On The Document Algebra

Yangjun Chen*

Dept. Business Computing, Winnipeg University
515 Portage Ave. Winnipeg, Manitoba, Canada, R3B 2E9

ABSTRACT

In this paper, a document algebra is proposed to support both document transformation and pattern matching. Based on the tree domain theory, the operational semantics for the document transformation are defined. Then, by equating a subtree structure from a DTD to an attribute from a relation schema, a set of operations for treating document sets is developed, which is equipped with the pattern matching to cope with the queries issued to a document database.

KEY WORDS

Document, Algebra, Tree domain theory

1. Introduction

Until the mid-1980s, the relational algebra received a lot of attention due to its simplicity, both for modeling and manipulating data. Recently, however, we became aware of its insufficiency when trying to model data applications beyond the traditional business-oriented applications, such as office automation, multimedia databases and text-oriented applications.

In this paper, we present a document algebra as a possible successor of the relational model, aiming at the document treatment. The proposed algebra is based on the *tree domain theory* which has been extensively used to study the tree logic [14], tree automaton [16] and tree periodicity [9].

We distinguish between two groups of operations: those for document transformation and those for manipulating sets of documents (by which the pattern matching is needed.) The first group is to operate on single documents. A lot of examples are given to show how the operations can be utilized to elegantly translate a document into another one and how an (approximate) DTD can be derived from a set of existing documents. The second group provides a series of operations analogous to the relational algebra. By abstractly equating a subtree structure from a DTD (Document Type Descriptor) to an attribute from a relation schema, we can see that a document roughly corresponds to an extended "tuple" (in which each value is associated with a *node address*; see below.) This observation enables us to build a bridge between the document algebra and the relational algebra and accordingly, w.r.t. the treatment of the sets of documents, the same operation set can be established for the document algebra as for the relational model. These two groups form the core of a query language wherein users can succinctly and naturally formulate complex problems typically encountered in document databases.

Recently, much research has been directed toward the data models which recognize as the most fundamental characteristic of data its hierarchical structure [3, 7, 10, 15]. In [10], an algebra for transforming tree structures was developed, in which several primitive operators are provided to change a tree. But no attention was paid to operate among single trees as well as tree sets. In [15], a so-called forest algebra is introduced based on the tree automaton theory [16] to support document transformation and pattern matching. But it is too intricate to be governed. It is more interesting from a theoretical perspective than from a practical viewpoint. In addition, how to define a deterministic forest automaton for a given DTD has never been clearly discussed. [7] is another interesting suggestion; but the algebra provided is not purely declarative, mixed with a predicate calculus. All the above methods are grammar-based.

An entirely different approach tries to extend the relational model and the object-oriented model to capture hierarchies of structured documents with complex values. Probably, the most notable example is by Christophides *et al*. [3]. They use O_2 as a basis and further introduce ordered tuples and marked unions to represent hierarchies of SGML/XML documents. However, extension in such a way can not capture the inherent hierarchical characteristic of documents, leading to cumbersome operations for manipulating data.

The rest part of the paper is organized as follows. In Section 2, we give the definition of the document tree according to the tree domain theory. Then, in Section 3, we define a set of operations for transforming document trees. In Section 4, a short conclusion is set forth.

2. DTD and Document trees

In this section, we introduce the concept of labeled trees which can be used to represent documents exactly.

Intuitively, a labeled tree can be considered as consisting of two parts: a tree domain and a set of labels associated with each element in the tree domain. The following is its formal definition.

Definition 2.1 (*tree*) Let $\Sigma = \{1, ..., k\}$ and let A be a finite alphabet. A *k-ary* (labeled) tree over A is a mapping T: $\Sigma^* \cup \{\varepsilon\} \rightarrow A$, where $\varepsilon \notin \Sigma$ and Σ^* denotes the set of all finite-length sequences of letters from Σ. The domain of T: $dom(T)$ is a finite and *prefix-closed* subset of Σ^* plus $\{\varepsilon\}$. We say a subset U of $\Sigma^* \cup \{\varepsilon\}$ to be *prefix-closed* if

(i) $\omega \in U \wedge \omega = uv \rightarrow u \in U$ $(\omega, u, v \in \Sigma^*)$
(ii) $\omega i \in U \wedge j \leq i \rightarrow \omega j \in U$ $(\omega \in \Sigma^*, i, j \in U)$.

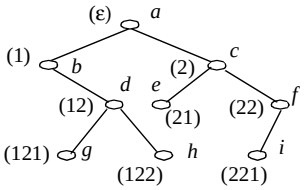
The elements in $dom(T)$ are called *nodes* (or node addresses)

* The author is supported by NSERC 239074-01 (242523) (Natural Sciences and Engineering Council of Canada).

and ε is always used to represent the root address of a tree. We say that T is an unlabeled tree if the alphabet contains only one element (i.e., all nodes have the same label.)

Example 2.1 Let $\Sigma = \{1, 2\}$ and $A = \{a, b, c, d, e, f, g, h, i\}$. Consider the mapping T: $\Sigma^* \cup \{\varepsilon\} \rightarrow A$ given by the equations shown in Fig.1(a), which corresponds to a labeled binary tree as shown in Fig. 1(b).

