Automating the Transformation of XML Documents

Hong Su
Computer Science Dept
Worcester Polytechnic Institute
Worcester, MA 01609
suhong@cs.wpi.edu

Harumi Kuno
Hewlett-Packard Labs
Palo Alto, CA 94304
harumi_kuno@hp.com

Elke A. Rundensteiner
Computer Science Dept
Worcester Polytechnic Institute
Worcester, MA 01609
rundenst@cs.wpi.edu

ABSTRACT

The advent of web services that use XML-based message exchanges has spurred many efforts that address issues related to inter-enterprise service electronic commerce interactions. Currently emerging standards and technologies enable enterprises to describe and advertise their own Web Services and to discover and determine how to interact with services fronted by other businesses. However, these technologies do not address the problem of how to reconcile structural differences between similar types of documents supported by different enterprises. Transformations between such documents must thus be created manually on a case-by-case basis. In this paper, we explore the problem of how to automate the transformation of XML E-business documents. We develop an integrated solution that automates as much as possible all steps of the document transformation process. One, we propose a set of schema transformation operations that establish semantic relationships between two XML document schemas. Two, we define a model that allows us to compare the cost of performing these operations. Three, we introduce an algorithm that discovers an efficient sequence of operations for transforming a source document schema into a target document schema based on our cost model. The operation sequence then is used to generate an equivalent XSLT transformation script. Experimental results indicate that our algorithm can satisfactorily discover acceptable transformations.

1. INTRODUCTION

1.1 Motivation

Web services [9] are significantly more loosely coupled than traditional applications. Web services are deployed on the behalf of diverse enterprises, and the programmers who implement them are unlikely to collaborate with each other during development. However, the purpose of web services is to enable business-to-business interactions. Web services should be able to discover new services and interact with them dynamically without requiring developers to update the code of either service. This need is spurring the creation of electronic commerce standards such as ebXML [16] and the Web Services Conversation Language (WSCL) [1] that all support different aspects of inter-enterprise service interactions.

WIDM 2001, Atlanta, GA, USA

However, these technologies do not address the significant problem of how to reconcile structural differences between similar types of documents supported by two different enterprises. For example, let Service A and Service B be services that front different companies. Suppose that these services want to engage in a shopping cart interaction, and that Service B requires Service A to submit a shipping information document. Service A might be able to provide this information, but in a slightly different format than Service B expects. For example, Service A's document might list the address first and Service B's might list it last. Similarly, Service B might call the zip code element "Postal Code" where Service A names it "Zip Code".

Currently, the only recourse of service developers is to create a transformation between the two documents by hand. Manual translation of the XML documents is time consuming and thus especially unacceptable for web services, where the information sources change frequently. Hence applications must evolve quickly. Clearly, tools are needed to automate or at least support this process in as much as is possible.

1.2 State of the Art

Schema Translation. *ARTEMIS* [2, 3] supports the analysis and reconciliation of sets of heterogeneous relational schemas by measuring the similarity of element names, data types, and structures. *Clio* [11, 21] uses reasoning about SQL queries to create initial mappings between relational schemas, then refines these mappings using data examples. However, because relational schemas are flat, neither *Clio* nor *ARTEMIS* can handle hierarchical XML schemas.

TranScm [12] uses schema matching to derive an automatic translation between schema instances. All input schemas are transformed into a common model, namely, labeled graphs. It offers a set of "rules" that describe how to match a component (i.e., a node in the labeled graph) in the source schema with a corresponding component in the target schema. The matching is performed node by node starting at the top. Rules are checked in a fixed order based on their priorities. However since *TranScm* is a system aiming to provide a

general approach, issues such as what special XML matching rules should be provided to the rule base and assignment of priority for each rule would first need to be solved to put the approach in the XML context.

The machine-learning approach taken in [6] attempts to train a learner by a set of user-provided mappings from a data source to the global schema and then discovers the characteristic instance patterns. Given a new data source, one-to-one mappings between the leaf nodes of two schema trees can be established by trying out those learned matches. However it does not match source-schema elements with a hierarchical structure, i.e., the inner nodes in the schema tree, as needed for XML. Furthermore, in cases where example data sets of both source and target XML documents are available, such an approach could not be applied.

Tree Matching. Much work has been done in the area of tree matching. [14] and [22] address the change detection problem for ordered and unordered trees respectively. However, the tree matching problem treats the label of each node as a second class citizen. For example, the cost of relabeling is assumed to be cheaper than that of deleting a node with the old label and inserting a node with the new label. However if we model an XML schema as a tree, some labels of the nodes can be names of the XML tags which carry semantic meaning. A relabel from one node to another semantic unrelated node will cause an undesirable result. Thus the assumption is invalid for the XML domain. We overcome this limitation in our work.

