two attribute nodes or two quantifier nodes. Note we disallow renaming between a sequence list node and a choice list node.
     - Conversion between an attribute's defaulty type *Required* and *Implied*.
   - *relabel across different types*: the relabeling changes the node's type.
     - Conversion between a sequence list node and an element node with a content model type of *CHILDREN*. This corresponds to expressing a concept by either using a group or encapsulating the group into a new element type. For example, we encapsulate a group composed of *street*, *city*, *state* and *zip* into element type *address* in Figure 3 while the concept of address is modeled as a group composed of *street*, *city*, *state* and *zip* in Figure 4.
     - Conversion between an attribute node with type *CDATA*, default type *#REQUIRED*, no default or fixed value and a *#PCDATA* element node whose parent is a tag node.

– Conversion between an attribute node of default property *#IMPLIED* and a *#PCDATA* element node with a qmark quantifier parent node whose parent is a tag node. Note this is actually a conversion between one node and two nodes. But since an attribute's occurrence in the element within which it is defined is modeled as its property while the occurrence of an element within its parent element is explicitly modeled as a quantifier node, we do not break down this conversion into a composition of a deletion (converting from element to attribute) or adding (converting from attribute to element) of a quantifier node and a relabeling (we do not allow such a standing alone relabeling) operation. We take this as a special complex relabeling. For example, In Figure 6, DTD2 can be transformed from DTD1.

```
DTD1:  <!ELEMENT company (EMPTY)>                 DTD2:  <!ELEMENT company (license?)>
       <!ATTLIST company license CDATA #IMPLIED>         <!ELEMENT license (#PCDATA)>
```

Figure 6: Two Example DTDs Showing a Conversion Cross Node Types

6. $unfold(T, < T_1, T_2, ..., T_i >)$: Replace subtree T with a sequence of subtrees $T_1, T_2, ..., T_i$. We have constraints on the operands. T must root at a repeatable quantifier node r. If r has only one child, construct T' as an isomorphic tree of the subtree in T without r; if r has more than one child, then construct T' as a tree rooting at a sequence list node where the children forest of the root are isomorphic to that of r. T_1, T_2, ..., and T_i satisfy that: (1) they are adjacent siblings; and (2) themselves or their subtrees without the optional quantifier root node are isomorphic. This corresponds to explicitly expressing a repeatable content particle in the format of a sequence of non-repeatable content particles. For example, *phone+* in DTD1 is unfolded to *phone?, phone* in DTD2.

```
DTD1: <Element student (phone+)>
DTD2: <Element student (phone?, phone)>
```

7. $fold(< T_1, T_2, ..., T_i >, T)$: This is the reverse operation of *unfold*.

8. $split(s1, < t1, t2 >)$: $s1$ is split to $t1$ and $t2$ with $s1$ a sequence list node, $t1$ a star quantifier node and $t2$ a choice list node. In DTD, there is no operator to create unordered sequences, it requires (a, b)|(b, a) in order to encode a tuple <a, b>. However, due to the exponential increase of the content particles with the number of elements in the tuple, people tend to use $(a|b)^*$ instead to encode the tuple. This operation then corresponds to drop the order constraint between content particles.

9. $merge(< s1, s2 >, t1)$: $n1$ and $n2$ are merged to a single node $n3$ with $n1$ a star quantifier node, $n2$ a choice list node and $n3$ a sequence list node. This is the reverse operation of $split(s1, < t1, t2 >)$.

4.2 Constraints on the Transformation

When we ensure that the atomic operation reflects common, intuitive transformation, some combination of operations may result in an nonintuitive transformation that violates our intention. For example, suppose we have an element declaration in DTD1: $<!Element\ a\ (b, c, d) >$, and we have DTD2: $<!Element\ a\ (e, d) >$ $<!Element\ e\ (b, c) >$, we can derive DTD2 from DTD 1 by first inserting a list group node above b and c, and then relabeling the group node to tag node e. It is equivalent to directly insert a tag node e above b and c which is an operation we forbid. For another example, suppose we have DTD1: $<!Element\ a\ (b^+) >$, $<!Element\ b\ (c, d) >$ and DTD2: $<!Element\ a\ (b1, b2) >$, $<!Element\ b1\ (c, d) >$, $<!Element\ b2\ (c, d) >$. We disallow the transformation that unfolding subtree representing b^+ to two subtrees representing $b1$, and then applying renaming $b1$ to $b2$. So we impose the following constraints:

Constraint. A node cannot be operated on directly more than once besides the following exceptions:
Exception 1. *unfold* following or followed by *relabel*.
Exception 2. *relabel* performed between attribute and element following or followed by deletion/addition of qmark quantifier node.