$T(\varepsilon) = a$, $T(1) = b$, $T(2) = c$,
$T(12) = d$, $T(21) = e$, $T(22) = f$,
$T(121) = g$, $T(122) = h$, $T(221) = i$,



(a) (b)

Fig. 1. An exemplary tree

This mapping is defined over the prefix-closed set: {1, 2, 12, 21, 22, 121, 122, 221} plus {ε}. For a label a in a tree, its corresponding nodes (or node addresses) are denoted by $T^{-1}(a)$. We notice that T^{-1} is a multi-valued function since many nodes may have the same label.

Definition 2.2 (*subtree*) Given a tree T and a node address α in $dom(T)$, the subtree rooted at α, denoted $T_{sub}(\alpha)$, is a tree such that $dom(T_{sub}(\alpha)) = \{v_1v_2 \ldots v_k \mid \alpha v_1v_2 \ldots v_k \in dom(T)$ and $T_{sub}(\alpha)(v_1v_2 \ldots v_k) = T(\alpha v_1v_2 \ldots v_k)\}$.

The empty tree, i.e., the tree mapping with domain ∅, is denoted by Λ.

A terminal of a tree T is an element whose node address $\omega \in dom(T)$ such that no extension of ω is also in $dom(T)$. The set of all terminals of T is called the *frontier* of T, denoted by $fr(T)$.

Now we consider an SGML/XML document. It can always be represented as a labeled tree. As an example, consider a possible DTD for *letter* documents shown in Fig.2. Its pictorial representation is as shown in Fig. 3.

```
1. <!DOCTYPE    letter [
2. <!ELEMENT    letter      - - (date, greeting, body, closing, sig)>
3. <!ATTLIST    letter
       filecode    NUMBER     #REQUIRED
       secret      (yes | no)   "no">
4. <!ELEMENT    body        - - (para)+>
5. <!ELEMENT    (date, greeting, closing, sig) - - (#PCDATA)>
6. <!ELEMENT    para        - - (text | emph)*>
7. <!ELEMENT    emph        - - (#PCDATA)>
8. <!ATTLIST    emph
       italic      (yes | no)    "yes">
9. <!ENTITY     salute       "Dear">
   ]>
```

Fig. 2. A sample DTD

The tree domain of this tree structure is a prefix-closed set: {1, 2, 3, 4, 5, 11, 21, 31, 311, 3111, 312, 3121, 41, 51} plus {ε}, labeled with the tag names: A = {letter, date, greeting, closing, sig, para, text, emph}, and "#PCDATA" which is a data type, more or less comparable to string. Such a tree is called a *DTD tree* or a *schema tree*. Note that in the cases of multiple appearance (as illustrated in Fig. 4(a)) and recursion (as illustrated in Fig. 4(b)), the structure of a DTD can also be represented as a tree as illustrated in Fig. 4(c) and 4(d), respectively.

In terms of the above discussion, a DTD can always be represented as a set of pairs with the following form: (α, t), (α, t*), or (α, t^+), where α is a node address and t is a tag name, a symbol indicating the data type or a complex value (see below); "*" and "+" are two connection indicators as defined for regular expressions.

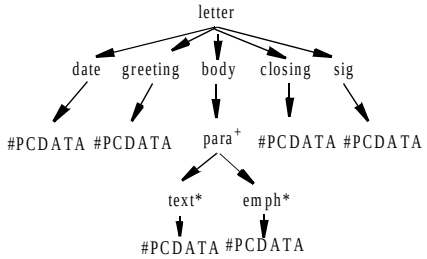


Fig. 3. Tree representation of the DTD shown in Fig. 2

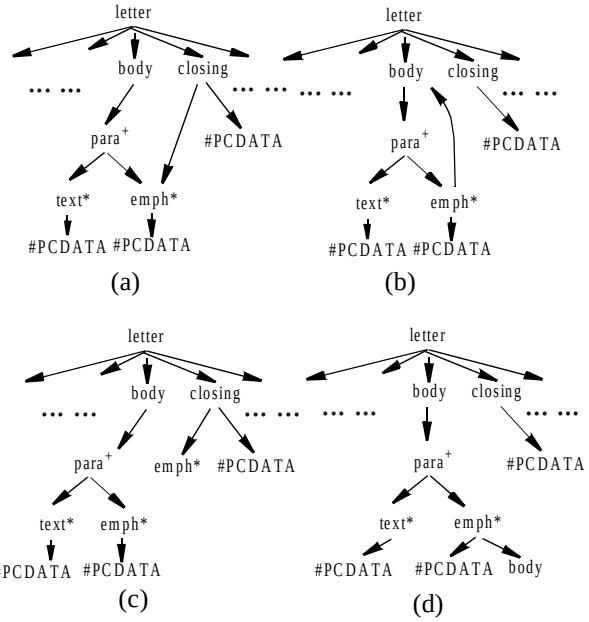


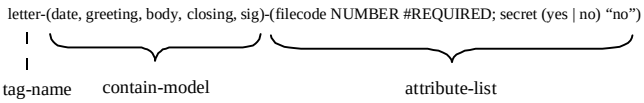
Fig. 4. Transformation of DTD trees

A complex value is defined as consisting of three parts as follows:

<complex value> := <tag-name>[-<contain-model>][-<attribute-list>].

It is used to capture part of a DTD structure, which can not be represented by the tree structure alone.