LaDiff [5] adapts a simple cost model in which *insert*, *delete* and *move* are all unit cost operations, i.e., cost is 1. We now refine the cost model to take XML characteristics into account. *LaDiff* also assumes that each node of the input trees has a special label that describes its semantics (semantic tag). For example, a tree representing a document may have tags "paragraph", "section", etc. And for each leaf node in the source tree, there is at most one leaf node in the target tree that is "close" to it (unique close partner). These assumptions facilitate the matching. *MH-Diff* [4] allows flexible cost models and drops the assumptions in [5] but then takes quadratic time in the size of the input.

There are a number of differences between the tree matching problem studied in [5, 4] and the specific problem of matching trees that model XML schemas. First, some of their edit operations such as *copy* and *glue* in [4] are not meaningful for an XML schema. Instead we need XML-schema-specific edit operations. Second, the assumption of unique close partner in [5] does not necessarily hold true. Meanwhile the assumption of semantic label holds for some of the nodes in the XML model. That is, some of the nodes do not have tags describing their semantics (e.g., constraint nodes which will be introduced in Section 2) while others do have them (e.g., tag nodes). Hence it is not possible to only use the assumptions to direct the mapping. Neither is it suitable to completely discard the assumption as [4] has done which will result in a high time complexity.

XML Document Restructuring. [7] studies how to change an XML document in terms of both schema and data. It propose a set of DTD change primitives that can be applied to an old DTD to derive a new DTD and corresponding data change will be made implicitly as well. These primitives, however, do not cover all the XML schema mappings that are most likely to happen.

1.3 The Xtra Approach

Since DTDs are currently the dominant industry standard, we address the problem of how to transform a document conforming to a source DTD so that it will conform to a target DTD. Our approach could easily be adapted to XML Schema [18]. Given a source and a target DTD, we first model each DTD as a tree. This allows us to express the problem as how to transform one *DTD tree* into another. To this end, we have defined a set of *DTD transformation operations* that establish the semantic relationships between two trees. We also define a cost model for choosing a sequence of transformation operations among multiple alternatives. We have developed an algorithm to discover a sequence of operations (i.e., *transformation script*) that transforms a source DTD tree into a target DTD tree. The discovery process is based on provided auxiliary information (e.g., synonym dictionary, domain ontology, etc.) and a cost model we define for choosing a transformation script among multiple alternatives. Lastly, we use the resulting transformation script to generate a eXtensible Stylesheet Language Transformations (XSLT) script [19]. The XSLT script can then be applied to source XML documents to transform them into XML documents conforming to the target DTD. Figure 1 shows the architecture of our system.

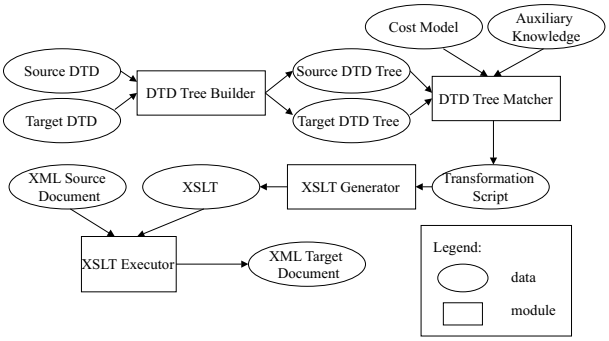

Figure 1: Xtra System Architecture

The primary contributions of our work include:

1. We propose a set of DTD transformation operations that capture common discrepancies between alternative DTD design behaviors for modeling real-world data.

2. We define a cost model (based on the concept of data capacity) for measuring the quality of DTD transformations.

3. We have implemented an XML TRAnslation prototype system (Xtra)[1], and run experiments on both real and synthetic XML document to verify the feasibility of our approach.

[1]Xtra has been demonstrated at ACM SIGMOD 2001.

2. DTD DATA MODEL

Document Type Definition (DTD) [17] describes the structure of XML documents as a list of element type declarations. Elements can in turn have content particles, 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., $a|b$) with both a and b being content particles again. For every content particle, the content-model can specify its occurrence in its parent content particle using regular expression operators (i.e., $?, *, +$). Attributes can be of various types such as *ID*, *CDATA*, etc. 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 relationship between two nodes, and l is the labeling function representing the properties of a node. We categorize a node $n \in N$ based on its label $l(n)$.

- **Tag node**:
  - **Element node**: Each element node n is associated with an element type T. $l(n)$ is a singleton in the format of $[Name]$ where $Name$ is T's name.
  - **Attribute node**: Each attribute node n is associated with an attribute type T. $l(n)$ is quadruple in the format of $[Name, Type, Def, Val]$ where $Name$ is T's name, $Type$ is T's data type (e.g., CDATA, etc.), Def is T's default property (e.g., $\#REQUIRED$, $\#IMPLIED$, etc.), and Val is T's default or fixed value if any.
- **Constraint node**:
  - **List node**: Each list node n indicates how its children are composed, that is, by sequence (i.e., $l(n) = [","]$) or by choice (i.e., $l(n) = ["|"]$).
  - **Quantifier node**: It represents whether its children occur in its parent's content model one or more (i.e., $l(q) = ["+"]$, called plus quantifier node), zero or more (i.e., $l(q) = ["*"]$, called star quantifier node), or zero or one times (i.e., $l(q) = ["?"]$, called qmark quantifier node).