For example, in Figure 2, Exception 1 allows deriving DTD2 from DTD1 by such a transformation script: (1) relabel the parent quantifier node of *phone* from star to plus; (2) unfold (*phone+*) to (*phone, phone*). In Figure 3, Exception 2 allows transforming DTD1 to DTD2 by: (1) relabel the attribute node *license* in DTD1 to an element node *license* with a parent qmark quantifier node (we then have *company (license?)*); (2) remove the parent qmark quantifer node of *license*.

```
DTD1: <!Element student (phone*)>
DTD2: <!Element student (phone, phone)>
```

Figure 7: DTD Pair 1

```
DTD1: <!ELEMENT company (EMPTY)>
      <!ATTLIST company license CDATA #IMPLIED)
DTD2: <!ELEMENT company (license)>
```

Figure 8: DTD Pair 2

5 Cost Model

Since there are many different ways to combine these operations to transform one tree to another, we also define a cost model to help select among alternate transformations.

Data capacity. Relative information capacity [Hul84] has been used to measure whether an underlying semantic connection exists between database schemata. Two schemata are equivalent if and only if there is one-to-one mapping between a data instance in the source schema and the target one. We prefer a mapping that will introduce as less discrepancy of data instance as possible.

We assume that the DTDs in our study is flat [LSS99], i.e., the component of the schema (e.g., element and attribute) are all names rather than values. Hence we only consider *PCDATA* and attribute values in XML documents as data. We refer data capacity of an XML document to the collection of all its data.

Data capacity gap. We call transformation operations that result in the loss of data *data capacity reducing* (*DC-Reduce*), e.g., delete. Correspondingly, we call operations that result in the addition of data *data* *capacity increasing* (*DC-Increase*), e.g., add. Operations that result in neither the loss nor addition of data are called *data capacity preserving* (*DC-Preserve*), e.g., merge. However, for some operations, it is difficult to determine from the DTDs alone whether the transformation will result in a loss, addition, or preservation of data capacity. For example, the operation *remove quantifier node* <"*"> changes the content particle from *non-required* to *required* which may cause an increase in data. It also changes the content particle from *repeatable* to *non-repeatable* which may cause data reduction. Hence reducing, increasing or preserving of data capacity are all possible and depend on the individual source XML document. We call those ambiguous transformations *data capacity gap ambiguity* (*DC-Ambiguous*). We use $DC(op)$ to denote the cost that the data capacity gap of the operation *op* contributes to *op*'s overall cost.

We have a heuristic here that information loss is expensive since it is not able to reconstruct the source document from the target document. We thus rank the operations by their data capacity gap cost from low to high in the order of: *DC-Preserve, DC-Increase, DC-Ambiguous and DC-Reduce.*

Operation	Type of Data capacity Change
add	DC-Increase
delete	DC-Reduce
insert	DC-Preserve
remove sequence list node	DC-Preserve
remove quantifer node	DC-Ambiguous
fold	DC-Preserve
unfold	DC-Ambiguous
merge	DC-Reduce
split	DC-Preserve
across type relabel	DC-Preserve

Table 1: Operation's property of data capacity change

Operation of Relabel within Same Type	Type of Data capacity Change
rename	DC-Preserve
relabel attribute type from Required to Implied	DC-Preserve
relabel attribute type from Implied to Required	DC-Ambiguous
rename "*" to "+"	DC-Ambiguous
rename "*" to "?"	DC-Ambiguous
rename "+" to "*"	DC-Preserve
rename "+" to "?"	DC-Ambiguous
rename "?" to "*"	DC-Preserve
rename "?" to "+"	DC-Ambiguous

Table 2: Operation *relabel* within the same type

Potential data capacity gap. Although some transformations are data capacity preserving, there may still be a potential data capacity gap between document conforming to the source DTD and one conforming to the target DTD. For example, the operation *insert a quantifier node* <"+"> is a *DC-Preserve* transformation. However, it changes its children content particles' occurrence property from *non-repeatable* to *repeatable.* Hence the DTD after transformation allows the XML documents to accommodate more data in the future.

We use $PDC(op)$ to denote the cost of the potential data capacity gap contributes to operation op's overall cost. Then we define $PDC(op) = w_{required} * required\_changed(op) + w_{repeatable} * repeatable\_changed(op)$, where $required\_changed(op)$ and $repeatable\_changed(op)$ are two boolean functions that indicate whether the properties "required" or "repeatable" of the content particles that are operated on by op are changed or not. Weights $w_{required}$ and $w_{repeatable}$ indicate the importance of the change of the corresponding property to the potential data capacity.

Scale of operands. The number and size of operands of some operations may have impact on the change of data capacity or potential data capacity. For instance, the operation of merging a smaller set of non-repeatable content particles to a repeatable content particle causes a huger potential data gap than that of merging a larger set. We use $Sca(op)$ to denote the cost the scale of operands of the operation op contributes. The operations whose operands' scale matters are:

- $add(T, n)$, $delete(T)$: $Sca(op)$ is proportional to T's leaf nodes' size.
- $fold(< T_1, T_2, ..., T_i >, T)$, $unfold(T, < T_1, T_2, ..., T_i >)$: $Sca(op)$ is proportional to T_1's leaf nodes' size scaled by i.