For instance, the complex value for the node labeled with "letter" is of the form:



Obviously, such a value is too complicated to be represented by the tree structure. Therefore, by the definition of the operations over trees, attention should be paid to the inner structutre of a label. That is, by any operation, we should specify whether a simple value (tag-name or data type) or a complex value is considered and in the latter case, it should be further specified which parts of a complex value is taken into account. To this end, we represent a label using a triple $<l_t, l_c(i), l_a(j)>$, where l_t is for the tag-name, $l_c(i)$ refers to the ith element in

the contain-model and $l_a(j)$ refers to the jth attribute of the attribute-list. If the whole contain-model or the whole attribute-list is considered, we simply use l_c or l_a, respectively.

In the following, however, we use the label for both the simple and complex values for simplicity. If the distinction between them is necessary, we utilize l_t, l_c and l_a to reference the different parts of a label as discussed above.

In addition, the connector "|" is ignored. That is, we regard simply each sub-element appearing in the content model of an element as a child, no matter whether they are connected with "|" or not. In fact, this connector is only for the syntactical description of a document and not related to the manipulation of tree structures themselves.

Accordingly, a document conforming to a DTD can also be represented as a set of pairs of the form: (α, t), where α has the same meaning as above and t is a tag name (possibly with the attribute value assignment) if α is the address of an interior node, or a value (a string, a picture, or something else, which can be treated by software) if α is a terminal node address. We call a tree for a document the *document tree.*

In the following, we will define all basic operations on trees (DTD trees or document trees) in Section 3.

3. Basic operations for DTD and document transformation

In this section, eight operations for tree transformation are defined based on the tree domain theory. They are *tree-union*, *tree-intersection*, *tree-symmetric-difference*, *tree-concatenation*, *tree-substitution*, *node-insertion*, *node-deletion*, and *label-renaming.*

To specify these basic operations over trees, we need some extra concepts.

Definition 3.1 (*interval closed*) Let $\Sigma = \{1, ..., k\}$. A subset S of Σ^* is interval closed if for any words v and $vab \in S$, we have $va \in S$, where $a, b \in \Sigma$.

Definition 3.2 (*bush*) Let $\Sigma = \{1, ..., k\}$ and let A be a finite alphabet. A *k-ary* (labeled) bush over A is a partial mapping B: $\Sigma^* \rightarrow A$ whose domain $dom(B)$ is a finite and interval closed subset of Σ^*.

Example 3.1 Let $\Sigma = \{1, 2\}$ and $A = \{d, e, f, g, h, i\}$. The mapping shown in Fig. 5(a) is a bush. Its pictorial representation is shown in Fig. 5(b).

Definition 3.3 (*compatible*) Let T_1 and T_2 be two labeled trees. We say that T_1 and T_2 are compatible if and only if they coincide as functions on the intersection of their domains (i.e., the nodes with the same addresses in T_1 and T_2 will have the same labels.) In formula, if $T(D)$ denotes the restriction of T to $D \subseteq dom(T)$, we write $T_1(dom(T_1) \cap dom(T_2)) = T_2(dom(T_1) \cap dom(T_2))$.

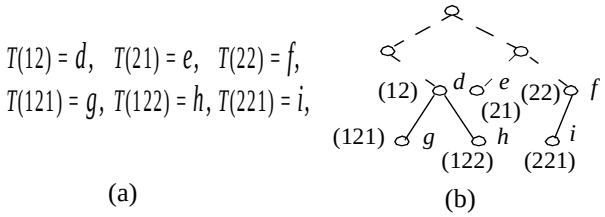


Fig. 5. Illustration for bush

Now we can define several operations among trees (DTD trees or document trees).

- tree-union ($\oplus$)

Let T_1 and T_2 be two compatible trees. The union of them ($T_1 \oplus T_2$) is defined by

(i) $dom(T_1 \oplus T_2) = dom(T_1) \cup dom(T_2)$;
(ii) $\forall x \in dom(T_1 \oplus T_2)$,

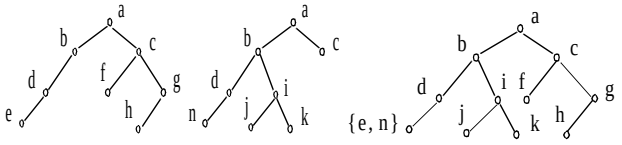
$$(T_1 \oplus T_2)(x) = \begin{cases} T_1(x), & \text{if } x \in dom(T_1); \\ T_2(x), & \text{otherwise} \end{cases}$$

This definition works well for the DTD trees. But for the document trees, a little bit modification is needed due to the fact that incompatible terminal nodes should be allowed to take the union operation for the practical purpose. In this case, the different terminal nodes (in different document trees) with the compatible parents will be put together to construct a bigger piece of texts. To this end, we change the above definition as follows:

$(T_1 \oplus T_2)(x) =$

$$\begin{cases} T_1(x), & \text{if } x \in dom(T_1)/\text{fr}(T_1) \text{ or } x \in \text{fr}(T_1) \text{ but } x\text{'s parent} \\ & \text{is not compatible with any terminal's parent in } T_2; \\ T_2(x), & \text{if } x \in dom(T_2)/\text{fr}(T_2) \text{ or } x \in \text{fr}(T_2) \text{ but } x\text{'s parent} \\ & \text{is not compatible with any terminal's parent in } T_1; \\ \{T_1(x)\} \cup \{T_2(x)\}, & \text{otherwise} \end{cases}$$

By this definition, we notice that the label of a terminal of a document tree is a value. Therefore, if for a terminal $x \in \text{fr}(T_1) \cap \text{fr}(T_2)$ its respective parents in T_1 and T_2 are compatible, the label of x in T_1 and the label of x in T_2 are joined together. That is, its corresponding new label is $\{T_1(x)\} \cup \{T_2(x)\}$. For instance, the union of two document trees D_1 and D_2 shown in Fig. 6(a) is a new tree shown in Fig. 6(b).