A tree rooted at a node of element type T is called T's *type declaration tree*. We assume each DTD has a unique root element type whose type declaration tree is then the *DTD tree*. For example, Figure 2 shows two sample DTDs of web-service purchase orders. These two DTDs will be used as the running example through the paper. The DTDs are modeled as DTD trees in Figures 3 and 4. For simplicity, we mark each node with its name rather than a complete label. In the following, we represent each node by its name n with a subscript i indicating the number of the DTD it is within (i.e., 1 or 2). The format is then $<n>_i$[2]. Since each element type declaration is composed of a list of content particles enclosed in a parenthes (optionally followed by a quantifier), we do not explicitly model the outermost parenthesis construct as a sequence list node in the DTD trees. For example, $<name>_2$ has two children $<first>_2$ and $<last>_2$ rather than a sequence list node.

[2]If there is duplicate name, another subscript specifying which one it is can be added, i.e., $<n_j>_i$

```
<!ELEMENT company (address, cname, personnel)>
<!ATTLIST company license CDATA #IMPLIED>
<!ELEMENT address (street, city, state, zip)>
<!ELEMENT cname (#PCDATA)>
<!ELEMENT personnel (person)+>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT person (name, email?,url?, fax+)>
<!ELEMENT name (family|given|middle?)*>
<!ELEMENT email (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
<!ELEMENT family (#PCDATA)>
<!ELEMENT given (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
                              (a)
<!ELEMENT company (cname,(street, city, state, postal),
                  personnel,license?)>
<!ELEMENT cname (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT postal (#PCDATA)>
<!ELEMENT personnel (person)+>
<!ELEMENT license (#PCDATA)>
<!ELEMENT person (name, email+, url?, fax, fax?, phonenum)>
<!ELEMENT name (first, last)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT url (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
<!ELEMENT phonenum (#PCDATA)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT last (#PCDATA)>
                              (b)
```

Figure 2: Example DTDs of Web Service A and B's Purchase Orders

3. TRANSFORMATION OPERATIONS

3.1 Taxonomy of Transformation Operations

We identify two primary causes of discrepancies between the DTD components modeling the same concepts: First, the properties of the concepts may differ. For example, *phone number* is required in *contact information* in DTD 2 while it is not required in DTD 1. Second, due to the relatively free-form nature of XML and lack of a standard for DTD design, a given concept can be modeled in a variety of ways. For example, an atomic property can be represented as either a $\#PCDATA$ sub-element or an attribute. We have studied the common DTD design patterns and correspondingly proposed a set of transformation operations, as listed below[3].

1. *add(T, n)*: Add a subtree T under node n, i.e., add a new content particle to element 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 and move a subset of p's children C to become n's children. If n is a quantifier node, the operation changes the occurrence property of the children C in p's content model from "exactly once" to correspond to n. If n is a sequence list node, it groups the nodes C.

   n cannot be an attribute node since an attribute node would not have any children. And we do not allow n to be an element node because it may cause undesirable matches. See Example 1.

[3]We use "child" to refer to direct child versus "descendants".