We then have, $Cost(op) = (DC(op) + PDC(op)) * Sca(op)$

6 Generation of Simplified Element Tree Matches

6.1 Simplified Element Trees

In our scenario, the documents exchanged between two E-services belong to the same domain, using uniform terms if ontology or some naming standards are provided. It is thus fairly reliable to use name affinity as the first heuristic indicator of a possible semantic relationship between two tag nodes. For example, in Figure 2, we know two root nodes match. Both of the roots have a child node labeled *address*, without looking at their descendants, we match these two nodes. We can derive the matching between two *address* nodes' descendants by comparing two *address*'s type declaration trees separately. However suppose in DTD2, *addr* is used instead of *address*, we look further at their descendants to decide whether to match them.

Based on this idea, we introduce the notion of a *simplified element subtree.* When two DTDs are provided, we say a tag node is *non-rename-able* if there exist tags in the other DTD with the same name. A simplified element tree of element type T, denoted as $ST(T)$, is a subtree of T's type declaration tree $T(E)$ that roots at $T(E)$'s root with each branch ending at the first non-rename-able node reached. In Figure 2 (1) (2), the subtrees within the dashed line are type *company*'s simplified element trees. Figure 2 (3) (4) show the simplified element trees of *contactinfo* in two DTDs respectively.

6.2 Outline of Matching Algorithm

Suppose T_1 is the source simplified element tree and T_2 is the target simplied element tree, we call nodes in T_1 *source nodes* and nodes in T_2 *target nodes*. If n_1 and n_2 are a source node and a target node respectively, we apply the algorithm *matchPropagate* to n_1 and n_2. It produces a transformation script, i.e., a sequence of operations that transforms the subtree rooted at n_1 to the subtree rooted at n_2. The cost of the script is then the cost of matching n_1 and n_2.

rootMatchPropagate is an XML-structure-specific tree matching algorithm. General unordered tree matching is a notoriously high complexity NP problem. As we have mentioned in Section 2.2, standard unordered tree matching techniques [Zha96] assume that relabeling is always preferable to deleting a node and inserting a new one. However, in our domain, labels are critical to tree matching; only certain kinds of relabeling are allowed. The assumption for the standard unordered tree matching does not necessarily hold here and thus those techniques do not apply to our scenario.

Also based on common design patterns, an element type declaration will not be deeply nested. A survey of real world DTDs [Sah00] analyzes 65 DTDs available at XML.ORG and computes the depth of content models. The depth of content models is defined as: 0 for $EMPTY$; 1 for a single element, a sequence or a choice; ...; n for an alternation [3] of sequences and choices of depth n. The maximum depth of the content models are almost around 2 and 3 (the average depth is even lower). This is because complex regular expressions are not advisable. It is difficult to understand and it is almost always the case that the complex expression can be rewritten by some simpler ones. According to this design principle, if a node n_1 has a matching partner n_2, it is highly likely that n_1 and n_2 have a similar depth in the trees rooted at their nearest matching ancestors. For this reason, we can use a non-exhaustive search strategy to produce a satisfactory result.

To derive the transformation from the subtree rooted at n_1 and the subtree rooted at n_2, for n_1's each child m_1, we attempt to find a matching partner m_2. If a node is removed or deleted, we say the node matches special node Φ_1 or Φ_2 respectively. This matching discovery is done in two passes. In pass 1, we visit each child m_1 of n_1 sequentially and compare it with a certain set of target nodes. We call the set of nodes that will be compared with the current source node *matching candidate set*. The following gives the source node's type and the conditions of which a target node will be put into the matching candidate set if satisfying one in Pass 1.

By recursively applying *matchPropagate* to m_1 and each node s in S, we find a node k with the least matching cost c. We have a control strategy to decide whether to match m_1 with k. In pass 1, we apply *delay-match* scheme which disallows matching m_1 to k if c is not low enough, i.e., less than the cost of deleting m_1.

After visiting all children of n_1, we begin pass 2 and traverse all unmatched children of n_1 again, comparing

[3](a, (b|(c, d))) has depth 3.

Source Node's Type	Conditions for Matching Candidate
element	element node on the same level
attribute	attribute node on the same level
choice list	choice list node on the same level
sequence list	sequence list node on the same level or one level deeper Φ_1
quantifier	quantifer node on the same level or one level deeper Φ_1

Table 3: Choosing Matching Candidate Set in Pass 1

them with possible candidates. Table shows the criterion for choosing matching candidate set.