(a) (b)

Fig. 6. Illustration for tree-union

- tree-intersection ($\bullet$)

Let T_1 and T_2 be again two compatible trees. The intersection of them ($T_1 \bullet T_2$) is a new tree such that

(i) $dom(T_1 \bullet T_2) = dom(T_1) \cap dom(T_2)$
(ii) $T_1 \bullet T_2 = T_1(dom(T_1) \cap dom(T_2))$
$= T_2(dom(T_1) \cap dom(T_2))$.

For example, the intersection of trees D_1 and D_2 shown in Fig. 6(a) is the tree shown in Fig. 7.

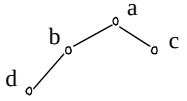


Fig. 7. Illustration for tree-intersection

- tree-symmetric-difference (~)

We can imagine that if we simply apply the normal set difference operation to the domains of two trees, we do not get a tree but a bush.

Let T_1 and T_2 be two compatible trees. We define the bush $T_1 \sim T_2$ as follows.

(i) $dom(T_1 \sim T_2) = dom(T_1 \oplus T_2) / dom(T_1 \bullet T_2)$;
(ii) $\forall x \in dom(T_1 \sim T_2)$,

$$(T_1 \sim T_2)(x) = \begin{cases} T_1(x), & \text{if } x \in dom(T_1); \\ T_2(x), & \text{otherwise} \end{cases}$$

For instance, the symmetric difference of trees D_1 and D_2 shown in Fig. 6(a) is a set of document pieces shown in Fig. 8.

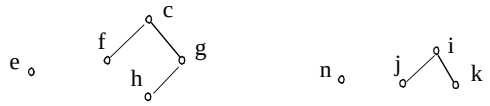


Fig. 8. Illustration for tree-symmetric-difference

We now define the concatenation between two trees. Intuitively, to concatenate two trees T_1 and T_2, we "attach" the root of T_2 to one of the elements of the border of T_1. The border of a tree can be defined as follows.

Definition 3.4 (*border*) Given a tree T, the border of T is the set $B(T) = \{wi \mid w \in dom(T), i \in \Sigma;$ but $wi \notin dom(T)\}$.

In general, as a result, we get more than one trees since the border of a tree normally contains more than one elements. Then, the concatenation between two trees will be a set of trees containing as many trees as the elements in $B(T)$. We first define formally the concatenation of two trees at a given element of the border. Then, the general concatenation operation can be defined.

Let T_1 and T_2 be two trees; and let $B(T_1)$ denote the border of T_1. The concatenation of T_1 and T_2 at $\alpha \in B(T_1)$ is a tree $T_1(\alpha)T_2$ defined as

(i) $dom(T_1(\alpha)T_2) = dom(T_1) \cup dom(T_2)$;
(ii) $\forall x \in dom(T_1(\alpha)T_2)$,

$(T_1(\alpha)T_2)(x) =$

$$\begin{cases} T_1(x), & \text{if } x \in dom(T_1); \\ T_2(y), \text{ where } y \in dom(T_2), \alpha y = x, & \text{otherwise} \end{cases}$$

- tree-concatenation ($\cdot$)

Let T_1 and T_2 be two trees; and let $B(T_1)$ denote the border of T_1. The concatenation of T_1 and T_2 is the set of trees:

$$T_1 \cdot T_2 = \{T_1(\alpha)T_2 \mid \alpha \in B(T_1)\}.$$

As an example, consider the trees T_1 and T_2 shown in Fig. 9.

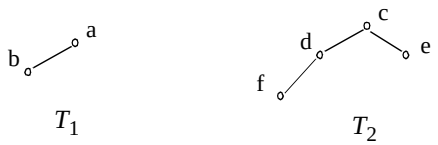


Fig. 9. Two simple trees

The concatenation $T_1 \cdot T_2$ is a set of trees as shown in Fig. 10.

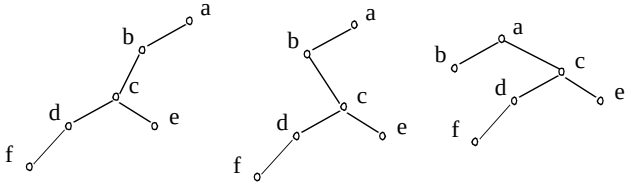


Fig. 10. Concatenation of two trees shown in Fig. 9

- subtree-substitution (/)

Given two trees T_1 and T_2 and a node address α in $dom(T_1)$, the substitution of T_2 for the subtree $T_{1sub}(\alpha)$, denoted $T_1[\alpha/T_2]$, is the tree defined by

(i) $dom(T_1[\alpha/T_2]) = (dom(T_1)/dom(T_{1sub}(\alpha))) \cup \{\alpha w \mid w \in dom(T_2)\}$;

(ii) $\forall x \in dom(T_1[\alpha/T_2])$,

$(T_1[\alpha/T_2])(x) =$

$$\begin{cases} T_1(x), & \text{if } x \in dom(T_1)/dom(T_{1sub}(\alpha)); \\ T_2(x), & \text{otherwise} \end{cases}$$

This operation can be illustrated as shown in Fig. 11.