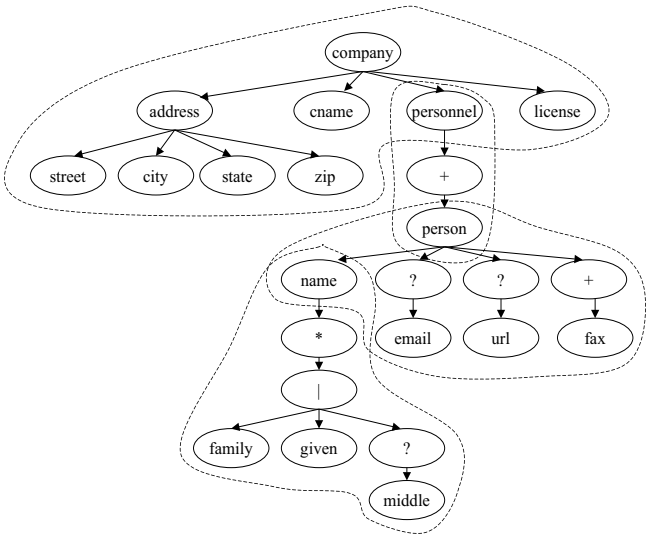

Figure 3: DTD 1's DTD Tree

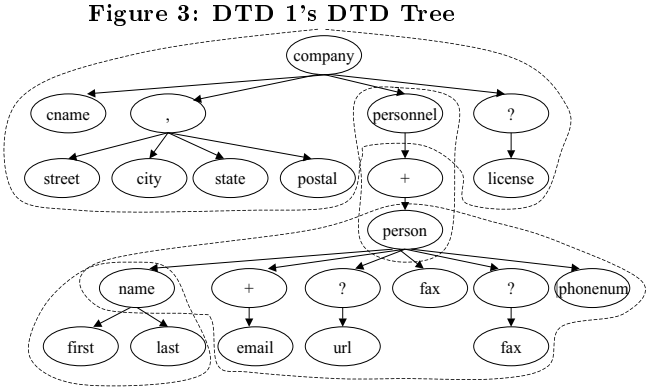

Figure 4: DTD 2's DTD Tree

3. *delete(T)*: Delete subtree T, i.e., delete a content particle T from a content model. This is the reverse operation of *add*.
4. *remove(n)*: Remove node n with n a quantifier or a sequence list node. All n's children now become $p(n)$'s children. This is the reverse operation of *insert*.
5. *relabel(n, l, l')*: Change node n's original label l to l'.
   - *relabel within the same type* (the operation does not change the node's type):

     (a) Renaming between two element nodes, two attribute nodes or two quantifier nodes but not between a sequence list node and a choice list node; (b) Conversion between an attribute's default type $\#REQUIRED$ and $\#IMPLIED$.
   - *relabel across different types* (the operation changes the node's type):

     (a) Conversion between a sequence list node and an element node which has children. This corresponds to using a group instead of an element type or encapsulating the group into a new element type. See Example 2. (b) Conversion between an attribute node with type *CDATA*, default type $\#REQUIRED$, no default or fixed value and a $\#PCDATA$ element node; (c) Conversion between an attribute node of default property $\#IMPLIED$ and a $\#PCDATA$ element node with a qmark quantifier parent node. (b) and (c) are allowed in that people can model a one-to-one relationship property as either an attribute or an subelement. See Example 3.
6. $unfold(T, <T_1, T_2, ..., T_i>)$: Replace subtree T with a sequence of subtrees $T_1, T_2, ..., T_i$. T must root at a repeatable quantifier node. T_1, T_2, ..., and T_i satisfy that: (1) they are adjacent siblings; and (2) they or their subtrees without a qmark quantifier root node are isomorphic. $unfold$ recasts a repeatable content particle as a sequence of non-repeatable content particles. See Example 5.
7. $fold(<T_1, T_2, ..., T_i>, T)$: This is the reverse operation of *unfold*.
8. $split(m_1, <n_1, n_2>)$: A sequence list node m_1 is split into a star quantifier node n_1 and a choice list node n_2. Because there is no DTD operator to create unordered sequences, tuples $<a, b>$ tend to be expressed using the construct $(a|b)^*$ rather than $(a, b)|(b, a)$. This operation corresponds to converting an ordered sequence to an unordered one. See Example 5.
9. $merge(<n_1, n_2>, m_1)$: n_1 and n_2 are merged into a single node m_1 with n_1 a star quantifier node, n_2 a choice list node and m_1 a sequence list node. This is the reverse operation of *split*.

EXAMPLE 1. *For the DTDs shown as below, it is inappropriate to derive* <name>$_2$ *from* <name>$_1$ *by inserting a tag node* CEO *between* company *and* name *since* <name>$_2$ *indicates the company's CEO's name while* <name>$_1$ *indicates the company's name. We would rather first delete* name *and then insert a subtree rooted at* CEO *to derive DTD 2.*

```
<!ELEMENT company (name,address,webpage)>
<!ELEMENT name    (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT webpage (#PCDATA)>
                DTD 1
<!ELEMENT company (CEO,webpage)>
<!ELEMENT CEO  (name, address)>
<!ELEMENT name    (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT webpage (#PCDATA)>
                DTD 2
```

EXAMPLE 2. *<!ELEMENT company (street, city, state, zip)> can be transformed from <!ELEMENT company (address)> <!ELEMENT address (street, city, state, zip)>*

EXAMPLE 3. *<!ELEMENT company (license?)> <!ELEMENT license (PCDATA)> can be transformed from <!ELEMENT company (EMPTY)> <!ATTLIST company license CDATA #IMPLIED> by a relabel operation.*

EXAMPLE 4. *<!ELEMENT person (fax+)> can be unfolded to <!ELEMENT person (fax, fax) or <!ELEMENT person (fax, fax?)>.*

EXAMPLE 5. *<!ELEMENT name (first, last)> is split into <!ELEMENT name (first|last)*>.*

3.2 Constraints on the Transformation

While our atomic operations reflect intuitive transformations, some combinations of operations may result in non-intuitive transformations. For example, for the DTDs shown in Example 1 in Section 3.1, we can derive DTD 2 from DTD 1 by first inserting a sequence list node above *name* and *address*, and then relabeling the sequence list node to tag node CEO. This is equivalent to the forbidden operation of inserting a tag node CEO above *name* and *address*.