Source Node's Type	Conditions for Matching Candidate
element	element node on the same level or one deeper level list sequence node on the same level attribute node on the same level
attribute	element node on the same level
choice list	choice list node on same level or one deeper level
sequence list	sequence list node on the same level or one level deeper Φ_1 quantifer node on the same level
quantifier	quantifier node on the same level or one level deeper Φ_1 sequence node on the same level

Table 4: Choosing Matching Candidate Set in Pass 2

Again, we apply *matchPropagate* to m_1 and each node s in S and find a node k with the least matching cost c. A *must-match* scheme is applied versus *delay-match* in pass 1. m_1 would be matched to k if c is less than the cost of deleting m_1 and adding k.

When all children of n_1 have a partner, the transformation operations for matching n_1 and n_2 are then composed of those for matching n_1's child m_1 and m_1's partner.

We assume that two DTDs' root element types R_1 and R_2 match. We apply the algorithm *matchPropagate* to the roots of R_1 and R_2's simplified trees to propagate the matches down the tree. matches between the name-match nodes of element types E_1 and E_2 respectively may be found. Then we recrusively apply *matchPropagate* algorithm to E_1 and E_2's simplified trees until no new name-match node matches are generated.

```
MatchPropagate(n1, n2)
{
 //preprocessing for discovery of fold or unfold operations
 For each child tag node m1 of n1
  If there are neighbor siblings which have the same names
   Merge the sequence of unrepeatable tag nodes into a repeatable tag node

 //Pass 1
 For each child tag node m1 of n1
  {
    If there is a child tag node m2 of n2 whose element type
     (1) has the same name or a synonym name of m1's element type's name
or   (2) has been matched with m1's element type before
       Match m1 with m2;
    }

 For each child choice list node m1 of n1
  {
    Find a node m2 which is a child choice node of n2
      associated with a least matching cost c_Intact;
    If c_Intact is less than the cost of deleting m1
      Match m1 and m2;
  }

 For each child sequence list node m1 of n1
  {
    //intact matching
    Find a node which is a child sequence list node m2 of n2
     and associated with a least matching cost c_Intact;

    //corresponding to an operation of  "insert a node above m1"
    Find a node which is a child sequence list node of n2's child
     and associated with a least matching cost c_Insert;

    //corresponding to an operation of "remove m1"
    c_Remove = 0;
    For each child node o1 of m1
    {
      Find a node which is a child node of n2, of the same type as o1,
       and associated with a least matching cost c_ChildIntact
      c_Remove = c_Remove + c_ChildIntact
    };

    Choose the smallest one cmin from c_Intact, c_Insert, c_Remove
      and the corresponding partner m2'.

    If cmin is less than the cost of deleting m1
     Match m1 and m2'.
 }

 For each child quantifier node m1 of n1
  {
    Find a node m2 which is a child node of n2,
     of type quantifier node or sequence list node,
      and associated with a least matching cost c
    If c is less than the cost of deleting m1
     Match m1 and m2;
  }

 //Pass 2
 For each unmatched child tag node m1 of n1
  {
    If it is a name-match node // a node can not be renamed
     {
```

```
    If there is an unmatched node o2 which is a child of m2 and
      has the same name/synonym
      or their element types have been matched before
    //corresponding to an insert operation
         Match m1 and o2;
    }
   Else //it is not a name-match node
     {
       Find a node m2 which is a child sequence list node
        or a non-name-match child node of n2
         and associated with a least matching cost c;
       If c is less than deleting m1 and adding m2
          Match m1 and m2.
     }
  }

For each unmatched child choice node m1 of n1
 {
  //intact matching
  Find a node m2 which is a child choice list node of n2
   and associated with a least matching cost c_Intact;

  //corresponding to an operation of "insert a node above m1"
  Find a node o2 which is a child choice node of n2's child sequence node
   and associated with least matching cost c_Insert;

  Choose the smallest one cmin from c_Intact, c_Insert
   and the corresponding partner m2'.

  If cmin is less than the cost of deleting m1 and adding m2'.
         Match m1 and m2'.
 }

For each unmatched sequence, quantifier child node m1 of n1
 {
  //intact matching
  Find a node m2 which is a child sequence or quantifier node of n2
   and associated with a least matching cost c_Intact;

  //corresponding to an operation of "insert a node above m1"
  Find a node o2 which is a child sequence
   or quantifier node of n2's child sequence list node,
   and associated with least matching cost c_Insert;

  //corresponding to an operation of "remove m1"
   c_Remove = 0;
   For each child node o1 of m1
    {
     Find a node which is a child node of n2, of the same type as o1,
       and associated with a least matching cost c_ChildIntact
        c_Remove = c_Remove + c_ChildIntact
     };

  Choose the smallest one cmin from c_Intact, c_Insert, c_Remove
```

```
        and the  corresponding partner m2'.

    If cmin is less than the cost of deleting m1 and adding m2'.
        Match m1 and m2'.
}
```

6.3 Matching Example DTDs

We now describe how the match discovery between DTD 1 and DTD 2 depicted in Figures 3 and 4 would be done by our system. We will use the same settings as shown in the examples in Section 4.2.