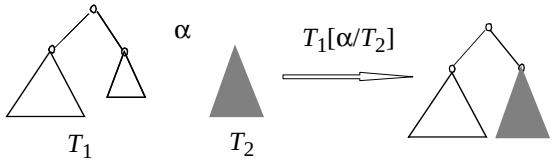


Fig. 11. Illustration for tree-substitution

- node-insertion

Let T be a tree. Let α be a node-address in the tree and β be one of α's children. The insertion of a node labeled a as a child of α but the parent of β, denoted $T[\alpha, \beta, a]$, can be defined as follows.

Let $A = dom(T)/dom(T_{sub}(\beta))$ and $B = \{\beta 1w \mid w \in dom(T_{sub}(\beta))\}$. Then we have

(i) $dom(T[\alpha, \beta, a]) = A \cup B \cup \{T^{-1}(\beta)\}$;
(ii) $\forall x \in dom(T[\alpha, \beta, a])$,

$(T[\alpha, \beta, a])(x) =$

$$\begin{cases} T(x), & \text{if } x \in A; \\ T(y) \text{ where } y \in dom(T_{sub}(\beta)), x = \beta 1y, & \text{if } x \in B; \\ a, & \text{otherwise} \end{cases}$$

See Fig. 12 for illustration.

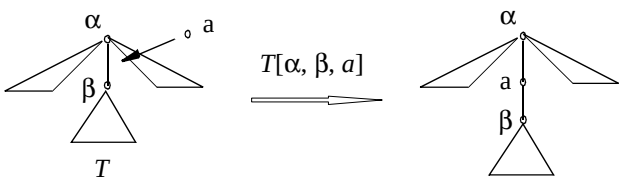


Fig. 12. Illustration for node-insertion

- node-deletion

Let T be a tree and α be a node address in the tree. The deletion of α from T, denoted $T[\sim\alpha]$ can be defined as follows. Let α

= *wi*, where $w \in dom(T)$ and *i* is an integer. Let $\beta_j = wij$ (j = 1, 2, ..., k) be the node addresses of α's children. Let $od(w)$ denote the outdegree of *w*. We first define three sets:

$A = dom(T) / \bigcup_{n=i}^{od(w)} dom(T_{sub}(wn))$;

$B = \{wju \mid u \in dom(T_{sub}(\beta_j), j = i+1, , ..., i+k\}$;

$C = \{wju \mid u \in dom(T_{sub}(wl), l = i+1, ..., od(w), j = i+k, ..., i+od(w)\}$.

Then we have

(i) $dom(T[\sim\alpha]) = A \cup B \cup C$;

(ii) $\forall x \in dom(T[\sim\alpha])$,

$(T[\sim\alpha])(x) =$

$$\begin{cases} T(x), & \text{if } x \in A; \\ T(y) \text{ where } y \in dom(T_{sub}(\beta_j)), x = wiy \text{ for some } j, & \text{if } x \in B; \\ T(z) \text{ where } z \in dom(T_{sub}(wl)), x = w(l-i)z \text{ for some } l, & \text{if } x \in C; \end{cases}$$

Fig. 13 helps for illustration.

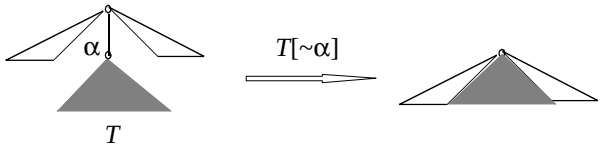


Fig. 13. Illustration for node-deletion

- label-renaming

Two label-renaming operations are provided. The first is used to replace the label of some node with a new one. The second substitutes a new label for all appearances of some label in the tree.

Let *T* be a tree. Let α be a node address in the tree. The first operation, denoted $T[T(\alpha) \leftarrow a]$ (where *a* is a label), can be defined as follows:

(i) $dom(T[T(\alpha) \leftarrow a]) = dom(T)$;
(ii) $\forall x \in dom(T[T(\alpha) \leftarrow a])$,

$(T[T(\alpha) \leftarrow a])(x) =$

$$\begin{cases} T(x), & \text{if } x \in dom(T)/\{\alpha\}; \\ a, & \text{otherwise} \end{cases}$$

The second operation is denoted $T[b \leftarrow a]$, by which all the appearances of *b* will be replaced with *a*. The following is its definition.

(i) $dom(T[b \leftarrow a]) = dom(T)$;
(ii) $\forall x \in dom(T[b \leftarrow a])$,

$(T[b \leftarrow a])(x) =$

$$\begin{cases} T(x), & \text{if } x \in dom(T)/T^{-1}(b); \\ a, & \text{otherwise} \end{cases}$$

The following example demonstrates how the above operations can be utilized to perform a document (DTD) transformation.