Common design patterns show that an element type declaration will not be deeply nested. A survey of real world DTDs [13] analyzes 65 DTDs available at [20]. 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 of sequences and choices of depth n. For example, $(a, (b|(c, d)))$ has depth 3. It turns out that the maximum depth of an element type's content type is almost around 2 and 3 with the average depth being even lower. This is because complex regular expressions are not advisable since it is difficult to understand. Also, usually the complex expression can be rewritten by some simpler ones. According to this design pattern, 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 subtrees rooted at their nearest matching ancestors in the DTD trees. This gives a hint that if the DTDs are designed in a common manner, the search space can be pruned to gain time efficiency. Therefore we discover only change scripts that do not violate the constraint that *a node can be only operated on by a non-relabel operation optionally followed or following a relabel operation*[4].

4. COST MODEL OF OPERATIONS

These operations can be combined into a variety of equivalent transformation scripts. In order to facilitate selection among alternative transformations, we propose a cost model that evaluates the cost of transformation operations in terms of their impact on the *data capacity* of the document schemas. *Relative information capacity* [8] measures the semantic connection between database schemas. That is, two schemas are considered equivalent if and only if there is a one-to-one mapping between all data instances in the source and the target schema. We assume that the DTDs in our study are flat [10], i.e., no schema information such as the names of element or attribute types in one DTD are stored as *PCDATA* or values of attributes in an XML document conforming to another DTD. Hence we only consider *PCDATA* and attribute values in XML documents as data. The *data capacity* of an XML document denotes the collection of all of its data.

4.1 Factors of the Cost Model

Data capacity gap. We say a transformation operation is *data capacity reducing* (*DC-Reduce*) if it must result in the loss of data, e.g., delete a subtree. Correspondingly, we have *data capacity increasing* (*DC-Increase*) operations, e.g., add a subtree, and *data capacity preserving* (*DC-Preserve*), e.g., relabel an element node to a sequence list node. However, for some operations, it is difficult to determine from the DTDs alone whether the transformation will result in the 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 these transformations *data capacity ambiguous* (*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. For any two *op*s falling into the same category, $DC(op)$ is the same and $0 \leq DC(op) \leq 1$.

Potential data capacity gap. Although some transformations are data capacity preserving, there may still be a potential data capacity gap between a 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 the children content particles' occurrence property from *countable* to *non-countable* and then allows the XML documents to accommodate more data in the future. We use $PDC(op)$ to denote the cost that the potential data capacity gap contributes to operation *op*'s overall cost. Then we define $PDC(op) = w_{required} * required\_changed(op) + w_{countable} * repeatable\_countable(op)$, where *required_changed(op)* and *countable_changed(op)* are two boolean functions that indicate whether the properties "required" or "countable" of the content particles that are operated on by *op* are changed or not. Weights $w_{required}$ and $w_{countable}$ indicate the importance of the change of the corresponding property for the potential data capacity. They satisfy $w_{required}+w_{countable} = 1$. Only operations that insert, remove or relabel a quantifier node may have a PDC cost that is not 0. For example, suppose $w_{required} = w_{countable} = 0.5$, then for *op* of *insert a quantifer node* < "+">, $required\_changed(op) = 0$, $countable\_changed(op) = 1$, therefore $PDC(op) = 0.5 * 0 + 0.5 * 1 = 0.5$.

Relative factors of operands. The number, size or property of operands involved in an operation may impact the data capacity or the potential data capacity gap. We use $Fac(op)$ to denote the factor. For example, for a content model $(fax+)$, a fold operation deriving it from (fax, fax) will be more expensive than that deriving it from (fax, fax, fax). This is because the former one causes a greater potential data gap. For another example, when relabeling occurs between two tag nodes, if their names are synonyms (e.g., "*zip*" and "*postal*"), $Fac(op)$ is 0. If no knowledge about the relationship of the two names' relationship is available, then $Fac(op)$ is proportional to the similarity of their name strings. For example, $Fac(op)$ of relabeling between "*address*"and "*addr*" will be less than that between "*address*" and "*capital*".

We then have, $Cost(op) = (DC(op) + PDC(op)) * Fac(op)$. The user of the Xtra system can customize the cost model by tuning the $DC(op)$, $PDC(op)$ and $Fac(op)$. We also provide a default setting. Intuitively information loss is not desirable in that old information cannot be reconstructed from the new information. Hence the more information loss the operation causes, the more expensive the operation is. Therefore we rank the cost of each data capacity gap category from

[4]Relabel is the only operation that does not change the tree's topology.

lower to higher in the order of *DC-Preserve*, *DC-Increase*, *DC-Ambiguous* and *DC-Reduce*. However our algorithm of discovery the transformation script does not depend on this particular relationship.