As shown in Figures 3 and 4, there are four pairs of simplified element trees, i.e., *company*, *personnel*, *person* and *name*. We apply *DMatch* to the root type *company*'s simplified element trees first. We traverse $<company>_1$'s children one by one. For $<address>_1$, its matching candidate set is empty since all the element nodes on the same level (i.e., 2) are non-rename-able. For $<cname>_1$, its matching candidate set contains only $<cname>_2$. Since they have the same name, they are matched. Similarly, $<personnel>_1$ is matched against $<personnel>_2$. For attribute $<id>_1$, its matching candidate set is empty. In pass 2, $<address>_1$'s matching candidate set contains only $<,>_2$. We apply *DMatch* to them and derive the transformation script composed of an operation of relabeling "*address*" to ",". As illustrated in Section 4, $<address>_1$ will be mapped to $<,>_2$. Attribute $<license>_1$'s matching candidate set now contains element $<license>_2$. And with the parameter setting, they will be matched. Now each of $<company>_1$'s children has a partner. Hence we are done with matching element type *company*.

We continuously apply *DMatch* to the element simplified trees of each pair of element type matched by name, i.e., *personnel*, *person* and *name*. In this way, all matches between them are discovered.

7 Generation of XSLT for Transforming Documents

Based on the established semantic relationship between two DTDs, we use XSLT [Gro], a language designed for transforming individual XML documents, to specify and then execute the transformation. XSLT uses XPath [W3C99] expressions to specify exactly which nodes in the XML documents are operated on. Each node n in the DTD tree is associcted with a set of nodes in the XML tree which can be specified by an XSLT expression. We call this XSLT expression *n's XSLT expression.*

For each element type matching, i.e., the two roots of the simplified element trees associated with the element types match, the XSLT generator generates a named template. It then will traverse the target simplified element tree in a width-first manner. The following gives what kind of XSLT expressions will be generated based on the node it is now visiting. We use Figure 1 and Figure 2 as our running examples. Appendix A shows the XSLT scripts that will be generated for transforming XML documents conforming to DTD1 in Figure 1 to XML documents conforming to DTD2 in Figure 2. We will use Figure 5 as our running example to show how the generator works.

1. element node:

(a) the element type is associated with a template. We define a named template for an element types in target DTD if it is associated with a simplified element tree pair. in source DTD. If the associated template has not been defined yet, generate the template as well.

Example 1 *In Figure 1 and 2, element type* person *in DTD1 matches* person *in DTD2 and then there is a named template* person-trans *defined for deriving target instances of element type* person *from source instances of* person. *Once the generator reaches the tag node with name* person, *it will generate the following XSLT expressions:*

```
<person>
  <xsl:call-template name = "person-trans"/>
</person>
```

If named template person-trans *has not been defined yet, the template will be then generated. It is similar to that to generate expressions for element type which is not associated with a template. The only difference that a special head which is always at the beginning of a template will be generated.*

(b) the element type is not associated with a template yet:

generate the tag of the element type and recursively apply the algorithm to its children. If this element node is of type *#PCDATA*, XSLT expressions *¡xsl:value-of¿* will be generated.

Example 2 *Element type* name *is associated with a named template* name-trans. *To generate this template, the generator traverses* name*'s simplified element trees which is composed of the root of element type* name *itself and two children leaf tag nodes of type* first *and* last. *The following scripts will be generated.*

```
<xsl:template match = "name"  name = "name-trans">
 <first>
   <xsl:value-of select="given"/>
 </first>
 <last>
   <xsl:value-of select="family"/>
 </last>
</xsl:template>
```

2. attribute node: generate the attribute with the tag along with the node's XSLT expression.

3. list node:

(a) sequence list node:

does not generate any XSLT expression.

(b) choice list node:

generates *making choices* XSLT expression, i.e., $< xsl : if >$. Since choice list node indicates that one branch of this node's children will be choosen, $< xsl : if >$ will change the output based on the input.

4. quantifier node: suppose quantifier node n has a matching partner n'. We have mentioned before that an absence of quantifer node between two non-quantifer nodes in DTD indicates that the content

particle represented by the child node appears exactly once in the content model of the content particle represented by the parent node. We can take matching Φ_1 to n (i.e., inserting n) as matching an implict quantifier node whose properties are *required* and *non-repeatable* to n.

(a) if changing from n' to n is a *data capacity preserving* transformation (refer to table):

generates *processing multiple elements* XSLT expression, i.e., $< xsl : for - each >$. In the *select* clause, it selects all the nearest decendant tag nodes of n'. For each such a selected tag node, $< xsl : if >$ with the test condition of deciding what element type the the input node is of is generated. Based on the element type, the algorithm is recursively applied (refer to item 1).