Example 3.2 Consider the DTD given in Section 2 again. Suppose we want to extract only the information on the letter texts and signatures. The corresponding DTD would then be transformed into the following form:

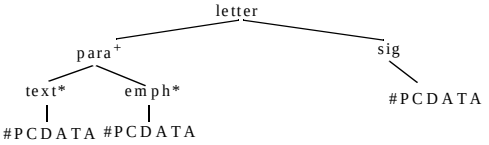


Fig. 14. A transformation of the DTD shown in Fig. 2

The transformation on the corresponding documents can be accomplished by consecutively performing the following operations:

for each $a \in T^{-1}$(date) **do**
  {$T[a/\Lambda]$}; (*replace, for example, the subtree rooted at (1, date) with Λ*)
for each $a \in T^{-1}$(greeting) **do**
  {$T[a/\Lambda]$}; (*replace, for example, the subtree rooted at (2, greeting) with Λ*)
for each $a \in T^{-1}$(closing) **do**
  {$T[a)/\Lambda]$}; (*replace, for example, the subtree rooted at (4, closing) with Λ*)
for each $a \in T^{-1}$(body) **do**
  {$T[\sim a]$}; (*delete, for example, the node (3, body)*)

Alternatively, we can write these operations in the following form for short.

$T[T^{-1}(\text{date})/\Lambda]; T[T^{-1}(\text{greeting})/\Lambda]; T[T^{-1}(\text{closing})/\Lambda]; T[\sim T^{-1}(\text{body})]$.

Hereafter, the concise representation like this will be employed without further declaration if no confusion arises.

In addition, we notice that each element $e \in dom(T)$ also represents a path from the root to some node with *e* as the address. Let $e = i_1.i_2. ... i_j$ and $a_1, a_2, ... a_j$ be the labels appearing on the path. Then $a_1.a_2. ... a_j$ is a label path. Similar to single labels, we define $T^{-1}(a_1.a_2. ... a_j)$ to be the elements in $dom(T)$, on which $a_1.a_2. ... a_j$ appears. This notation will be used in the subsequent discussion.

Operations based on relaxed compatibility

In the above, we defined a set of operations based on a "strict" compatibility. For the practice purpose, however, more relaxed compatibilities should be considered. To this end, we define the concept of tree isomorphism.

Definition 3.5 (*tree isomorphism*) Let T_1 and T_2 be two trees. T_1 and T_2 are said to be *isomorphic*, denoted $T_1 \cong T_2$ if there exists an one-to-one mapping from the nodes of T_1 to the nodes of T_2 that preserves labels and tree structure.

(The algorithm for the tree isomorphism is available and the check to see whether two trees are isomorphic can always be done in linear time [12, 13].)

This definition leads directly to the following result.

Proposition 3.1 If T_1 and T_2 are isomorphic, then we can always transform T_2 into T_1 by applying a series of label-renaming operations on T_2 or *vice versa.*

Let $T[b_1 \leftarrow a_1]$, $T[b_2 \leftarrow a_2]$, ..., $T[b_n \leftarrow a_n]$ be a sequence of label-renaming operations. We write $T[B \leftarrow A]$ for it for short, where $B = \{b_1, b_2, ..., b_n\}$ and $A = \{a_1, a_2, ..., a_n\}$.

Based on this concept, we define a relaxed compatibility as follows.

Definition 3.6 (*compatible up to isomorphism*) Let T_1 and T_2 be two document trees. We say that T_1 and T_2 are compatible up to isomorphism if and only if there exists some tree else T such that $T_1 \cong T$ and T is compatible with T_2. (Note that if T_1 and T_2 are compatible, they are also compatible up to isomorphism.)

In terms of this notion, we redefine the first three basic operations discussed above.

- tree-union$_{iso}$ ($\oplus_{iso}$)

Let T_1 and T_2 be compatible upon isomorphism. Let $T_1[B \leftarrow A]$ be the label-renaming sequence transforming T_2 into a tree compatible to T_1. The union of them ($T_1 \oplus_{iso} T_2$) is defined by

(i) $dom(T_1 \oplus T_2) = dom(T_1) \cup dom(T_2)$;
(ii) $\forall x \in dom(T_1 \oplus T_2)$,

$(T_1 \oplus T_2)(x) =$

$$\begin{cases} T_1(x), & \text{if } x \in dom(T_1)/\mathrm{fr}(T_1) \text{ or } x \in \mathrm{fr}(T_1) \text{ but } x\text{'s parent is not compatible with any terminal's parent in } T_2 \\ T_2[B \leftarrow A](x), & \text{if } x \in dom(T_2)/\mathrm{fr}(T_2) \text{ or } x \in \mathrm{fr}(T_2) \text{ but } x\text{'s parent is not compatible with any terminal's parent in } T_1 \\ \{T_1(x)\} \cup \{T_2[B \leftarrow A](x)\}, & \text{otherwise} \end{cases}$$

In a similar way, we can define the corresponding "intersection" and "symmetric-difference":

- tree-intersection$_{iso}$ ($\bullet_{iso}$),
- tree-symmetric-difference$_{iso}$ ($\sim_{iso}$).

Now we turn to an interesting problem to derive a DTD from a set of documents. We have this problem by organizing a set of documents (got, for example, through the World-Wild Web) into a database. We need such a DTD to govern the document loading process, such as the schema establishment for accommodating them, the index construction on some document elements, etc.