4.2 Examples

We illustrate how to use the cost model to choose a matching plan from multiple candidates using the running example. Assume we have the following settings: *DC-Preserve*, 0.6; *DC-Increase*, 0.8; *DC-Ambiguous*, 0.9; *DC-Reduce*, 1.0. And $Fac(op)$ for an operation *op* of relabeling between an element node and a sequence list node is 3 while that for relabeling between two synonym element nodes is 0. Also, for an operation *op* that adds a subtree, we define that $Fac(op)$ is proportional to the number i of subtree's leaf nodes (i.e., $Fac(op) = k * i$) and we assume $k = 1$. Suppose now we have two options to derive the subtree rooted at $<,>_2$. The first option is to match $<address>_1$ to $<,>_2$, i.e., relabel $<address>_1$ from [*address*] to [,] and relabel $<zip>_1$ from [*zip*] to [*postal*]. The second option is to add a new subtree which is the one finally rooted at $(<,>_2$. For the first option, the cost of the first relabeling is $(DC(op)+PDC(op))*Fac(op) = (0.6+0)*3 = 1.8$ while the cost of the second relabeling is $(DC(op)+PDC(op))*Fac(op) = (0.6+0)*0 = 0$. The total cost is then $1.8 + 0 = 1.8$. For the second option, the cost is $(DC(op)+PDC(op))*Fac(op) = (1.0+0)*4 = 4$. In this case, the first option is preferable due to its lower cost. However, suppose that $< address >_1$ only has one single child node $<postal>_1$. The first option now has three more operations than the original sequence of operations, i.e., add $<street>_1$, add $<city>_2$ and add $<state>_1$. These three additional operations cost $(0.8+0)*1 = 0.8$ each. The total cost of the first option is therefore $1.8 + 0.8 * 3 = 4.2$. The cost of the second option is (i.e., 4) does not change since the sequence of operations does not change. This time the first option is not preferable since it is more expensive than the second option.

5. GENERATION OF DTD TREE MATCHES

5.1 Simplified Element Trees

We constrain our investigation to the domain of E-business documents that are exchanged between services that share a common ontology. We thus can use name similarity as the first heuristic indicator of a possible semantic relationship between two tag nodes. For example, in Figures 3 and 4, the matching document root type all have a child node named *personnel*, so without looking at their descendants, we can match these two nodes. We can derive the matching between two *personnel* nodes' descendants by comparing two *personnel*'s type declaration trees separately. However suppose in DTD 2, *people* were used instead of *personnel* and no synonym knowledge was given. We would then need to look further at *personnel* and *people*'s descendants to decide whether to match them.

We therefore introduce the concept of a *simplified element tree*. Such a tree is designed to capture the relationship indicated by names between specific elements of the two documents. When two DTDs are provided, we say a tag node is *non-rename-able* if there exists any tag in the other DTD whose name is the same or a synonym. In another word, the $Fac(op)$ cost of an operation *op* of renaming a *non-rename-able* node to another node with a non-synonym name is infinitely expensive. A simplified element tree of element type E, denoted as ST, is a subtree of E's type declaration tree T that roots at T's root with each branch ending at the first reachable non-rename-able node. In Figures 3 and 4, the four subtrees within the dashed lines are simplified element trees of *company*, *personnel*, *person* and *name* in the two DTDs respectively.

5.2 The Matching Algorithm

DMatch(Dtd Match) is an XML-structure-specific tree matching algorithm. The general unordered tree matching problem is a notoriously high complexity NP problem [22]. As we have mentioned in Section 1.2, the typical assumption about relabeling in general tree matching does not hold in our scenario. Thus those techniques do not apply to our scenario. Our *dMatch* tree matching algorithm here incorporates the domain characteristics of specific DTD tree transformation operations, the imposed constraints and the cost model.

Given a source simplified element tree T_1, and a target simplified element tree T_2, we call nodes in T_1 *source nodes* and nodes in T_2 *target nodes*. If n_1 and n_2 are a source and a target node respectively, the $DMatch(n_1, n_2)$ algorithm discovers a sequence of operations (i.e., the transformation script) that transforms the subtree rooted at n_1 to the subtree rooted at n_2. We call the sequence of operations a *transformation script*. The total cost of the script is then the cost of matching (deriving) n_1 and n_2. For the source nodes that are deleted or removed, since they are not mapped to any existing target node, in order to simplify the description, we specify two special nodes, namely, Φ_1 and Φ_2. A node which is removed is said to be mapped to Φ_1. And a node which is deleted is said to be mapped to Φ_2.