Example 3
```
<xsl:for-each select = "person">
  <xsl:if test = "(local-name() = 'person')">
   <person>
     <xsl:call-template name = "person-trans"/>
   </person>
  </xsl:if>
 </xsl:for-each>
</xsl:template>
```

(b) if the transformation of changing from n' to n changes the property of "required" from *not required* to *required* or from *countable-repeatable* to *countable-repeatable* with an increasing repeating number but does not change the property of "repeatable" from *repeatable* to *non-repeatable* or *countable-repeatable*, this may encounter a situation where at least one target XML data node in target XML document is required to be instantiated while its data source, a corresponding source XML data node, is not provided. In such circumstance, the generator will generate $< xsl : if >$ to test whether the source data is available. If not, tags with a content which is a message to remind that additional data is needed there.

Example 4 *Content particle* email* *in element type* person *is changed to* email+. *The following XSLT script will be generated.*

```
<xsl:if test = "(count(email)=0)">
  <email>
   value needed here
  </email>
</xsl:if>
<xsl:for-each select = "email">
  <xsl:if test = "(local-name() = 'email')">
   <email>
     <xsl:apply-templates/>
   </email>
  </xsl:if>
</xsl:for-each>
```

(c) if the transformation of changing from n' to n changes the property of "repeatable" either (1) from *repeatable* or *countable-repetable* to *non-repeatable*, or (2) from *countable-repeatable* to *countable-repeatable* with a decrease of the repeating number, and if the transformation does not change the property of "required" from *not required* to *required*, this may encounter a situation where only a subset of multiple data source are needed to instantiate the target XML data nodes. Then the *select* clause is slightly different from the routine expression $< xsl : for - each >$ generated for quantifer node current visited. We by default instantiate the target XML data nodes by assigning the value from the first several source XML data nodes among all the available source XML data nodes.

Example 5 *suppose in DTD2, in* person*'s content model, content particles* fax, fax *are replaced by* fax*, then at most one XML data node of type* fax *can be present. The following XSLT scripts will be generated.*

```
<xsl:for-each select = "fax[position()=1]">
  <xsl:if test = "(local-name() = 'fax')">
   <fax>
     <xsl:apply-templates/>
   </fax>
  </xsl:if>
</xsl:for-each>
```

(d) qmark node: if the transformation of changing from n' to n changes the properties from *not required* to *required* and either (1) from *repeatable* or *countable-repeatable* to *non-repeatable*, or (2) from *countable-repeatable* to *countable-repeatable* with a changing repeating number, this may encounter a situation that is the compound of the two scenarios discussed in, both the expressions for generating tags with reminding message and selecting the first available data source will be generated.

Example 6 *Content particle* fax+ *in element type* person *in DTD1 is changed to* fax, fax *in DTD2. The first* fax *can be derived, but the second* fax *is not ensured to have data source. The following XSLT script then will be generated.*

```
<xsl:for-each select = "fax[position()=1|position()=2]">
  <xsl:if test = "(local-name() = 'fax')">
   <fax>
     <xsl:apply-templates/>
   </fax>
  </xsl:if>
</xsl:for-each>
<xsl:if test = "(count(fax)=1)">
  <email>
   value needed here
  </email>
</xsl:if>
```

8 Implementation and Experimentation

Experimental Setup In order to evaluate our solution, we have implemented a prototype system, *EXTRA* (E-business Xml document TRAnslation), using Java, IBM XML4J parser and IBM LotusXSL. We have run experiments evaluating the precision of the DTD matching. As data sets, we selected a DTD *Journalist.dtd* from XML.org DTD repository [Org98] and *DocBook.dtd* from Oasis [Org01]. *Journalist.dtd* is loosely based on *DocBook.dtd.* It started with elements in *DocBook.dtd* but some content models have been simplified and some parts of the hierarchy have been flattened. In addition, new element types are added and renaming has occurred. Most importantly, *Jounalist.dtd* provides a readme document that lists changes made to *DocBook.dtd.* Thus we can test our system on these real changes.

Due to the absence of an ontology in this domain, we use WordNet [CSL], a lexical database providing a synonym dictionary, identifying synonym renaming such as between "remark" and "comment". We use a *longest matching subsequence* [oM01] scheme to measure the similarity of two strings. For example, the longest matching subsequence of strings "DocInfo" and "PrefaceInfo" is "Info". We normalize this subsequence's length (e.g., 4) by the longer length of the two strings (e.g., 11), deduct it from 1, and scale it by a parameter (e.g., 1.5) to calculate the renaming cost (1-4/11)* 1.5 = 0.9.