Example 3.3 Consider a set of documents T_1, T_2, ..., T_n. We construct a new document T:

$$T = T_1 \oplus_{iso} T_2 \oplus_{iso} ... \oplus_{iso} T_n.$$

In T, each terminal node is of the form: $\{c_1, c_2, ..., c_n\}$, where each c_i is either an empty string or the label (value) of a terminal node of T_i. By a simple analysis, the data type for c_i (i = 1, 2, ..., n) can be determined, say #PCDATA. Then, substitute #PCDATA for $\{c_1, c_2, ..., c_n\}$ in T. We can replace each terminal node of T with the corresponding data type in this way to obtain a new tree T' which can be employed as an approximate DTD for T_i (i = 1, 2, ..., n).

We conclude this section with another example involving recursion.

Example 3.4 As is well known, the recursion is beyond the expressiveness of the relational algebra [20]. The same is also true for the document algebra. But we can develop a similar way as the deductive database to handle this problem [4]. Consider the DTD piece shown in Fig. 15(a), in which each "segment" possesses a "title", some "paragraphs" and also "segements" recursively.

```
<!ELEMENT   doc   - - (seg*)>
<!ELEMENT   seg   - - (title, para*, seg*)>
<!ELEMENT   title - - (#PCDATA)>
<!ELEMENT   para  - - (#PCDATA)>
```

(a)

```
<!ELEMENT   doc   - - (seg*)>
<!ELEMENT   seg   - - (title, para*, (seg | topic)*)>
<!ELEMENT   topic - - (title, para*)>
<!ELEMENT   title - - (#PCDATA)>
<!ELEMENT   para  - - (#PCDATA)>
```

(b)

Fig. 15. DTD transformation

We want to rename the lowest-level "segments" as "topics". For the DTD tree, it is very simple, which can be done as follows:

$T[T(1) \leftarrow$ "seg-(title, para*, (seg | topic)*)"];
(*renaming*)
$T(1.4)T_{sub}(T^{-1}$(doc.seg));
(*union of T and the subtree (rooted at "doc.seg") at node address "1.4".*)
$T[T(1.4) \leftarrow$ "topic-(title, para*)"];
(*renaming*)
$T[\sim T^{-1}$(doc.seg.topic.seg)].
(*removing the node labeled with "seg" below the node labeled with "topic"*)

The resulting DTD is shown in Fig. 15(b).

However, for the documents conforming to this DTD, a more complicated method has to be employed for the corresponding transformation. We need a predicate *Pterminal*(x) to check whether a node is a parent of some terminal node. In addition, we regard $T_{sub}(\alpha)$ as a predicate to quantify subtrees. When the subtree rooted at α exists, it evaluates to *true*. Otherwise, it evaluates to *false*. We construct the following deductive rule:

rule: $T_{sub}(T^{-1}(x.\text{seg})) :- T_{sub}(T^{-1}(x))$, seg $\in T(T^{-1}(x))$, $\neg$*Pterminal*$(T^{-1}(x.\text{seg}))$.

The rule means that if x is a label path leading to a subtree ($T_{sub}(T^{-1}(x))$), the root of the subtree is labeled with "seg" (seg $\in T(T^{-1}(x))$) and at the same time $T^{-1}(x.\text{seg})$ is not a parent of some terminal node ($\neg$*Pterminal*$(T^{-1}(x.\text{seg}))$), then $T_{sub}(T^{-1}(x.\text{seg}))$ evaluates to *true*.

This rule is recursive. To finish the above task for the document transformation, we execute the following procedure:

A := {doc.seg}; $\Delta := \emptyset$;
repeat
  for each $x \in A$ **do**
    {evaluate *rule*;
     $\Delta := \Delta \cup$ answers;}
  if $\Delta \neq \emptyset$ **then**
    {$A := \Delta$; $\Delta := \emptyset$;} (*note that the answers are a set of label paths; only the answers last

produced are kept.*)
until no more new answers
for each $d \in A$ **do**
{
$T[T(d) \leftarrow \text{topic}]$; (*renaming*)
}

In the above, all the operations for the transformation of DTDs and single document trees were discussed. To manipulate document sets, more operations should be established. In fact, three operations analogous to the relational algebra can be defined. They are projection, selection and join. Different from the relational algebra, some tree specific operators are involved such as *tree matching* and *tree inclusion*, which are especially useful for web recognition [5, 18, 19].

Definition 1 Let S_D be a set of documents conforming to a certain DTD. The projection of S_D onto a subset X of DTD (recall that DTD is a set of pairs; see Section 2) such that $dom(X)$ is interval closed, written as $\pi_X(S_D)$, is the document pieces defined as follows

$A_i = \{T_{i\,sub}(\alpha_{ij}) \mid T_i \in S_D; \exists\, d \in dom(X) \text{ such that } \alpha_{ij} d \in dom(\text{DTD})\}$,
$\pi_X(S_D) = \{A_i \mid i = 1, ..., n\}$,

where n is the number of documents in S_D.