DMatch is composed of two phases. The first phase is preprocessing. And We have two special nodes, namely, Φ_1 and Φ_2. Φ_1 is mapped to nodes which are operated on by *remove* operation. And Φ_2 is mapped to nodes which are operated on by *delete* operation. The operations *add*, *insert*, *delete*, *remove* and *relabel* set up a one-to-one mapping relationship. However, the operations *unfold*, *fold*, *split* and *merge* set up a one-to-many relationship. For example, *unfold* maps one subtree to multiple subtrees, *split* maps two nodes (a star quantifier and a choice list node) to a sequence list node. In order to make the matching discovery process for each node be uniform, we preprocess each of the simplified element trees.

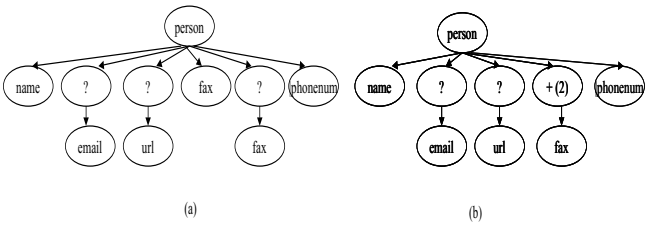

Figure 5: Preprocessing: Fold

In the preprocessing phase, we will convert all the representations of a sequence of non-repeatable content particles into the format of a repeatable content particle, i.e., perform

fold operations. For example, Figure 5 (a) will be converted to Figure 5 (b). In Figure 5 (b), we mark the plus quantifier node with a number (i.e., two) indicating the maximum occurrence of content particle *fax*. By marking these nodes resulting from the preprocessing, we are able to keep track of where they are derived from. Thus we will not lose the information needed for computing the overall cost. Second, we will impose a certain order on those representations that allow arbitrary combination of content particles, i.e., perform *merge* operations.

In the second phase, we then find one-to-one node mappings. To derive the transformation from the subtree rooted at n_1 and the subtree rooted at n_2, for each child m_1 of n_1, we attempt to find a matching partner m_2 (a matching partner can be either be also a special node Φ_1 or Φ_2). 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*. Since we have the constraints that a node cannot be directly operated on more than once, m_1's matching partner m_2 can only be on the same level as m_1 (no operation or *relabel* operated on m_1) or one level deeper than m_1 (*insert* operated on m_1) or a special node (*delete* or *remove* operated on m_1). By recursively applying *DMatch* to m_1 and each node s_i $(0 < i < |S|)$ in S, we find a node s_k with the least matching cost c. We have a control strategy to decide whether to match m_1 with s_k. In pass 1, we apply a *delay-match* scheme which delays matching s_k with m_1 if the cost is not low enough, i.e., c is not less than the cost of deleting m_1. Thus s_k can be preserved to be matched with another node associated with a satisfactorily low cost.

Source	Matching Candidate Set
element	element node on the same level.
attribute	attribute node on the same level.
choice	choice node on the same level.
sequence	sequence node on same level or one level deeper; Φ_1.
quantifier	quantifier node on same level or one level deeper; Φ_1.

Figure 6: Matching Candidate Set in Pass 1

Source	Matching Candidate Set
element	element node on same level or one level deeper; sequence node on same level; attribute node on same level.
attribute	element node on the same level.
choice	choice node on same level or one level deeper.
sequence	sequence node on same level or one level deeper; Φ_1; quantifier node on same level.
quantifier	quantifier node on same level or one level deeper; Φ_1; sequence node on same level.

Figure 7: Matching Candidate Set in Pass 2

After visiting all children of n_1, we begin pass 2 and traverse all unmatched children of n_1 again, comparing them with possible candidates. Again, we apply *DMatch* to m_1 and each node s_i in S and find a node s_k with the least matching cost c. Now a *must-match* scheme is applied in contrast to the *delay-match* scheme in pass 1. m_1 would be matched with s_k if c is less than the cost of deleting m_1 and adding s_k. Figures 6 and 7 show the nodes that will be selected into the matching candidate set S in pass 1 and pass 2 respectively.

The pseudo code of the *DMatch* algorithm is shown in Figure 8. The full details of the algorithm, including discussion of optimality, time complexity, etc. can be found in [15].

```
DMatch(n1, n2)
{//pass 1
   for n1's each child m1
    {
     Set S = m1's matching candidate set in pass 1;
     for each node si ∈ S
      apply DMatch(m1, si);
     select a node sk in S associated with a least cost; c
     if c < the cost of deleting m1
      match m1 and sk. //delay-match scheme
    }

   //pass 2
    for n1's each unmatched child m'1
     {
      Set S' = m'1's matching candidate set in pass 2;
      for each node s'i ∈ S'
       apply DMatch(m'1, s'i);
      select a node s'k in S' associated with a least cost c';
      if c' < the cost of deleting m'1 + the cost of adding s'k
       match m'1 and s'k. //must-match scheme
     }
    }
```

Figure 8: Pseudo Code of *DMatch* Algorithm

Given matching root element types, R_1 and R_2 of two DTDs, we apply *DMatch* to the roots of R_1 and R_2's simplified tree. The root match is propagated down the tree and matches between the name-match nodes of element types E_1 and E_2 are identified. We then recursively apply the *DMatch* algorithm to E_1 and E_2's simplified trees until no new name-match node matches are generated.