Experimentatal Studies To evaluate our algorithms, we compare the discovered transformation scripts with the documented real transformation scripts. We use two settings of parameters, illustrating how the tuning of the parameters can affect the discovery of matches. In setting 1, the cost of each data capacity gap category ranks from lower to higher in the order of *DC-Preserve* (0.25), *DC-Increase* (0.5), *DC-Ambiguous* (0.75) and *DC-Reduce* (1.0). We assign 0.5 to both potential data capacity gap parameters $w_{required}$ and $w_{repeatable}$. The scale parameter for string similarity comparison is 1.5. In above example, this allows renaming a leaf node "DocInfo" to another leaf node "PrefaceInfo" since the cost (0.9) is lower than that of deleting a leaf node (1.0).

In parameter setting 2, we assign 1.0 to all data capacity gap parameters while all the other parameters are kept the same. In table 1, for each kind of transformation operations, we show its occurrence number in the real change scripts, the discovered change scripts and the number of the discovered changes consistent with the real changes with two parameter settings respectively. We can see parameter setting 2 produces suboptimal scripts while parameter setting 1 produces precise scripts. This is because our algorithm assumes the order of *DC-Preserve*, *DC-Increase*, *DC-Ambiguous*, *DC-Decrease* in terms of their costs from low to high. When this is violated, suboptimal transformation scripts may arise. For example, with parameter setting 2, *DC-Decrease* is no more expensive than other operations. Therefore deleting a node may be chosen rather than attempting to match this node. However, we can see the result is still acceptable and close to the real change.

9 Conclusion

We have addressed two problems in this work: First, how to automate the identification of semantic relationships between XML-based documents. Second, how to leverage this knowledge to transform an XML-based document from a given schema to a different, yet related, schema. This work is unique because we incorporate domain-specific characteristics of the XML documents, such as domain ontology, common transformation types, and specific DTD modeling constructs such as quantifiers and type-constructors. This allows us to

		Match	Rename	Insert	Remove	Add	Delete
Real Changes		15	3	1	1	1	7
Setting 1	Changes Discovered	15	3	2	1	1	7
	Correct Changes	15	3	2	1	1	7
Setting 2	Changes Discovered	12	2	1	1	1	5
	Correct Changes	12	2	1	1	0	4

Table 5: Comparison of Precision of Discovered Changes with Different Parameter Settings

avoid the high level of user interaction as well as complexity required by other approaches.

As XML-Schema emerging as a potential schema standard for describing XML documents, in the future we will investigate how to adapt our approach to exploit the richer treatment of types offered by XML Schema as additional hints of similarity.

References

[BCV] S. Bergamaschi, S. Castano, and M. Vincini. Semantic Integration of Semistructured and Structured Data Sources.

[BCVV98] S. Bergamaschi, S. Castano, S. De Capitani Di Vimeracati, and M. Vincini. An intelligent approach to information integration. In *Int. Conference on Formal Ontology in Information Systems*, 1998.

[Biz01] BizTalk. Enabling software to speak the language of business. http://www.biztalk.org/, 2001.

[BKGD00] D. Bell, H. Kuno, J. Gustafson, and A. Durante. A Business Communication Specification for E-Services. In *HP Laboratories Technical Report*, 2000.

[CA99] S. Castano and V. D. Antonellis. A schema analysis and reconciliation tool environment for heterogeneous databases. In *Int. Database Engineering and Applications Symposium (IDEAS-99)*, 1999.

[CGM97] S. S. Chawathe and H. Garcia-Molina. Meaningful Change Detection in Structured Data. In *SIGMOD*, 1997.

[CRGMW96] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *SIGMOD*, 1996.

[CSL] Princeton University Cognitive Science Lab. WordNet - a Lexical database for English. http://www.cogsci.princeton/ wn.

[cXM01] cXML. Commerce XML Resources. http://www.cxml.org/, 2001.

[DDL00] A. Doan, P. Domingos, and A. Levy. Learning source descriptions for data integration. In *WebDB International Workshop on the Web and Databases*, pages 81–86, 2000.

[Gro] W3C XSL Working Group. XSL Transformations (XSLT). http://www.w3.org/TR/xslt/.

[Hul84] Richard Hull. Relative Information Capacity. In *Association for Computing Machinery*, pages 97–109, 1984.

[Kun00] H. Kuno. Surveying the E-Services Technical Landscape. In *Second International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, 2000.

[LSS99] L.V.S. Lakshmanan, F. Sadri, and I.N Subramanian. On Efficiently Implementing SchemaSQL on a SQL Database System. In *vldb*, 1999.

[MIR93] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Int. Conference on Very Large Data Bases*, pages 120–133, 1993.

[MS92] V. M. Markowitz and A. Shoshani. Representing extended entity-relationship structures in relational database. In *ACM Trans on DB Systems, 17(3)*, pages 423–464, 1992.