Definition 2 (*selection*) The selection of a document set S_D under a formula F is the subset of S_D, written as $\sigma_F(S_D)$, consisting of all those documents T of S_D such that each such document satisfies F, i.e.,

$$\sigma_F(S_D) = \{T \mid T \in S_D \text{ and } T \text{ satisfies } F\}.$$

Definition 3 (*join*) Let S_1 and S_2 be two document sets conforming to DTD_1 and DTD_2, respectively. Let A be a subtree of $DTD_1$and B a subtree of DTD_2. The join of S_1 and S_2 on A and B, written as $S_1[A \varphi B]S_2$, is the set consisting of every triple of the form (n_{12}, T_1, T_2) representing a new tree with n_{12} being the root and T_1 and T_2 being n_{12}'s left and right subtrees, respectively, where T_1 is in S_1, T_2 is in S_2 such that $T_1 \varphi T_2$ evaluates to *true*. n_{12} is a virtual node used as the root of the new document containing T_1 and T_2, i.e.,

$$S_1[A \varphi B]S_2 = \{(n_{12}, T_1, T_2) \mid T_1 \text{ in } S_1, T_2 \text{ in } S_2 \text{ and } T_1 \varphi T_2\}.$$

4. Conclusion

In this paper, a document algebra is proposed. Based on the tree domain theory, a group of operations for the tree transformation is formally defined, which can be utilized to change a DTD or a document structure and derive a DTD from the existing documents. Another group of operations is developed by equating a subtree structure from a DTD to an attribute from a relation schema. This group can be used to perform "projection", "selection" and "join" as well as the other set-oriented operations on document sets.

References

[1] K. Bharat and A.Z. Broder, Mirror, Mirror, on the web: A study of host pairs with replicated content, in *Proc. of 8th Int. Conf. on World Wide Web (WWW'99)*, May 1999.

[2] A.Z. Broder, S.C. Glassman and M.S. Manasse, Syntactic clustering of the web, in *Proc. of 6th Int. World Wide Web Conference*, April 1997, pp. 391-404.

[3] V. Christophides, S. Abiteboul, S. Clut and M. Scholl, "From structured documents to novel query facilities," *SIGMOD Record* **23:2** (1994), pp 313 -324.

[4] Y. Chen, A Bottom-up Query Evaluation Method for Stratified Databases, in: *Proc. of 9th Int. Conf. on Data Engineering*, Vienna, Austria: IEEE, April 1993, pp. 568-575.

[5] Y. Chen, T. Liu. and P. Sorenson, Personal Web Space, accepted by 4th Intl. Workshop on Multimedia Network Systems and Applications held together with ICDCS2002, Vienna, Austria, July 2-5, 2002.

[6] E.F. Codd, "Relational completness of data base sublanguage," *ibid*, 1972, pp. 65 - 98.

[7] L. Colby, V. Gucht and D. Saxton, "Concepts for modeling and querying list-structured data," Information Processing & Managements **30:5** (1994), pp. 687 - 709.

[8] J. Cho, N. Shivakumar, H. Garcia-Molina, "Finding replicated web collections," http://dbpubs.stafford.edu/pub/1999-64.

[9] D. Giammarrest, S. Mantaci, F. Migosi and A. Restivo, "Periodicities on trees," *J. of Theoretical Computer Science*, Vol. 205, No. 15, July. 1998, pp. 145 - 181.

[10] M. Gyssens, J. Paredans and D.V. Gucht, "A Grammar-Based Approach Unifying Hierarchical Data Models," *SIAM J. Comput*., Vol. 23, No. 6, Dec. 1994, pp. 1093 - 1137.

[11] S. Ginsburg and X. Wang, "Pattern matching by rs-operations: towards a unified approach to quering squenced data," *Proc. of the 11th ACM SIGACT-SIGMOD Symposium on Principles of Database System*, San Diego, CA, 1992, pp. 293 - 300.

[12] J. Hopcroft and R. Tarjan, "Isomorphism of planar graphs," In *Complexity of Compu-tation*, R. Miller and J. Thatcher, Eds., Plenum Press, New York, 1972, pp. 143 - 150.

[13] J. Hopcroft and J. Wong, "Linear time algorithm for isomorphism of planar graphs," *Proc. of 6th Annual ACM Symp. on Theory of Computing*, Seattle, Wash., 1974, pp. 172 -184.

[14] R. Kaivola, "Axiomatising extended computation tree logic," *J. of Theoretical Computer Science*, Vol. 190, No. 1, Jan. 1998, pp. 41 - 60.

[15] M. Murata, "Transformation of documents and schemas by patterns and contextual conditions," *Lecture Notes in Computer Science* **1293** (1997), pp. 153 - 169.

[16] J. W. Thatcher, "Tree automata: An informal survey," in A.V. Aho editor, *Currents in the theory of computing*, Prentice Hall, 1987, pp. 143 - 172.

[17] G. Salton, *Introduction to modern information retrieval*, McGraw-Hill, New York, 1983.

[18] N. Shivalumar and H. Garcia-Molina, SCAM: a copy detection mechanism for digital documents, in *proc. of 2nd Int. Conf. on Theory and Practice of Digital Libraries (DL'95)*, Austin, Texas, June 1995.

[19] N. Shivalumar and H. Garcia-Molina, Building a scalable and accurate copy detection mechanism, in *proc. of 1st Int. Conf. on Digital Libraries (DL'96)*, Bethesda, Maryland, March 1996.

[20] J.D. Ullman, "*Principles of databases and knowledge-base systems*," Computer science press Inc., Maryland, 1988.

[21] C.-C. Yang, "*Relational Databases*," Prentice-Hall, New Jersey, 1986.