5.3 Example Illustrating Matching Process

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.

6. DISCUSSION

Once the relationship has been set up, an XSLT generator will generate an XSLT script for transforming the source XML documents into the target format. We have implemented a working system *XTra* and run experiments on it. The data sets we are using for experimentation include both real world data collected from [20] and synthetic data. It turns out that our algorithm can satisfactorily discover acceptable transformations. Due to the space limitation, we do not further discuss them here. The details can be found in [15].

7. CONCLUSION AND FUTURE WORK

This work proposes an approach for automating the transformation of XML documents. Specifically, we focus on two fundamental problems. First, we address the problem of how to automate the identification of semantic relationships between XML-based documents. To this end, we propose a set of DTD transformation operations that capture common discrepancies between alternative DTD design behaviors for modeling a given entity. We also define a cost model for quantifying the quality of XML schema transformations. Second, we have developed an algorithm that performs the actual transformation of an XML-based document from a given schema to a different, yet related, schema. Our 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 (e.g., quantifiers and type-constructors). This allows us to avoid the high level of user interaction as well as the complexity required by other approaches. We have implemented a prototype system (Xtra), and run experiments on both real and synthetic data to verify the validity of our approach [15].

XML-Schema [18] is emerging as a potential standard for describing the structure of XML documents. In the future we could investigate how to adapt our approach to exploit the richer treatment of types offered by XML Schema as additional hints of similarity.

Acknowledgements. This work was supported in part by several grants from NSF, namely, the NSF NYI grant #IRI 94–57609, the NSF CISE Instrumentation grant #IRIS 97–29878, and the NSF grant #IIS 9988776. We thank HP for partial support of Hong Su in terms of a HP Labs Grassroots Basic Research Grant from 2000 to 2001 and IBM for support of Hong Su in terms of a Corporate Fellowship from 2001 to 2002. We are grateful to Umesh Dayal for helpful discussion. We would also like to thank Carolina Ruiz and the entire DSRG group at WPI for many inspiring discussions.

8. REFERENCES

[1] Arindam Banerji and Claudio Bartolini and Dorothea Beringer et, al. WSCL 1.0. http://www.e-speak.hp.com/media/wscl_5_16_01.pdf, 2001.

[2] S. Bergamaschi, S. Castano, S. D. C. D. Vimeracati, and M. Vincini. An intelligent approach to information integration. In *Int. Conference on Formal Ontology in Information Systems*, pages 253–267, 1998.

[3] 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)*, pages 53–62, 1999.

[4] S. S. Chawathe and H. Garcia-Molina. Meaningful Change Detection in Structured Data. In *SIGMOD*, pages 26–37, 1997.

[5] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *SIGMOD*, pages 493–504, 1996.

[6] A. Doan, P. Domingos, and A. Levy. Reconciling Schemas of Disparate Data Sources: A Maching-Learning Approach. In *SIGMOD*, pages 509–520, 2001.

[7] H. Su and D. Kramer et, al. XEM: Managing the Evolution of XML Documents. In *International Workshop on Research Issues in Data Engineering (RIDE-DM'2001)*, pages 103 – 110, 2000.

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

[9] H. Kuno. Surveying the E-Services Technical Landscape. In *International Workshop on Advance Issues of E-Commerce and Web-Based Information Systems (WECWIS)*, pages 94 – 101, 2000.

[10] L. Lakshmanan, F. Sadri, and I. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-database Systems. In *VLDB*, pages 239–250, 1996.

[11] R. Miller, L. Haas, and M. Hernandez. Schema mapping as query discovery. In *VLDB*, pages 77–88, 2000.

[12] T. Milo and S. Zohar. Using schema matching to simplify heterogenous data translation. In *VLDB*, pages 122–133, 1998.

[13] A. Sahuguet. Everything you ever wanted to know about DTDs, but were afraid to ask. In *Third International Workshop on the Web and Databases (WebDB'00)*, pages 69–74, 2000.

[14] 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.

[15] H. Su, H. Kuno, and E. A. Rundensteiner. Automating the Translation of XML Documents. Technical Report WPI-CS-TR-01-13, Computer Science Department, Worcester Polytechnic Institute, 2001.

[16] United Nations (UN/CEFACT) and OASIS. ebXML. http://www.ebxml.org, 2001.

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

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

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

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

[21] L. Yan, R. J. Miller, L. M. Hass, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In *SIGMOD*, pages 485–496, 2001.

[22] K. Zhang, J. Wang, and D. Shasha. On the editing distance between undirected acyclic graphs. *International Journal of Foundations of Computer Science*, pages 395–407, 1995.