[MZ98] T. Milo and S. Zohar. Using schema matching to simplify heterogenous data translation. In *SIGMOD*, 1998.

[oM01] University of Maryland. String similarity. http://www.cs.umd.edu/Outreach/hsContest98/questions/node5.html, 2001.

[Org98] XML Org. XML.Org Registry Open for Business. www.xml.org/registry, 1998.

[Org01] Oasis Org. DocBook 3.1 XML. http://www.oasis-open.org/docbook, 2001.

[PSU98] L. Palopoli, D. Sacca, and D. Ursino. Semi-automatic, semantic discovery of properties from database schemes. In *Int. Database Engineering and Applications Symposium (IDEAS-98)*, 1998.

[Ros01] RosettaNet. Lingua franca for eBusiness. http://www.rosettanet.org, 2001.

[RR87] A. Rosenthal and D. Reiner. Theorectically Sound Transformations for Practical Database Design. In *Proc of Entity Rlationship Conf.*, pages 115–131, 1987.

[Sah00] Arnaud Sahuguet. Everything you ever wanted to know about DTDs, but were afraid to ask. In *WebDB International Workshop on the Web and Databases*, pages 69–74, 2000.

[SWZS90] D. Shasha, J. Wang, K. Zhang, and F. Shih. Fast algorithms for the unit cost editing distance between trees. In *Journal of Algorithms*, pages 581–621, 1990.

[W3C98] W3C. XML^{TM} . http://www.w3.org/XML, 1998.

[W3C99] W3C. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, 1999.

[W3C01] W3C. *XML Schema* . http://www.w3.org/XML/Schema, 2001.

[Zha96] K. Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, pages 205–222, 1996.

[ZS89] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, pages 18(6):1234–1262, 1989.

[ZWS95] K. Zhang, J. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, 1995.

A XSLT Examples

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match ="/*">
<company>
<cname>
  <xsl:apply-templates select = "cname"/>
</cname>
<street>
  <xsl:apply-templates select = "address/street"/>
</street>
<city>
  <xsl:apply-templates select = "address/city"/>
</city>
<state>
  <xsl:apply-templates select = "address/state"/>
</state>
<zip>
  <xsl:apply-templates select = "address/zip"/>
</zip>
<personnel>
  <xsl:apply-templates select = "personnel"/>
</personnel>
</company>
</xsl:template>

<xsl:template match="cname//*|@*|comment()|processing-instruction()|text()">
  <xsl:copy>
   <xsl:apply-templates  select="*|@*|text()"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="street//*|@*|comment()|processing-instruction()|text()">
  <xsl:copy>
   <xsl:apply-templates  select="*|@*|text()"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="city//*|@*|comment()|processing-instruction()|text()">
  <xsl:copy>
   <xsl:apply-templates  select="*|@*|text()"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="state//*|@*|comment()|processing-instruction()|text()">
  <xsl:copy>
   <xsl:apply-templates  select="*|@*|text()"/>
  </xsl:copy>
</xsl:template>

<xsl:template match="zip//*|@*|comment()|processing-instruction()|text()">
  <xsl:copy>
   <xsl:apply-templates  select="*|@*|text()"/>
  </xsl:copy>
</xsl:template>

<xsl:template match = "personnel"  name = "personnel-trans">
  <xsl:for-each select = "person">
    <xsl:if test = "(local-name() = 'person')">
     <person>
       <xsl:call-template name = "person-trans"/>
     </person>
    </xsl:if>
  </xsl:for-each>
</xsl:template>

<xsl:template match = "person"  name = "person-trans">
  <name>
    <xsl:apply-templates select = "name"/>
  </name>
  <xsl:if test = "(count(email)=0)">
   <email>
     value needed here
   </email>
  </xsl:if>
  <xsl:for-each select = "email">
    <xsl:if test = "(local-name() = 'email')">
      <email>
        <xsl:apply-templates/>
      </email>
    </xsl:if>
  </xsl:for-each>
  <xsl:for-each select = "fax[position()=1]">
    <xsl:if test = "(local-name() = 'fax')">
      <fax>
         <xsl:apply-templates/>
      </fax>
    </xsl:if>
  </xsl:for-each>
<phonenum>
  value needed here
</phonenum>
</xsl:template>

<xsl:template match = "name"  name = "name-trans">
<first>
  <xsl:value-of select="given"/>
</first>
<last>
  <xsl:value-of select="family"/>
</last>
</xsl:template>

<xsl:template match="email//*|@*|comment()|processing-instruction()|text()">
  <xsl:copy>
   <xsl:apply-templates  select="*|@*|text()"/>
  </xsl:copy>
```

```
</xsl:template>

<xsl:template match="fax//*|@*|comment()|processing-instruction()|text()">
  <xsl:copy>
   <xsl:apply-templates  select="*|@*|text()"/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```